Dynamic Module Library Generation for FPGA-based Run-Time Reconfigurable Systems

John Kipp Bowen

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

Master of Science
in
Computer Engineering

Dr. Cameron Patterson, Chair
Dr. Mark Jones
Dr. Thomas Martin

January 18, 2008
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: FPGA, Partial Reconfiguration, Module Library, Run-time Reconfiguration
Copyright ©2008, John Kipp Bowen. All Rights Reserved.

Dynamic Module Library Generation for FPGA-based
Run-Time Reconfigurable Systems

John Kipp Bowen

ABSTRACT

Modern Field Programmable Gate Arrays (FPGAs) can implement entire run-time reconfigurable systems using partial reconfiguration. Module-based run-time reconfiguration permits the construction of custom applications at run-time using pre-compiled Intellectual Property (IP) from a module library. The need for both flexible module placement and custom inter-module communication is mostly ignored by existing modular run-time reconfiguration approaches and few existing tool flows for module generation address the need for automation. This thesis introduces an automated compile-time tool flow for generating dynamic modules that allow flexible run-time placement and communication synthesis.

Acknowledgments

I would like to first thank Dr. Cameron Patterson, my academic advisor and committee chair, for guiding me throughout my time in graduate school and offering me the opportunity to join the Configurable Computing Machines Lab at Virginia Tech. The knowledge and experience that I gained during my time in the CCM Lab have contributed more to my education than I could have ever imagined. I would also like to thank Dr. Patterson for his unparalleled ability to edit documents, which made reading this thesis bearable for the next few people I'd like to thank.

In addition to my committee chair, I would like to thank Dr. Mark Jones and Dr. Tom Martin for serving on my committee and Dr. Peter Athanas for serving as an official proxy during my defense. All three have taught me much during my undergraduate and graduate studies at Virginia Tech and have provided a relaxed – and sometimes entertaining – learning environment in the CCM Lab, with a little help from some minibikes and a Foosball table.

Dr. Patterson and Dr. Athanas have also provided advice and guidance throughout the Wires on Demand project at Virginia Tech, along with Jonathan Graf, Mark Bucciero, and John Hallman of Luna Innovations. This work would also not be possible without the other members of the Wires on Demand team: Tim Dunham, Justin Rice, Matt Shelburne, and Jorge Surís. Thank you all for your help in each step of my research.

Thanks to all of my friends in the CCM Lab; you have made my two years in graduate school

an amazing experience. I wish you all the best of luck in your future endeavors.

My parents and my brother have always supported me in everything that I do. After I graduated from Virginia Tech with a Bachelor's Degree, they motivated me to continue in graduate school, which ultimately led to this thesis. Thank you Jack and Carroll, and thank you Chris for your continued love and support; none of this would have been possible without you.

Lastly, thanks to all the friends I've made during my many semesters and summers at Virginia Tech, and thanks for making my six and a half years in Blacksburg unforgettable. Go Hokies!

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

Contents

List of Figures

List of Tables

Glossary

AM Amplitude Modulation

ADC Analog to Digital Converter

ASIC Application Specific Integrated Circuit

BRAM Block RAM

CLB Configurable Logic Block

DCM Digital Clock Manager

DCT Discrete Cosine Transform

DSP Digital Signal Processor

EDIF Electronic Data Interchange Format

FFT Fast Fourier Transform

FLEX Fast Lexical Analyzer

FPGA Field Programmable Gate Array

GPP General Purpose Processor

HDL Hardware Design Language

ICAP Internal Configuration Access Port

IOB Input/Output Block

IP Intellectual Property

LUT Look-Up Table

NCD Xilinx Circuit Description

NGC Xilinx Netlist

NGD Native Generic Database

NoC Network on a Chip

PAR Place And Route

PC Personal Computer

PR Partial Reconfiguration

RAM Random Access Memory

ROM Read Only Memory

RTR Run-time Reconfiguration

SDR Software Defined Radio

SoC System on a Chip

UCF User Constraints File

VHDL VHSIC Hardware Description Language

V4 Virtex-4

WoD Wires on Demand

XML Extensible Markup Language

XST Xilinx Synthesis Technology

Chapter 1

Introduction

1.1 Overview

Field Programmable Gate Array (FPGA) devices are large arrays of uncommitted logic elements and interconnect resources used for processing digital information. The function of an FPGA is not decided at the time of manufacture, rather it is programmed by the end-user. FPGAs are no longer solely used for prototyping as they were initially, but have been used in final products for years. Advances in FPGAs have continued to where FPGAs are now used to implement entire systems. With the recent advances in FPGAs and the likelihood that FPGAs will continue to become larger, faster, and more complex, FPGAs have the potential to improve price, performance, and power efficiency of systems currently designed using other means. Through the use of reconfiguration, smaller FPGAs are used to reduce price. Performance and perhaps power efficiency is improved by implementing logic specialized for a specific application.

The most suitable applications for FPGAs make use of the massive parallelism potential inherent in the architecture. Applications that take advantage of parallelism include cryp-

tography applications such as encryption/decryption [1][2][3] and cipher breaking [4] as well as digital signal processing applications such as Software Defined Radio (SDR) [5], Fast Fourier Transform (FFT) [6], and Discrete Cosine Transform (DCT) [7]. Other examples of applications that are well suited for implementation on FPGAs are hardware emulation [8], prototyping [9], matrix multiplication [10], among others.

FPGA design requires less effort than Application Specific Integrated Circuit (ASIC) design, primarily because the FPGA hardware has already been tested by the manufacturer. However, the effort required for FPGA design greatly exceeds software design. Although software may not deliver the same performance and power efficiency advantages of an FPGA, the design overhead makes software the preferred solution if it can meet the performance constraints.

Some FPGAs have the unique capability to change device configuration during a computation task, which is referred to as Run-time Reconfiguration (RTR). Figure 1.1 illustrates this capability by showing that a portion of the device changes configuration while the remainder of the device continues regular operation. Region A and Region B in Figure 1.1 continue to operate while Region C is reconfigured to perform a different function. The current means of RTR presented by Xilinx is complicated, cumbersome, and inflexible. The use of Partial Reconfiguration (PR) tools developed by Xilinx requires detailed knowledge of the FPGA architecture and there is no support for modeling or simulation. Verification can only be done by testing the design on actual hardware, which can be very difficult to debug if there are errors. The Xilinx PR flow uses a "cookie-cutter" technique by creating one or many predefined "slot(s)" where several different modules may be placed. In Figure 1.1, Region B and Region C are both examples of predefined reconfigurable slots. Each slot has a static communication interface implemented using Xilinx Bus Macros. The responsibility of generating inter-module communication falls on the designer. Communication between modules is static and may use buses.

Inter-module communication needs to be specialized to make full use of the reconfiguration

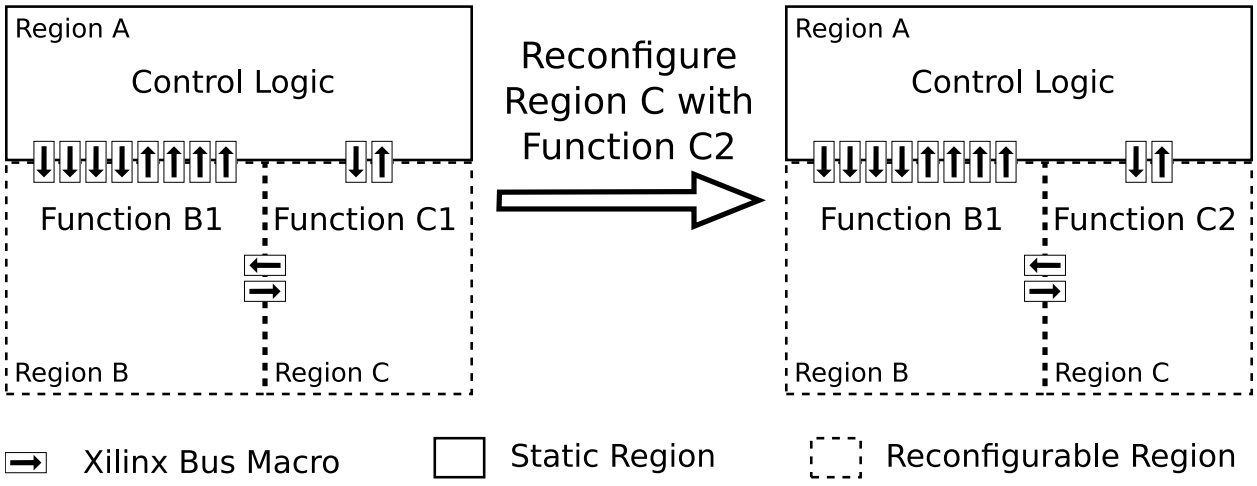

Figure 1.1: Run-Time Reconfiguration

ability inherent in FPGA architecture. High performance systems require efficient communication that are hard to provide using the Xilinx PR flow since the inter-module communication is static. RTR should be used to provide optimal inter-module communication, custom generated for each module pair. Buses, on-chip networks, and crossbar switches are alternatives to custom, point-to-point communication; however each of these introduce unnecessary power, latency, and/or area overhead. The majority of the area of an FPGA is wires. To ignore the ability to reconfigure these wires is eliminating an entire dimension of potential optimization. Most of the wires are left unused in a typical design, which leaves many wires that could be used for custom, point-to-point communication between dynamic modules.

Automatic generation of custom interconnect between cores is referred to as communication synthesis [11]. A flow for run-time communication synthesis is needed for RTR, but is missing. If a flow existed for RTR, dynamic module placement and communication synthesis would occur at run-time. Cores used in the system have to be specially prepared to provide the run-time flexibility, however this could be done at compile-time.

1.2 Contributions

This thesis describes a compile-time tool flow that supports a flexible run-time system, called Wires on Demand (WoD), that manages the placement and routing of dynamic Intellectual Property (IP) modules within a sandbox region [12]. The compile-time tools include a template file, a preprocessor, a wrapper structure, a module description file, a bitstream post-processing tool, and an automated build flow.

The template file is used to describe an IP core and its associated ports, and is created in a language developed specifically for describing all the necessary data used by the preprocessor. At compile-time, the designer generates a template file for each core that requires dynamic behavior at run-time. The preprocessor reads the template file and IP core source files prior to the build process, generates Verilog Hardware Design Language (HDL) source files of the wrapper structure, Extensible Markup Language (XML) description files necessary for both the postprocessor and the run-time tools, as well as scripts that automate portions of the build process. The wrapper structure is necessary to allow dynamic placement and custom inter-module routing at run-time. Additional logic is implemented in the wrapper structure to assist the communication between modules and help with testing and debugging cores. The source files of the wrapper structure are packaged with the original, unchanged HDL source for the IP core.

The automated build flow creates an appropriate folder structure for the build process, invokes the preprocessor, invokes the standard Xilinx build tools and the custom scripts generated by the preprocessor, invokes the postprocessor, and produces the final bitstream and XML description files necessary for the module library. The designer starts the automated build flow once the template file and IP core source files are prepared, and then waits for the build process to complete without further interaction. The postprocessor uses a bitstream toolbox to modify the bitstreams generated by the Xilinx tools. The bitstream toolbox allows the manipulation of bitstreams without the need of mainstream tools, and was

developed separately from the other compile-time tools described in this thesis. The module library stores RTR-ready bitstreams and XML description files for use in the run-time system.

1.3 Thesis Organization

This thesis is organized into six chapters. Chapter 1 presents an overview of the problem and the contributions made by the author. Chapter 2 discusses background information and gives an overview of related work. Chapter 3 introduces a flexible run-time reconfigurable computing system and an example of the compile-time tool flow. The next chapter describes the compile-time tool flow in detail, including automation tools, the template file format, the preprocessor, and the postprocessor. Chapter 6 presents the results of the flow using a test application. The final chapter draws conclusions and describes future work.

Chapter 2

Background

2.1 Overview

This chapter presents background information as well as previous efforts in research areas that pertain to this work. The background begins with a definition of an FPGA and a description of FPGA capabilities, then discusses PR and RTR. Finally, existing approaches to RTR are presented.

2.2 Field Programmable Gate Arrays

As mentioned in Section 1.1, FPGAs are useful for implementing many applications from cryptography and digital signal processing to hardware emulation and ASIC prototyping. Applications that perform several tasks in parallel or make use of the configurability of FPGAs benefit the most from an FPGA implementation. The alternatives to using FPGAs range from fully software solutions to custom ASIC design, or some mix of the two. The factors that usually end up determining which technology is chosen for the solution are price,

Technology	**Unit Price**	**Performance**	**Power Efficiency**	**Run-time Adaptability**	**Design Effort**
Custom ASIC	B+	A+	A+	D	F
Commodity DSP or GPP	A	C	C	A+	A
Multi-core Processors	B	B	B-	A	B
SoC*	B+	B+	B	C+	D
Statically Configured FPGAs	B-	B+	C	C	A-
Partially Reconfigured FPGAs	B+	A-	C+	A-	C

*SoC = Processor(s) + Function Accelerator(s)

Table 2.1: Technology report card

performance, power efficiency, adaptability, and design effort (time and cost) required to use the technology. A comparison of several technologies based on the metrics listed above is shown in Table 2.1 [12]. When evaluating the tradeoffs, it is hard to assign an overall winner.

When compared to ASICs, FPGAs have advantages including reusability, debugging on the target hardware, in-field updates, and rapid prototyping. FPGAs also have shorter time to market and require less design effort. Another key feature of FPGAs not available with ASICs is dynamic and partial reconfiguration (described in Section 2.4). ASICs, however, typically have the advantage of increased performance and power efficiency. When compared to fully software (General Purpose Processor (GPP) or Digital Signal Processor (DSP)) solutions, reusability, debugging on the target hardware, in-field updates, and rapid prototyping are capabilities shared by both technologies. Software solutions have shorter time to market and require less design effort than FPGAs. Software solutions are also generally less expensive to develop than FPGA solutions. However, software solutions cannot provide the same level of performance or throughput as FPGAs, which may be required by some applications.

The two leading manufacturers of FPGAs are Xilinx and Altera [13]. Xilinx introduced

the first commercially available FPGA in 1985 [14]. There are currently two series of Xilinx FPGAs: the low-cost Spartan family and the high-performance Virtex family. Xilinx FPGAs support PR with an approved design flow, however Altera does not have a design flow for PR at this time [15]. Since the work presented in this thesis requires the use of PR, Xilinx's Virtex-4 (V4) family of FPGAs is used exclusively.

2.3 FPGA Architecture

FPGA devices are large arrays of uncommitted logic elements and interconnect resources, as mentioned in Section 1.1. The basic building block of a Xilinx V4 FPGA is called a Configurable Logic Block (CLB). In addition to CLBs, Xilinx V4 FPGAs are composed of Block RAM (BRAM), DSP, Digital Clock Manager (DCM), and Input/Output Block (IOB) elements. Switch matrices inside each block provide connections between blocks. The general layout of a Xilinx V4 FPGA is shown in Figure 2.1.

CLBs are used to implement basic logic functions. A V4 CLB is composed of four slices and a switch matrix, as shown in Figure 2.2. Each slice is composed of two flip-flops and two LUTs as well as muxes, arithmetic logic, and carry logic. A V4 Look-Up Table (LUT) has four input signals and can be used to implement any arbitrary four input Boolean function. LUTs can also be configured as Random Access Memory (RAM), Read Only Memory (ROM), or shift registers [16].

V4 BRAMs store 18K bits of data and serve as large memory storage for FPGA designs. DSPs implement a multiply accumulate function to improve signal processing performance. An IOB is connected to each pin on the device. IOBs are generally located along the outer edge of an FPGA, however the V4 family is different in that it also has a column of IOBs down the center of the device. DCMs provide several clock management features such as clock deskew, frequency synthesis, and phase shifting. All elements of an FPGA have identical

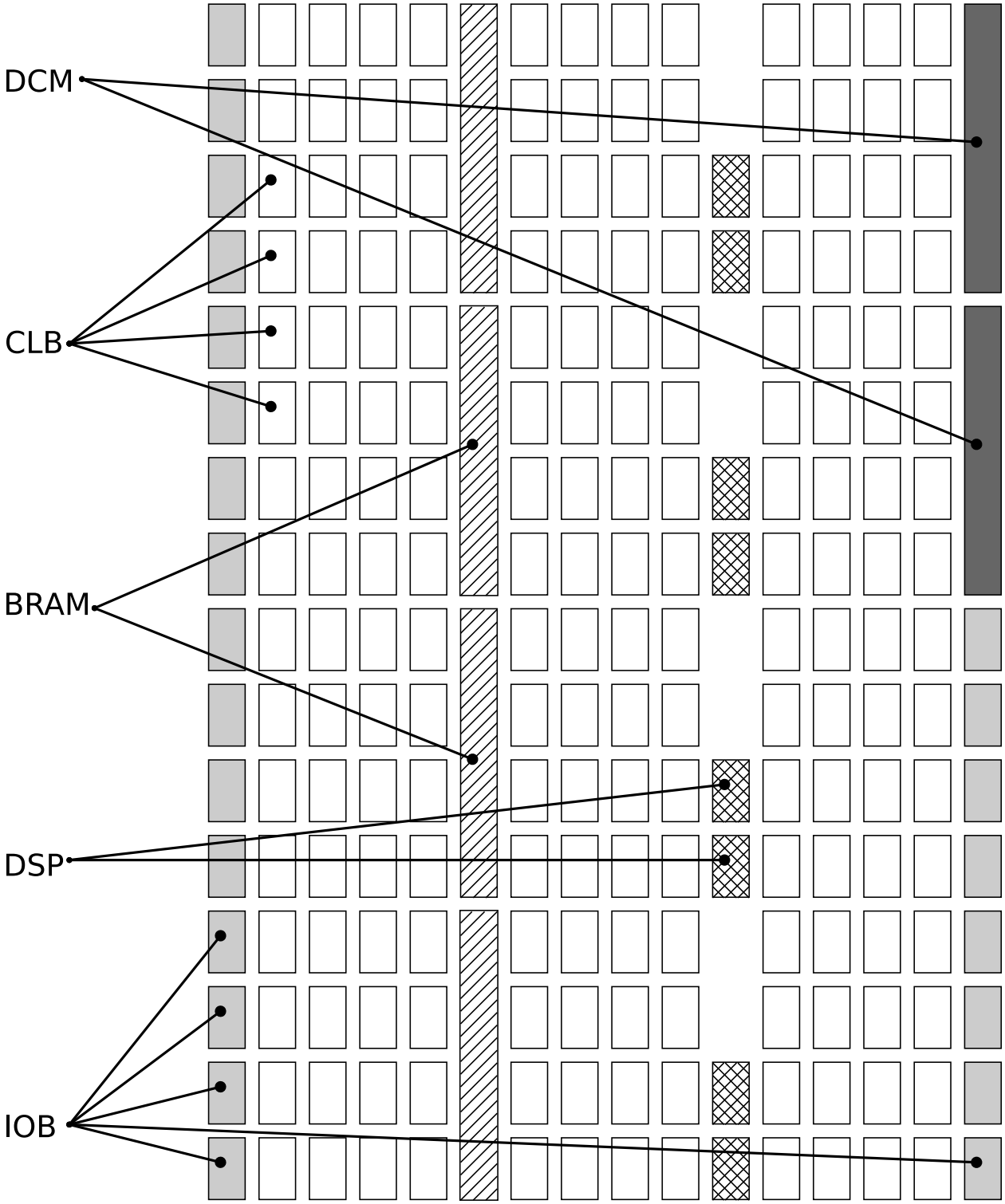

Figure 2.1: Xilinx XC4VLX15 Architecture (Upper-left Clock Region)

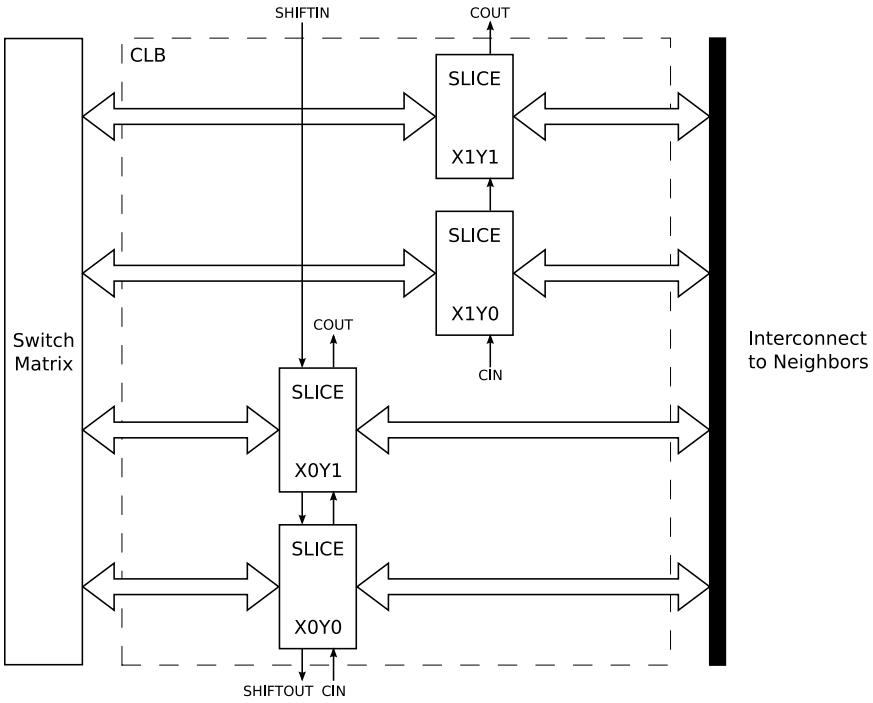

Figure 2.2: Xilinx Virtex-4 Configurable Logic Block

routing interfaces through the use of switch matrices, which allows a high level of scalability and flexibility for device layouts. For example, Xilinx V4 FPGAs range from 64 by 24 (row by column) to 192 by 116 matrices of CLBs with varying numbers of BRAMs and DSPs interspersed among the CLBs[17].

2.4 FPGA Configuration

FPGAs can be configured to perform a specific function by loading configuration data called a bitstream. Configuration may be either static or dynamic. Static configuration usually occurs when the FPGA is powered on or reset; dynamic configuration differs in that the FPGA may be loaded with a new bitstream to adapt to system requirements. Some FPGAs also support PR, in which the FPGA is loaded with a partial bitstream to change the configuration of a portion of the FPGA without affecting the remainder of the device. Dynamic

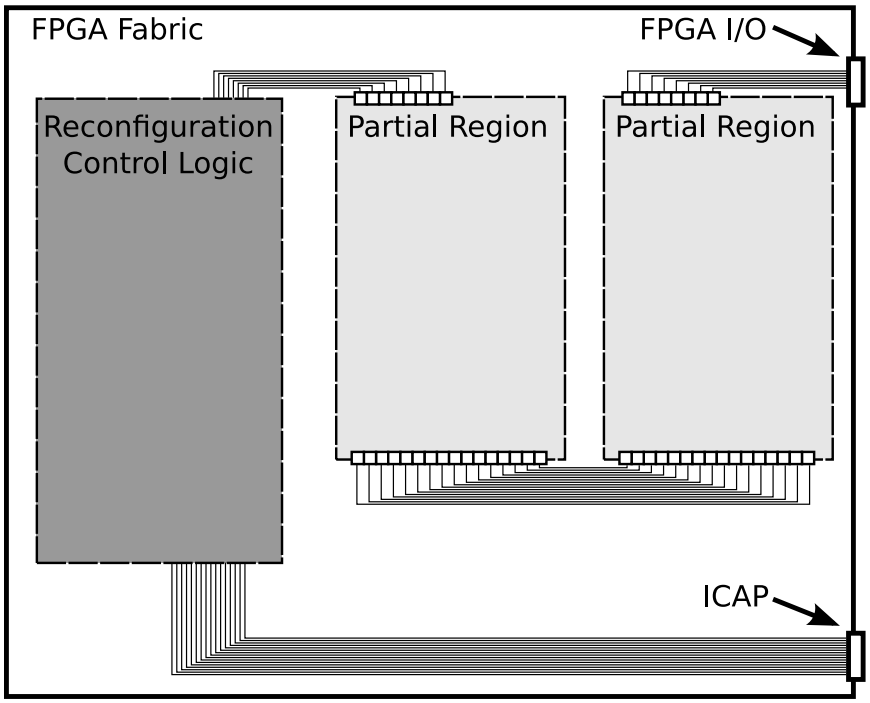

Figure 2.3: FPGA layout for self reconfiguration

PR is the focus in this thesis.

RTR dynamically reconfigures an FPGA without disturbing all of its operating functions. PR is especially useful in RTR when the logic controlling the reconfiguration is implemented on the FPGA itself, such as in Figure 2.3. By using PR, the new configuration is completely isolated from the logic performing the reconfiguration. Systems designed in this manner can make use of the Internal Configuration Access Port (ICAP) for loading the partial bitstreams. The ICAP eliminates the need for the partial bitstream data to exit the FPGA during reconfiguration. This is useful for both security reasons and performance.

2.4.1 Advantages and Disadvantages to Run-time Reconfiguration

RTR provides several benefits. Through the use of RTR, it is possible to implement a system using fewer FPGA resources than an equivalent system that does not use RTR because entire

IP cores can be removed when not in use to make room for other IP cores that may be needed. FPGA area reuse reduces the cost and size of static systems, and expands capabilities by offering a wider range of available IP cores. Performance is increased through the use of hardware to implement functions that would have previously been implemented in software. By being able to support a wider range of specialized hardware IP cores, power consumption can also be reduced. RTR can also be used to update or fix current FPGA designs similar to software patches and updates [18]. This can extend the lifespan of hardware, which will reduce long term costs for the user.

RTR also has some disadvantages. Perhaps the most notable is the cumbersome, difficult, and error-prone design process. The current tools provided by Xilinx for PR fall short in many areas. Xilinx's PR flow requires many manual steps and an in-depth knowledge of the FPGA architecture [19]. The interfaces or communication paths between modules must also be designed by the user and the Xilinx PR flow does not provide a means for debugging or testing designs. Despite this, many research groups still make use of the Xilinx PR flow for implementing RTR systems, which is discussed in more depth later in this section. A disadvantage pertaining to all RTR flows is the added time required for loading reconfiguration data to an FPGA. Some proposed RTR systems try to reduce the effect of this by switching tasks during reconfiguration [20], however the FPGA's ability to perform tasks in parallel is reduced.

2.4.2 Xilinx Partial Reconfiguration Flow

The Xilinx PR flow is complicated and difficult to summarize. The purpose of this section is to provide a general understanding of the flow and the responsibilities of the user. For a complete description of the Xilinx PR flow, see [19]. To better understand the PR flow, it is important to understand Xilinx's standard design flow shown in Figure 2.4. Figure 2.4 also lists the Xilinx tool associated with each step, although there are several alternative tools

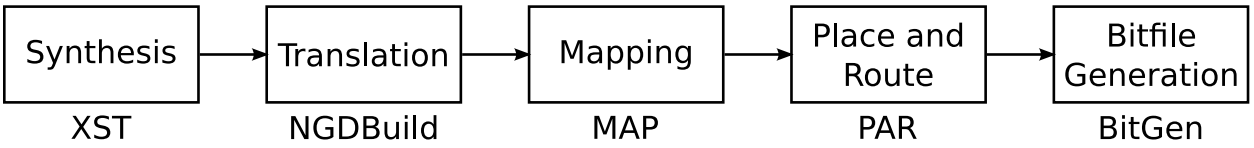

Figure 2.4: Xilinx Design Flow

for synthesis.

First, a hardware designer implements a design using a HDL such as Verilog or VHSIC Hardware Description Language (VHDL). The HDL is converted to a netlist through a process called synthesis. Synthesis starts from a high level of logic abstraction (Verilog or VHDL) and automatically creates a lower level of logic abstraction (netlist) targeting a library of primitives [21]. XST is a synthesis program available from Xilinx that creates a netlist in the form of a Xilinx Netlist (NGC) file. However, other synthesis tools produce netlists in different formats such as Electronic Data Interchange Format (EDIF). The netlist is converted to a logical design in the form of a Native Generic Database (NGD) file using NGDBuild. NGDBuild is a translation tool that creates Xilinx format files from other formats [21]. NGDBuild also merges all netlist files into a single NGD file.

A Xilinx Circuit Description (NCD) file is produced by MAP, which maps the design to a specific device. Mapping is the process of assigning logic elements of a design to specific physical elements that actually implement logic functions in a device [21]. Once mapping is completed, the design is placed and routed using PAR. Placement is the process of assigning physical device cell locations to the logic in a design and routing is the process of assigning logical nets to physical wire segments in the FPGA that interconnect logic cells [21]. PAR modifies the original NCD file with placement and routing information. PAR also generates a timing report to ensure that the design will meet the timing constraints. The final step in the standard Xilinx design flow is to have BitGen produce a bitstream file to configure an FPGA.

It is also necessary to understand the complications introduced when using PR. The Xilinx

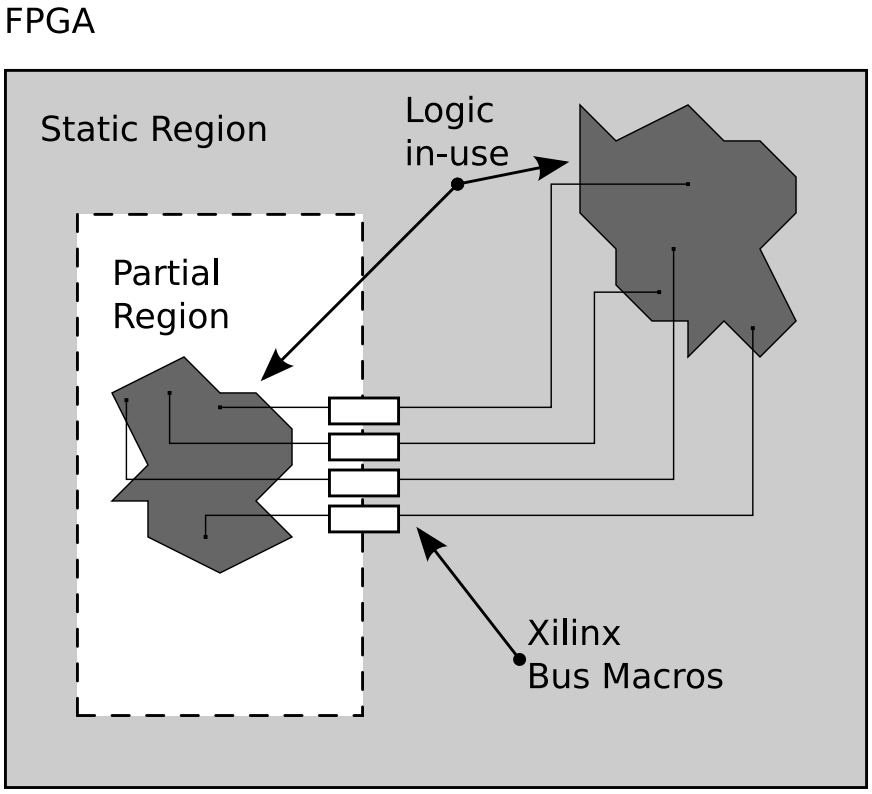

Figure 2.5: FPGA with Partial Region and Xilinx Bus Macros

PR flow requires that specific locations be defined where PR will take place, as shown in Figure 2.5. These locations, called partial regions here, are where a variety of modules can be placed during PR. When partially reconfiguring an FPGA it is important that the static system build is not using any resources in the partial region required by the partial module. It is also important that the signals that communicate with parts of the FPGA external to the partial region are connected properly using Xilinx Bus Macros. Any communication crossing the boundary of the partial region must pass through Bus Macros on the partial region boundary. Bus Macros are used to ensure that the signals crossing the PR boundary are consistent between the static design and each PR design.

The Xilinx PR flow uses many of the same tools as the standard flow, however some of the tools have been modified specifically for PR. The PR flow also starts with a hardware designer implementing a design using a HDL, as shown in Figure 2.6. It is presumed that

the design contains a base system that will remain static and a specified area where several different modules will be loaded to change the behavior of the system. The top-level HDL module must contain all global logic such as I/O signals, global clocks, and DCMs. Also, the top-level HDL module will have black box instantiations of the base system, the PR module, and the bus macros. The name of each PR module must match its file name for the Xilinx tools to work properly. Xilinx suggests a folder structure to keep the files separate [19]. There are synthesis constraints that must also be applied to the HDL to ensure that each module of the design remains distinct.

As mentioned previously, any communication between the static system build and the partially reconfigurable module must be carefully managed. Xilinx achieves this through the use of hard macros called Bus Macros that reserve specific routing that crosses the partial region boundary. This is necessary to ensure that partial modules communication signals connect properly to the static logic signals. Bus Macros must be placed manually by the designer in the HDL before synthesis, and must appear in the top-level HDL module. Since the partial region uses Bus Macros for all communication to and from the partial region, it is necessary for all partial modules intended for that region have the same port interface. At this point, the HDL design can be synthesized using XST. Each partial module will need to be synthesized separately to produce a separate netlist for each module.

The next step is to place a variety of constraints for the design in a User Constraints File (UCF). These constraints include assigning different groups for the partial module and the static logic, and setting a specific rectangular area and location for the partial region. Each Bus Macro must also be placed properly such that it crosses the partial region boundary. Guidelines and additional constraints are available [19]. Once the constraints are in place, it is advised that the design be tested without the PR region set as reconfigurable in the UCF. The purpose of this is to make sure the modified design still works before introducing the additional complications of PR. Once the design has been confirmed to work properly, the base static design is implemented with tools similar to the standard Xilinx translation,

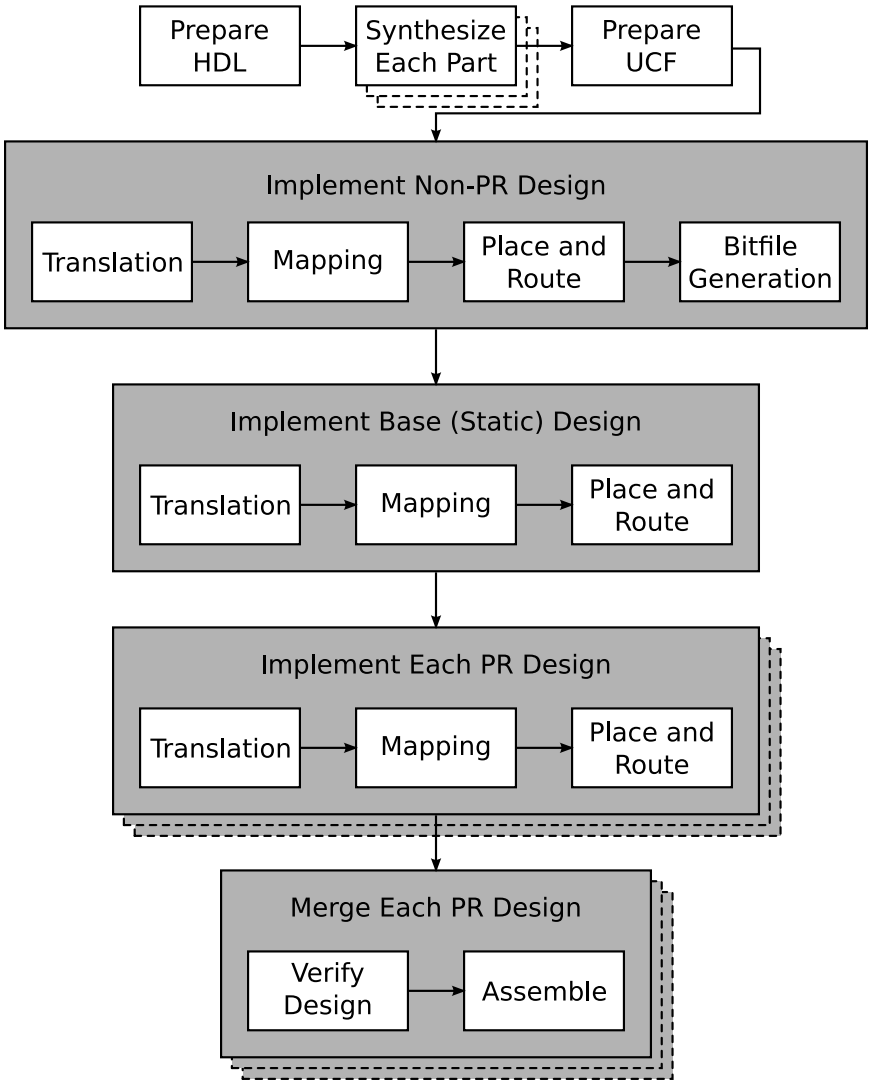

Figure 2.6: Xilinx Partial Reconfiguration Flow

mapping, placing and routing tools. The Xilinx tools used for each step must first be patched to support PR, and only specific versions of the tools can be patched. Each partial module included in the design is similarly processed.

The final step in the Xilinx PR flow is to merge the individual pieces together, including the top level, base static build, and each partial module. The merge process is not part of the standard Xilinx build flow. The final output of the merge step is a series of bitstreams. For each partial module, there is a full system bitstream, a partial module bitstream, and a blanking bitstream (used for clearing the partial region) produced. This flow is useful for producing partial modules, however the process requires many manual steps and specific knowledge of the FPGA's architecture. A portion of the work presented in this thesis simplifies the Xilinx PR flow.

2.5 Applications

The nature of the proposed FPGA run-time system maps well to problems requiring System on a Chip (SoC). An SoC is composed of several processing cores on a single chip that communicate with each other through the use of buses or on-chip networks. By using the run-time system proposed in this thesis, the buses and on-chip networks are replaced by more efficient point-to-point connections. Also, less actual chip-area and less power consumption is required since cores can be swapped in as needed by the run-time system instead of all possible cores existing simultaneously. SoC performance and power efficiency should benefit from the proposed run-time system by having a wider variety of specialized cores available for use. There is essentially no limit to the number of cores available in the module library. The possible applications of SoC have a broad range, from gaming consoles to cell phones. SDR can also be considered an application of SoC [5].

A radio designed with software components that can be reconfigured to alter the behavior

of the radio is termed a SDR [22] [23]. The definition of SDR implies that the changes made to alter the operation of the radio are done using software. This behavior allows for more flexibility in the signals an SDR can transmit and receive, however conventional software is not suited to high speed streaming data. Reed defines *soft radio* as configurable radio with hardware that changes function as well as software [22]. In this thesis, SDR is used to encompass soft radios as well. In the proposed run-time system, the programmable nature of the SDR is achieved by instancing separate pre-compiled cores, or a combination of pre-compiled cores, to achieve the different waveforms desired. This moves some of the software computation to hardware, which will improve performance.

2.6 Related Work

Several approaches have been proposed to prepare IP modules for run-time reconfigurable systems. As described in Section 2.4.2, the Xilinx PR flow is one such approach and is also the basis for several other approaches. The compile-time tool flow is complicated and lacks automation. In addition to having a complicated tool flow, the modules produced do not allow for flexible run-time placement or routing. Some alternative efforts discussed in this section are summarized in Table 2.2 based on the automation of the compile-time tool flow and the flexibility of the resulting IP modules.

Run-time communication synthesis could be very beneficial to a RTR system. Tools exist that automate the system-level communication and interconnect for SoC, Network on a Chip (NoC), and other similar static designs. Sonics SMART Interconnects and IBM's CORAL are examples of tools that perform communication synthesis for static designs [24][25]. In the Sonics and CORAL flows, the designer provides the cores and a description of the communication requirements to the tools and the physical constraints are automatically generated. However, no RTR design flows reviewed implement run-time communication synthesis.

PADReH is a framework to assist in the design and implementation of dynamic reconfigurable systems [26]. The run-time system uses pre-compiled modules in the form of partial bitstreams. The partial bitstreams are produced using the Xilinx PR flow, however MDLauncher, or Modular Design Launcher, is a tool developed to automate portions of this process. The user creates a top level module and each partial module as input to MDLauncher. The tedious tasks of designing inter-module communication, inserting bus macros, and creating the user constraints file fall on the user. MDLauncher automates the tool flow by generating several scripts that perform all steps of the Xilinx PR tool flow. The resulting partial bitstreams are identical to those produced by manually using the Xilinx PR flow, which means they have the same shortcomings. Modules produced using MDLauncher are intended for pre-defined slots of a particular system with static communication to and from each slot. Any modules intended for the same slot must have identical port interfaces and dimensions.

The approach taken by the COMMA project proposes both compile-time automation and run-time flexibility [27]. The Xilinx PR flow is used to produce partial bitstreams for modules used in the run-time system, however there is no automation of the flow. The IP modules have to be specially prepared before implementation with the Xilinx PR flow in order to support the desired run-time flexibility. IP modules are wrapped with Reconfigurable Data Ports (RDPs) which connect the logical ports of the module to the physical ports of the bus macros. The RDPs are customized for each module based on knowledge of the run-time system. Bus macros connect the module to an infrastructure that allows flexible communication at run-time. The infrastructure, generated at compile-time, is composed of multiplexers that allow flexible module to module and module to I/O connections, and is designed specifically for each application with some knowledge of the run-time behavior and module placement schedule. The multiplexers do not allow custom point-to-point connections between any two ports of the system, but only between ports that will likely need to be connected at some point during operation as determined by the compile-time tools. This project is in the early stages of development and little has been implemented.

PARBIT (PARtial BItfile Transformer) is a low-level tool developed to allow relocation and dynamic loading of IP modules through bitstream manipulation [28]. The process of generating the original bitstream is not addressed, however PARBIT does not set limitations on which flow to use. Using PARBIT, a module within a bitfile is transformed to a Dynamically loadable Hardware Plug-in (DHP) module in the form of a partial bitstream. The DHP module can be loaded to a specified region of an FPGA device. The specified region need not be the same region the DHP module existed in the original bitfile since PARBIT supports relocation. The only requirement is that the target location is empty, which implies some notion of a slot with no static logic. PARBIT leaves it to the designer to ensure that the DHP module communication matches the communication of the static logic at the target location.

Bitstream Intellectual Property (BIP) core generation is an extension of PARBIT that addresses the issue of communication between the static design and the partially reconfigurable module [18]. Bus macros are used to define the communication interface. The design flow is very similar to the standard Xilinx PR flow with only one noticeable advantage: BIP core relocation. Run-time systems using BIP cores must be designed with one or more empty slot(s). The empty slot(s) and the BIP core(s) must have the same dimensions and must have bus macros placed in the same positions relative to the BIP core(s). An advantage over the standard Xilinx PR flow is that the partial bitstream generated from PARBIT can be modified with new location information using PARBIT.

Xilinx's unreleased merge partial reconfiguration is an approach similar to PARBIT with the BIP extension [29]. The run-time system must be designed with slots where IP modules will be loaded. The slots use what the designers call slice macros, which are similar to bus macros (if not the same). An advantage merge partial reconfiguration has over PARBIT is that static routes are allowed to intersect or pass through the slots intended for IP modules by reserving routing for the static design. The designer synthesizes and implements the static and module designs using standard Xilinx tools. A custom tool is then used to re-route any signals that

violate the constraints enforced by reserving routing. The static design will be modified to use only the reserved routing through a module slot and the module design will be modified to only use routing that is not reserved for the static design. Standard Xilinx tools are then used to generate the static bitstream for the static design and the partial bitstream for the module. An additional step is required to remove redundancy between the static bitstream and the partial bitstream for the run-time tools to work properly. At runtime the FPGA configuration is read back and modified with the partial bitstream. The static routes are preserved since the partial bitstream is merged with the existing configuration of the FPGA.

The run-time system proposed by M. Ullmann, et al., for automotive control units addresses IP module preparation [30]. However, the process of building IP modules is not addressed. The run-time system proposed uses a soft-processor with several partially reconfigurable slots attached for hardware acceleration. Communication between the slots and the processor is implemented using an on-chip bus. Bus macros are used to connect the slots to the bus. The modules use a predefined interface using bus macros to communicate with the bus, which eliminates the need for flexible routing. However, bus communication introduces overhead that is not present in custom point-to-point connections.

The majority of the approaches reviewed fail to address the difficult design and tool flows associated with dynamic reconfiguration. PADReH introduces automation to the standard Xilinx PR flow and COMMA automates a portion of the design flow. All approaches reviewed target slot-based partial reconfiguration and all but COMMA target static routing from the slots to the rest of the system. Slot-based partial reconfiguration limits the flexibility of the run-time system by forcing modules designed for the same slot to have the same dimensions, port interface, and resource requirements. Static inter-module routing also places limitations on the run-time system by predefining which parts of the system can communicate with the slot. The multiplexed routing of COMMA is an improvement over static routing but still falls short of custom point-to-point connections. The ability to relocate IP modules in the run-time system adds flexibility, however, relocation in slot-based run-time systems limits

	Type(s) of Automation	**Slot Based?**	**Type of Relocation**	**Intermodule Routing**	**Interface Options**
Xilinx PR	NA	yes	NA	static	user defined
PADReH	tool flow	yes	NA	static	user defined
COMMA	design flow	yes	between similar slots	multiplexed	user defined
PARBIT/ BIP	none	yes	between similar slots	static	user defined
Merge PR	none	yes	between similar slots	static	user defined
Automotive	none	yes	NA	static	bus
WoD	tool flow design flow	no	arbitrary	dynamic	user defined

Table 2.2: Summary of related research

IP modules to slots of similar characteristics.

Chapter 3

Tool Flow for Flexible Run-time Reconfigurable Computing

3.1 Overview

The Wires on Demand project in the Configurable Computing Lab at Virginia Tech proposes a run-time reconfigurable computing flow enabling dynamic module placement and communication synthesis. A compile-time tool flow that prepares IP modules for dynamic placement and routing is necessary for the flexibility desired at run-time. However, current industry tools have not been able to provide a tool flow capable of run-time placement and communication synthesis. The compile-time tool flow presented in this thesis was designed and implemented to meet the needs of the WoD run-time flow. This chapter presents the WoD run-time and compile-time tool flows.

3.2 Design Objectives

As mentioned in Section 1.1, current means of RTR in FPGAs lack some desirable capabilities that should be achievable given the architecture of FPGAs. The goal of the WoD project was to create a tool flow that supports capabilities mainstream tool flows ignore. The WoD tool flow had a number of objectives to meet:

1. Develop a flexible RTR flow for data flow applications.
2. Insulate the designer from the low-level details of partial reconfiguration.
3. Automate the compile-time flow.
4. Achieve flexible run-time placement and routing.
5. Achieve efficient run-time reconfiguration.
6. Target the Xilinx V4 architecture.

3.3 Proposed Flow

The WoD flow includes compile-time tools that generate a dynamic module library and run-time tools that implement a working system using the dynamic module library. An example system layout on an FPGA is shown in Figure 3.1. In this system layout, the target FPGA is a Xilinx XC4VLX60. The left half of the FPGA contains a sandbox area with several dynamically placed modules from the library and routing channels used for run-time communication synthesis. The right half of the chip contains another sandbox region that is empty. The WoD flow used to implement such a system is shown in Figure 3.2. The design flow begins with dynamic module library generation.

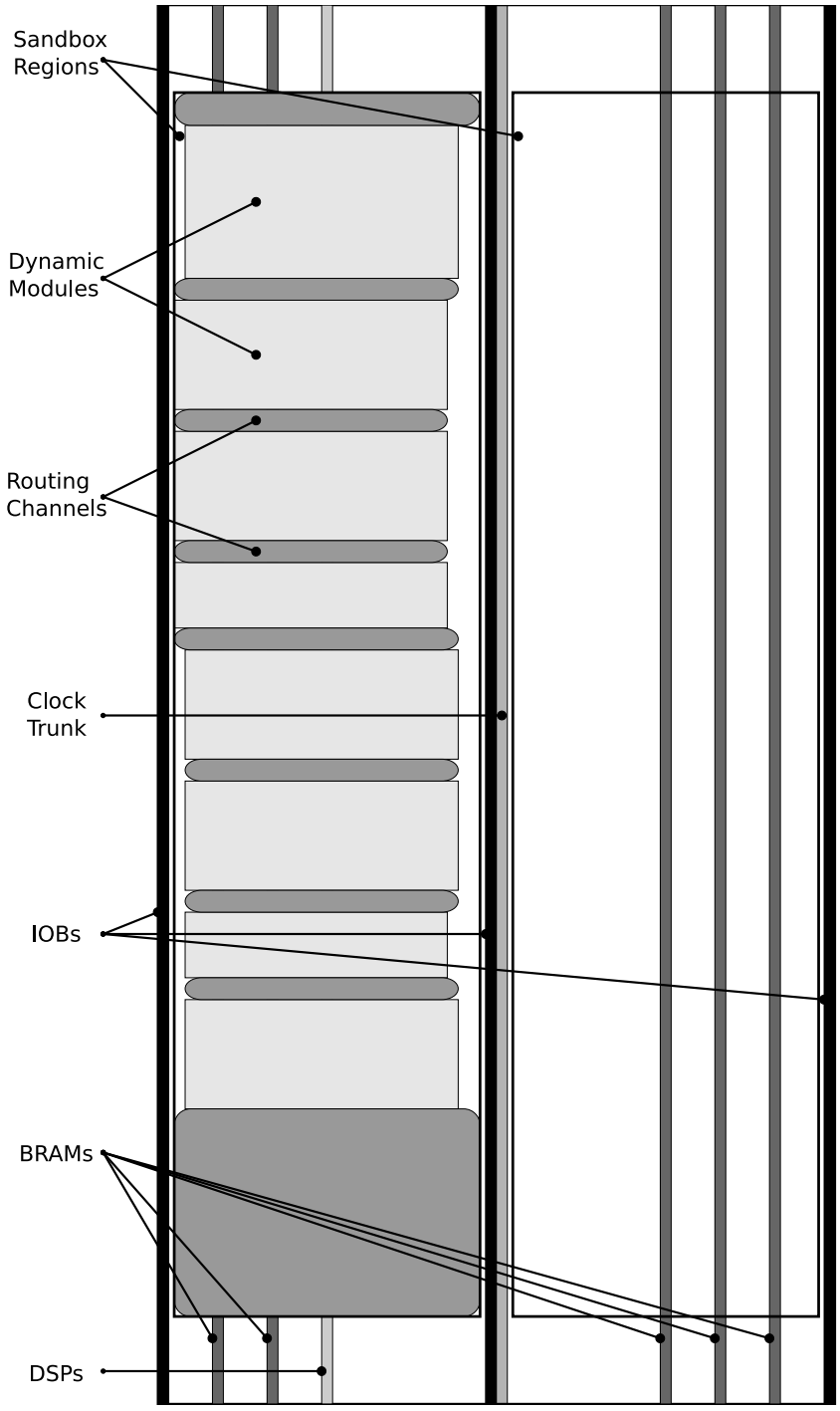

Figure 3.1: Example System Layout

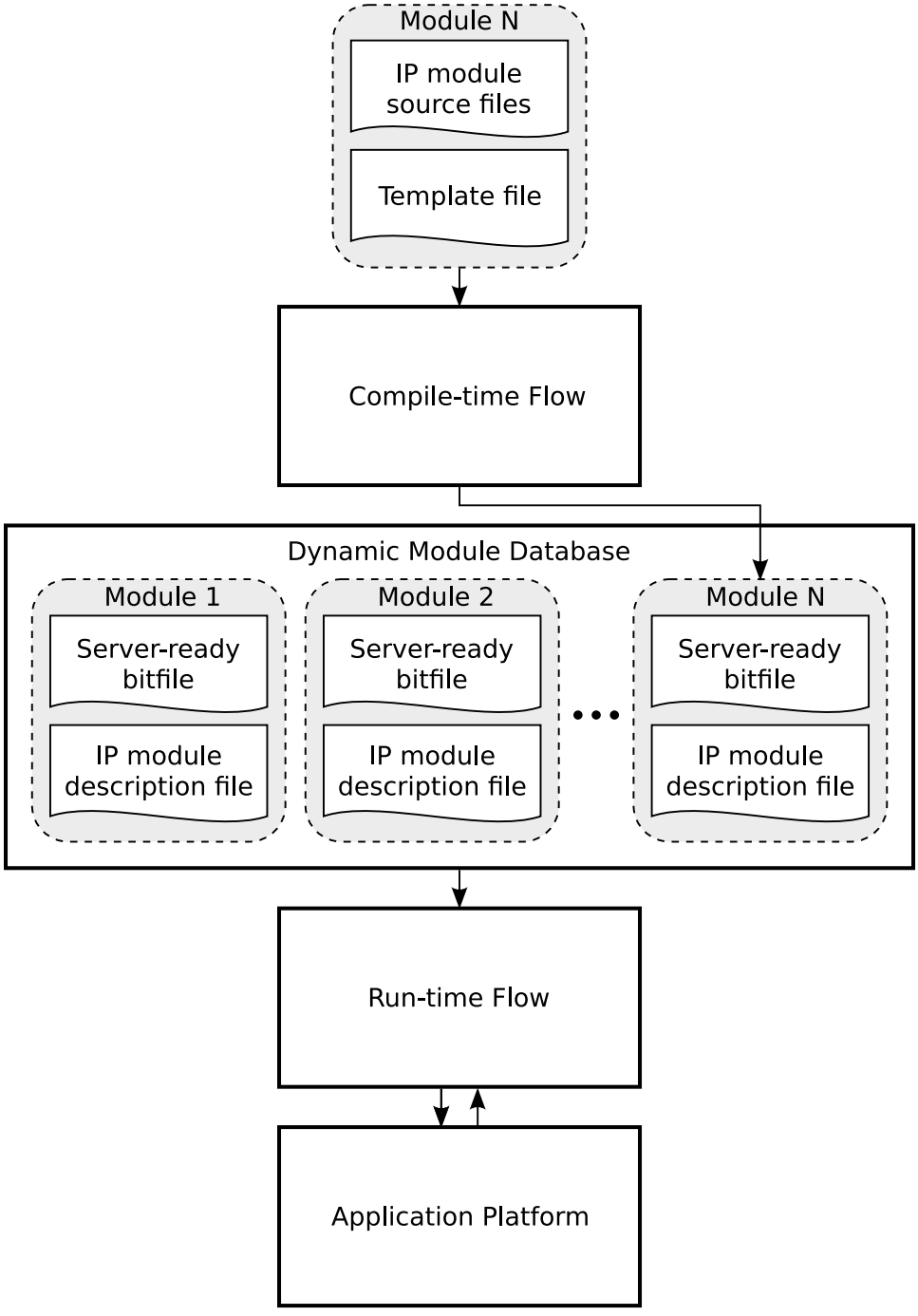

Figure 3.2: Wires on Demand Flow

3.3.1 Compile-time Flow

Dynamic modules are generated individually using the compile-time tool flow, described in detail in Section 4.1. The compile-time tool flow, illustrated in Figure 4.1, begins with HDL design or existing IP modules available in either HDL source or synthesized netlist format. The designer must create a template file that describes the IP module before executing the automated compile-time tool flow. The compile-time tool flow generates a partial bitstream and an XML description file for the module. The tools utilized by the compile-time tool flow include a preprocessor, the Xilinx PR tools, and a postprocessor. The only responsibility the designer has is to provide the IP module source and prepare the template file; the tool flow takes care of the rest. The partial bitstream and the description file are stored in the dynamic module library for use by the run-time system.

3.3.2 Run-time Flow

The run-time flow performs dynamic module placement and inter-module communication synthesis through bitstream manipulation. Two user interface options have been developed to provide access to the tools: a command line interface and a web page interface. Both user interface options provide the tools to build and modify a system at the user's request. IP modules in the dynamic module library serve as building blocks for construction of a run-time application. The user has the ability to add and remove modules from the current data path in order to change the behavior of the system. When a module is chosen for addition to the data path, the placement tool is used to insert the module in the specified location. If the module chosen will not fit in the specified location of the data path, space is made by moving the existing modules and re-routing the inter-module communication signals. The router tool is used to generate connections between modules of the data path. XML description files produced by the compile-time tools provides the necessary information for placement and routing of modules. The bitstream manipulation tools are used by the

placement and routing tools to produce a partial bitstream that will make the requested changes to the current FPGA configuration.

3.4 Compile-time Tool Flow Example

An absolute value module useful for applications such as the test application described in Section 5.2 will be used for this example. The designer must first prepare a template file that provides details about the target device and the absolute value module, as shown in Figure 3.3. The template file format is described in more detail in Section 4.2. Appendix A.1 shows the Verilog module definition of an absolute value module named `AbsVal` and Appendix A.2 shows a template file for the absolute value module. The file that contains the source HDL is named `AbsVal.v`. Source files for the module and the template file are located in the `source` folder following the folder structure outlined in Section 4.1. The template file targets an XC4VLX60 FPGA and provides some additional physical preferences about the absolute value module as well as describing port details.

The master makefile for the compile-time tool flow must be modified for use with the absolute value module. Usually updating the makefile with the current module name is the only change that has to be made, however it is important to make sure the rest of the parameters are correct as well for the build to succeed. The makefile for the absolute value module in this example is shown in Appendix A.3. The makefile and compile-time tool flow are explained in more detail in Section 4.1. To execute the makefile, simply type `make` from a command prompt while located in the build directory for the module. The makefile will create the necessary folder structure for the build flow and then execute the preprocessor.

The preprocessor parses the template file and generates several HDL files, a UCF file, an XML description file, and several scripts. The HDL and UCF files provide a wrapper structure for the absolute value module necessary to achieve the desired run-time flexibility.

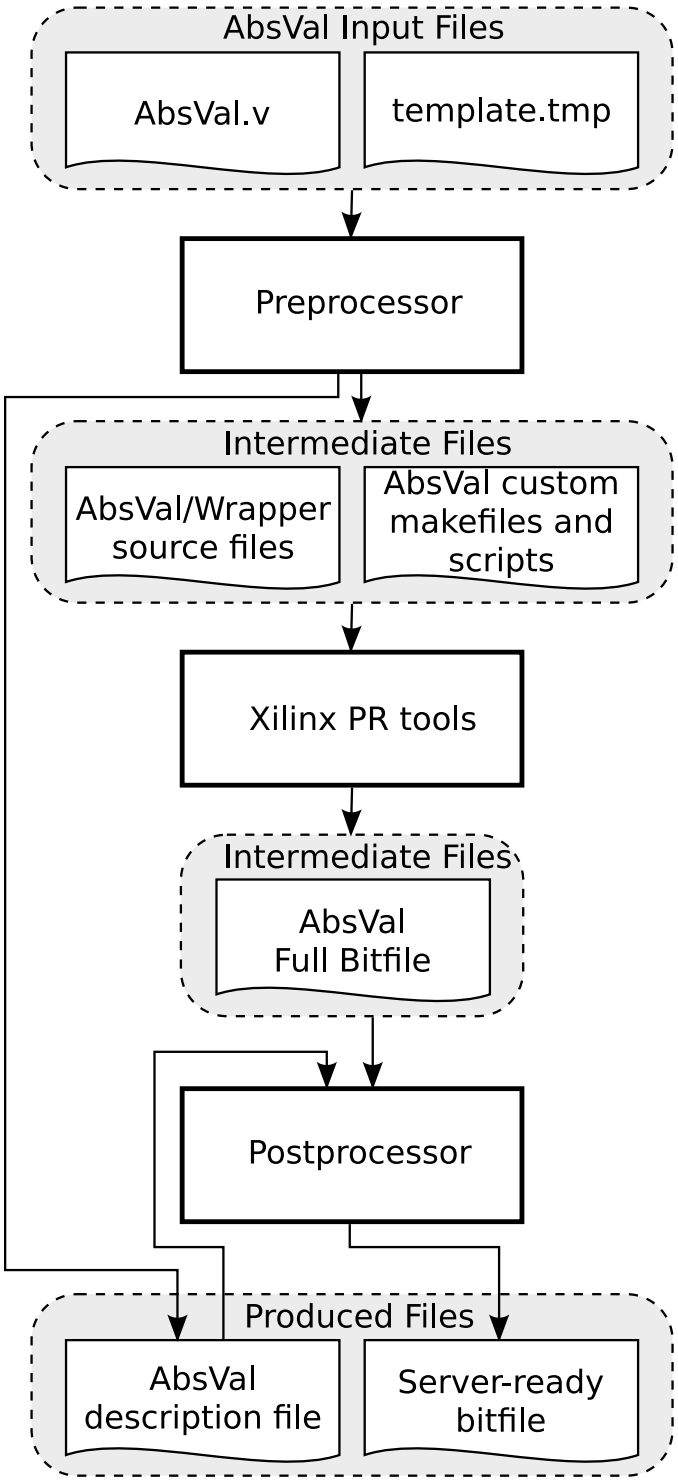

Figure 3.3: AbsVal Example Compile-time Flow

Appendix A.4, A.5, and A.6 show the HDL files generated by the preprocessor. The UCF file is shown in Appendix A.7. The script files generated by the preprocessor are used to automate the the Xilinx PR tools in the next step of the compile-time tool flow. The XML file (see Appendix A.8) is used by the postprocessor later in the compile-time tool flow and by the run-time tools.

Once the preprocessor completes execution, a Perl script generated by the preprocessor is executed. The Perl script, shown in Appendix A.9, generates a list of the source files used during the synthesis step of the Xilinx PR flow. Three of the scripts generated by the preprocessor are makefiles that automate the Xilinx PR flow. Bus macro source files are copied to the build directory of the module by the first of the three generated makefiles, shown in Appendix A.11. The next makefile, shown in Appendix A.12, is responsible for the bulk of the Xilinx PR flow, excluding the merge step. Automation of the merge step is achieved by the last of the generated makefiles, shown in Appendix A.14. The preprocessor and the files it produces are described in more detail in Section 4.3.

Three bitfiles are produced by the Xilinx PR flow. Two of the bitfiles produced, a blanking bitfile and a partial bitfile, are not used by the WoD flow. The bitfile used is the full bitfile. Post-processing the full bitfile is the final step of the compile-time tool flow. A partial bitfile is produced by the postprocessor using the XML description file produced by the preprocessor and the full bitfile produced by the Xilinx PR flow. The partial bitfile and the XML description file are stored in the module database for use by the run-time tools.

Chapter 4

Compile-Time Tool Flow

The compile-time tool flow, shown in Figure 4.1, begins with the designer providing IP module source files and preparing a template file describing the IP module. A makefile is used to automate the tool flow. The makefile creates the necessary folder structure, executes the preprocessor, the Xilinx PR tools, and the postprocessor. Two files are produced as a result: a partial bitstream and an XML description file. The two files are stored in the dynamic module library for use by the run-time tools. Automating the build process both insulates the designer from low-level partial reconfiguration details and automates part of the compile-time flow.

4.1 Automated Tool Flow

A makefile automates the compile-time tool flow. For the makefile to work properly, the folder structure shown in Figure 4.2 must be used. The HDL source for the module must be placed in the `source` folder along with the template file. The module name and the source file name (excluding extension) must be identical. If the source for the module is a netlist the designer must create an HDL wrapper for the module that defines the module name and

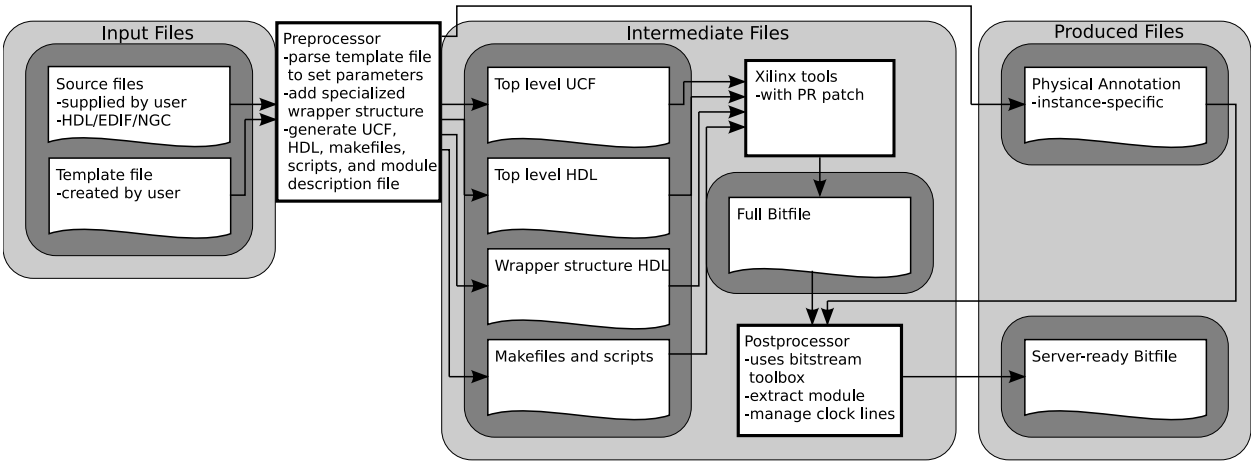

Figure 4.1: Compile-time Tool Flow

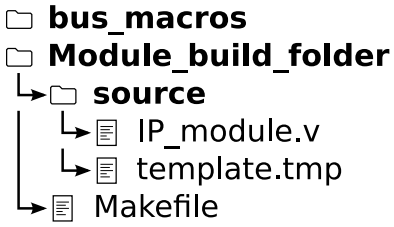

Figure 4.2: Compile-time tool flow folder structure

port list, such as the Verilog file shown in Appendix A.1. Additional source files may be placed in a folder of the designer's choosing by adding the folder path to the makefile, as described next.

The makefile has several parameters located near the beginning of the file, as shown in Appendix A.3. All parameters are case sensitive. The `MOD_NAME` parameter must be assigned the module name that appears in the HDL source and the `TEMPLATE` parameter must be assigned the file name of the template file. Since both files must exist in the `source` folder the path should not be included. Typically, the other parameters should only need to be changed the first time the makefile is used on a system. `PreP` and `PostP` are the names of the preprocessor and postprocessor executables, respectively. The `BIN_PATH` parameter is the full path to the folder that contains the preprocessor and postprocessor executables. Lastly, `ADDL_SRC` can be used to specify the locations of additional source files necessary for the

module to build properly. Multiple locations can be specified separated using spaces. This feature is useful for modules that use a common module for the communication interface. The source for the interface module does not need to be copied into the source folder of each project directory. Instead, the designer can add the path of the folder where the interface module source code is located to the `ADDL_SRC` parameter. The remainder of the makefile should not be modified.

Once the source HDL files are in the correct locations, the template file is prepared, and the makefile is configured correctly, the automated tool flow is ready to start. The tool flow is started by typing `make` from a command prompt while located in the build directory for the module. The makefile will first create the folder structure necessary for the tool flow, shown in Figure 4.3, and then execute the preprocessor. Several files are generated by the preprocessor that are necessary for the Xilinx PR flow and the run-time tool flow. Excluding the bitfile located in the `final` folder and the source files provided by the designer, all files shown in Figure 4.3 are produced by the preprocessor. The preprocessor and the files it generates are described in detail in Section 4.3.

A Perl script generated by the preprocessor is used to generate a list of the module source files necessary for the synthesis step of the Xilinx PR flow. Three makefiles are also generated by the preprocessor. The first makefile moves the necessary bus macro source files to the build directory of the module. The second makefile automates the bulk of the Xilinx PR flow, excluding the merge step. The last of the three makefiles automates the merge step. Three bitfiles are produced by the Xilinx PR flow, however only one of the three files is used. The blanking bitfile and the partial bitfile are not used; only the full bitfile is used.

Post-processing the full bitfile is the next step of the compile-time flow. The postprocessor uses the XML description file produced by the preprocessor and the full bitfile produced by the Xilinx PR flow to produce a partial bitfile. The partial bitfile produced by the postprocessor is different from the partial bitfile produced by the Xilinx PR flow. The differences are discussed in Section 4.4. The partial bitfile and XML description file generated

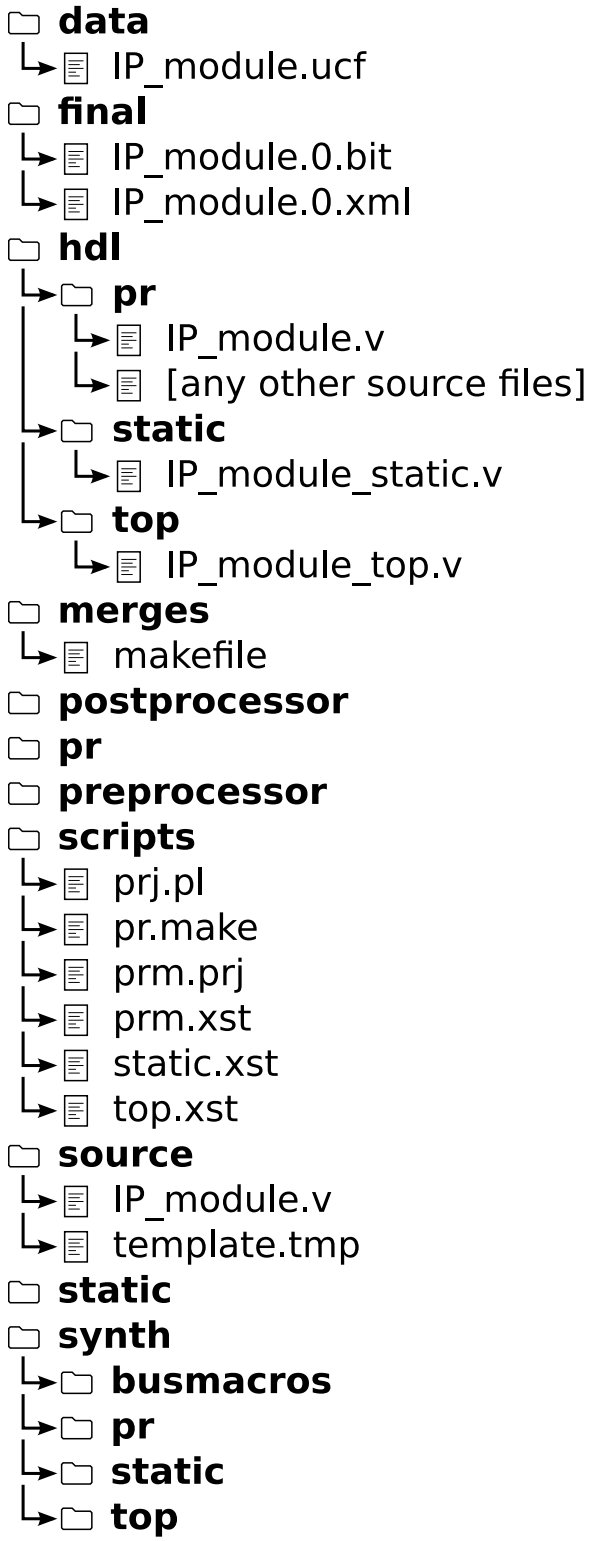

Figure 4.3: Makefile generated folder structure

are located in the `final` folder upon completion of the compile-time tool flow. These two files will be stored in the dynamic module library for use by the run-time tool flow.

4.2 Template File

The template file is prepared by the designer to provide the preprocessor with specific details for a particular core. The information provided in the template file is useful to compile-time tools as well as run-time tools. The preprocessor uses information gathered from the template file to generate the UCF, set the module dimensions and placement, anchor port locations, set port directions, set the correct data flow, generate the correct wrapper structure, and generate accurate description files for the module database. Without a correct template file, either the build-flow will fail or the resulting module database bitstreams will be incorrect. The designer is insulated from partial reconfiguration and low-level FPGA details by supplying all necessary data in the template file format.

The template file needs to be human readable and editable while keeping a format that can be parsed by a computer program with ease. The resulting template file format is structured into four sections (`global`, `module`, `restrictions`, and `ports`) that provide global details, module details, placement restrictions, and port details, respectively. The format of the `global`, `module`, and `restrictions` sections are all similar; within each section there are several parameters that have values assigned to them. Each parameter must appear on a separate line within the section. The ports section is slightly different because it has subsections and each subsection is similar to the sections previously described. The `ports` section has a `port` subsection for each port signal. Similar to the previous three sections described, the `port` subsection has parameter-value pairs. The template file was designed to easily describe dataflow applications. The structure of the template file is best described using an example, provided in Appendix A.2.

Each section and subsection of the template file is started by a section or subsection keyword followed by an open curly brace ("{"). Both sections and subsections are referred to as sections from this point forward. To mark the end of a section, a closing curly brace ("}") is used. The order of the sections must be `global`, `module`, `restrictions`, `ports`. A section must be closed before starting a new section.

Within a specific section, each parameter associated with that section must appear on a separate line. The value assigned to a parameter must appear on the same line after the parameter keyword, separated by white space. To insert a comment in the template file, the hash character ('`#`') is used. Any text located to the right of a comment character on the same line is ignored by the preprocessor. Each section name and parameter have a keyword associated to them that cannot be used elsewhere in the template file. Table 4.1 lists all of the keywords, which are explained in more depth below.

4.2.1 `global` Section

The `global` section contains data relevant to all cores that are part of the same system. The `global` section provides details about the target FPGA part and architecture (including package and speed grade), clock buffer location, desired I/O standard, and timing information. The timing information can be provided using period, frequency, or the exact timespec string to be included in the UCF.

`part` specifies the full part type, including package and speed grade.
Example: `part xc4vlx60-ff668-10`

`bufg` specifies the exact value needed in a UCF file `LOC` constraint to choose a particular clock buffer. Any clock buffer location will work equally well as long as it is a valid clock buffer site for the part chosen.
Example: `bufg BUFGCTRL_X0Y0`

Sections	Subsections	Parameters
global		part bufg iostandard period frequency timespec htrunkline
module		modulename filename dimensions dataflow version passthroughs
restrictions		location
ports	port	name type bits dir wrap dataflow stream bus_type

Table 4.1: Template file keywords

`iostandard` specifies the I/O standard desired. Generally, this parameter will not have any effect on the module library core produced since the partial modules do not directly interface with the I/O ports.

Example: `iostandard LVCMOS25`

`timespec` specifies the exact string to be entered in a UCF file to determine clock behavior. Since the timespec is a string, the string delimiters [and] are used to mark the beginning and end of the string, respectively.

Example: `timespec [TIMESPEC "TS_clk" = PERIOD "clk" 10.00 ns HIGH 50%;]`

`period` specifies the period of the clock signal in nanoseconds. The period value can be specified in integer or floating point form.

Example: `period 10.0`

`frequency` specifies the frequency in MHz. The frequency value must be specified in integer form.

Example: `period 100`

`htrunkline` specifies the horizontal clock trunk used in the sandbox of the static design. This is now redundant since the run-time tools have been updated to modify the bitstream to use the correct horizontal clock trunk. Also as a result, if the wrong clock trunk is specified or if the module is to be used on several sandboxes with different horizontal clock trunks, the run-time system will not be affected.

Example: `htrunkline 0`

As mentioned previously, only one of the timespec, period, and frequency parameters may be specified because providing more than one is redundant.

4.2.2 `module` Section

The `module` section contains data relevant to the specific core being used for a particular RTR system. Details specified here include the HDL module name used, the source HDL filename, the desired dimensions (X and Y) for the module specified in number of CLBs, the desired dataflow (N, S, E, W), the version number, and whether passthrough signals should be added. The preprocessor supports using netlists instead of HDL when necessary, however there must be an HDL black box that defines the module name and port list. This is described in more detail in Section 4.1. The module name and source HDL filename refer to the black box HDL file when using netlist source files. Otherwise, the HDL source for the core is used. The dimensions given must provide sufficient resources to implement the core. Dataflow describes the side of the module that data signals exit the module. The version identifier is used to distinguish different builds of the same core. The passthrough option specifies whether the module should include passthrough signals in addition to the necessary port signals. A passthrough signal, as described in Section 4.3.1, has an input port on one edge of a module and an output port on the opposite edge of the module to allow the run-time system to pass signals through the module.

`modulename` specifies the name used for the module in the HDL source or HDL black box file. The name must be exactly as it appears in the HDL (case sensitive).
Example: `modulename sample_module`

`filename` specifies the filename of the HDL source or HDL black box file for the module being processed. This is also case sensitive.
Example: `filename sample_module.v`

`dimensions` specifies the desired dimensions of the core prior to adding any wrapper structure. Once the wrapper structure is added by the preprocessor, the dimensions will increase along the edges where the ports are placed because additional area is needed

for port anchors. Dimensions are provided in number of CLBs.

Example: `dimensions X16Y6`

`dataflow` specifies the flow of data through the core. The only four valid values are `north`, `south`, `east`, or `west`. The appropriate value is chosen by determining which edge of the core the data will exit (output).

Example: `dataflow south`

`version` specifies the difference between different builds of the same core. Different versions must perform the same function and have the same port interface. Versions may differ in resource requirements, dimensions, location, dataflow, etc. Value must be specified in integer format.

Example: `version 0`

`passthroughs` specifies whether passthrough signals will be added to the module. The addition of passthrough signals may cause the need to increase the module dimensions if the build fails. This is because the passthrough signals use much of the routing resources in one direction (either vertical or horizontal depending on dataflow). The only two valid values are `include` and `exclude`.

Example: `passthroughs include`

4.2.3 `restrictions` Section

The `restrictions` section contains data used to determine the module placement for the build process. Module placement is specified using the CLB coordinate system. Initially this section was designed to contain additional information but it was decided that the additional information was unnecessary or will be implemented in a later release of the preprocessor.

`location` specifies the exact location of the lower-left corner of the core prior to having the wrapper structure added by the preprocessor. By using the position as a pre-wrapper

value, any special resources are kept in the same position relative to the core logic. The partial region associated with the module will extend a single CLB on each side of the module that has ports. For example, a module with data that flows south will have ports on the top and bottom edges of the module. The partial region extends one CLB above and one CLB below the module region specified by the module dimensions and module location. This is important when building a module that uses special resources such as BRAM or DSPs since the partial region must include the entire special resource. For the example given above, choosing a module position a single CLB above the lower edge or below the upper edge of a special resource will include the entire special resource in the partial region. Position coordinates must be specified in CLB coordinates.
Example: `position X66Y40`

4.2.4 `ports` Section

The `ports` section is used to describe the port list and how the core will communicate with other elements of the system. As mentioned at the beginning of Section 4.2, the `ports` section contains a `port` subsection for each port signal of the core. Within each `port` subsection, details are provided to describe the individual port.

`port` specifies information describing a port the module uses to communicate with the rest of the system.
Example: `port {...}`

`port` Subsection

The `port` subsection is different from other sections because the information provided changes depending on the port since not all ports require the same information. Clock ports do not

require data flow, bus type, or stream id information. Ports that are members of a bus require bus type and stream id information, but all other ports do not.

Information that must be provided in each `port` subsection includes the port name as it appears in HDL, port type, port width (in bits), port direction, and whether the signal is included in the wrapper structure. The HDL port name comes from the HDL source or the HDL black box for the module. The port type is used to properly identify the port for routing connections in the run-time system. The port name and port type are both case sensitive. Ports of neighboring modules with the same port type will be connected together. Port types can be named arbitrarily as long as they match up correctly with other cores in the system, however special cases are clock signals, reset signals, and reconfiguration control signals. A clock signal must be of type `clock` and a reset signal must be of type `reset` for the module to behave correctly in the run-time environment. Special types for reconfiguration control signals are `RC_IGNORE_US`, `RC_IGNORE_DS`, and `RC_RESET`. Additional information about the reconfiguration control special cases is provided in Section 4.3. The port width is the number of bits of the signal. Port direction is either `input` or `output`. The only valid values for designating whether the port is included in the wrapper structure are `include` and `exclude`. In most cases, the only port that will be excluded is the clock signal.

Additional information required by some ports include the data flow, bus type, and stream id. Port data flow is given in relation to the module data flow. The only valid values for port data flow are `with` or `against`. Most ports will be `with` data flow, however any feedback signals that go against the system data flow will be designated as `against` data flow. The bus type and stream id parameters are used to group signals together as a bus. The bus type is user defined to represent the type of bus to which the port belongs. The stream id is necessary because some modules may have multiple buses of the same type. Ports on the same side of the module with the same stream id and bus type will be part of the same bus. Two buses can have the same stream id and bus type as long as they exist on different sides of the module.

The reset port for a module must have a type of `reset` and the clock port for a module must have a type of `clock`. The clock port must also be excluded from the wrapper structure. Ports that are excluded from the wrapper structure do not require the `dataflow` parameter, however all other ports must specify the data flow. The side of the module that the port resides is determined by the port direction, port data flow relative to the module data flow, and the module data flow. Ports with the same direction and same data flow or ports with different direction and different data flow will be located on the same side. For example, all input ports with the data flow and all output ports against the data flow will be on the same side. All input ports against the data flow and all output ports with the data flow will be on the opposite side.

Parameters that appear inside a port statement are listed below. The port described in the examples is a 16-bit input port that is carrying raw data, not control data. The ports name is "data_i". The port may be defined in Verilog HDL as follows: `input [15:0] data_i;`

`name` specifies the name used to define the port in the source HDL or black box HDL file for the module.

Example: `name data_i`

`type` specifies the type of port being defined. Port types are used to make run-time connections. Special case types are `clock`, `reset`, and reconfiguration control signals as described previously in this section. The reconfiguration control signals are not actual ports that are driven by a signal from a driver elsewhere on the FPGA. Instead, the signals are driven by LUTs located in the bus macros. The signal remains constant until the run-time tools modify the LUT contents by loading a partial bitstream. Due to the nature of the reconfiguration control ports, they must be processed differently than other ports, which is why the types are special cases.

Example: `type data`

Note: The corresponding output port (i.e. "data_o") of a neighboring module must have the same type for the two ports to be connected at run-time.

`bits` specifies the width or number of bits of the port signal. The port width must be specified in integer form.

Example: `bits 16`

`dir` specifies whether the port is input or output to the core. The only two valid values are `input` and `output`.

Example: `dir input`

Note: The corresponding port of a neighboring module would have the opposite direction, `output`.

`wrap` specifies whether the port is to be included in the wrapper structure or not. Generally, the only port that should be excluded from the wrapper structure is the clock port. The only two valid values are `include` and `exclude`.

Example: `wrap include`

`dataflow` specifies whether the port signal is communicating with or against the module dataflow (as defined in the `module` section of the template file). The only two valid values are `with` and `against`.

Example: `dataflow with`

`stream` specifies the data stream or control stream the port is associated with and is only necessary when the port is part of a bus. Valid values must be in integer form.

Example: `stream 1`

`bus_type` specifies the type of bus the port is part of and is only necessary when the port is part of a bus. The bus type is used to make run-time connections. Values are user defined.

Example: `bus_type latol`

4.3 Preprocessor

Development of the preprocessor was motivated by the need to generate a wrapper structure for the IP module source before using the Xilinx PR tools to generate a bitstream. The purpose of the preprocessor has grown to include the generation of XML description files. User constraint files and scripts that automate the compile-time tool flow generated by the preprocessor insulate the designer from the low-level FPGA and partial reconfiguration details.

4.3.1 Generated HDL and the Wrapper Structure

The preprocessor generates several source files, in addition to the IP module source, that are used by the Xilinx PR flow. Generated source files include a wrapper structure, static logic, a top level module, and a user constraints file. A wrapper structure is necessary to provide the run-time flexibility desired and the static logic is necessary to prevent the Xilinx PR tools from removing important parts of the design. Instantiations of the wrapper structure, the static logic, and the bus macros are made in the top-level module. The user constraints file provides the necessary directives for synthesis and implementation using the Xilinx PR flow.

Wrapper Structure HDL

The wrapper structure was originally designed to provide a reconfigurable port interface to an IP module. Multiplexers were placed along the edge of the module that could route port connections to several physical locations and provide passthrough signals. The flexibility provided by the multiplexers, illustrated in Figure 4.4, proved to be unnecessary as the WoD project progressed. A need for passthrough signals remained, but the multiplexers were removed from the wrapper structure. The current wrapper structure, shown in Figure 4.5,

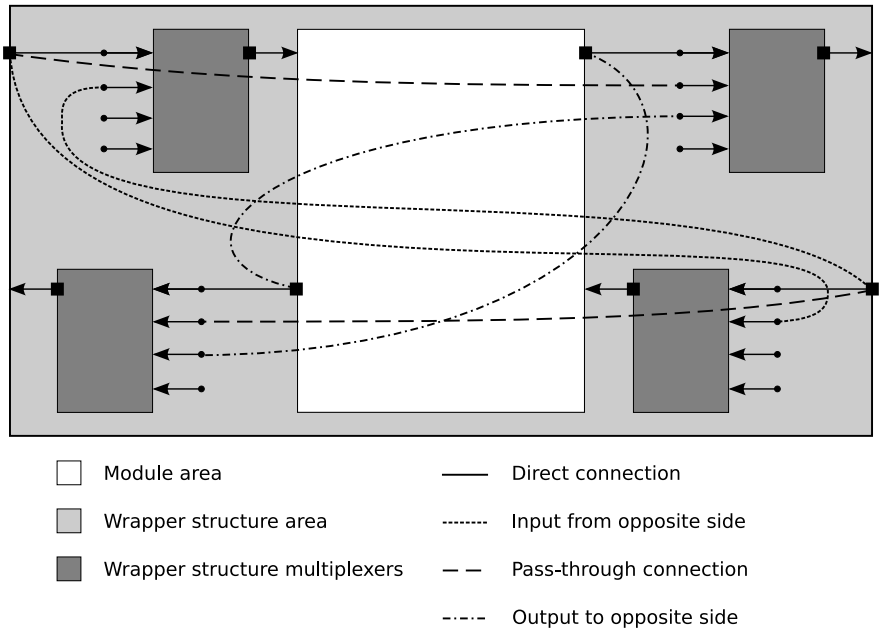

Figure 4.4: Wrapper structure with multiplexers

has a smaller footprint and reduces latency by eliminating the additional logic needed to implement the multiplexers. Ports are anchored to the edge of the module area as defined in the template file using bus macros. Any additional real estate available on the module edge can be used for passthrough signals, which is an option specified in the template file. An example of the HDL source generated for the wrapper structure is shown in Appendix A.6.

Static Logic HDL

The Xilinx PR flow is the most time consuming portion of the compile-time tool flow. In fact, the execution times of all other tools are negligible when compared with the time taken to execute the Xilinx PR flow. To help reduce the execution time of the Xilinx PR flow, all static logic surrounding the partial region was removed. This reduces the execution time because the static system is not synthesized and implemented each time a module is built. However, if no static logic surrounds the partial region the Xilinx tools will produce an error

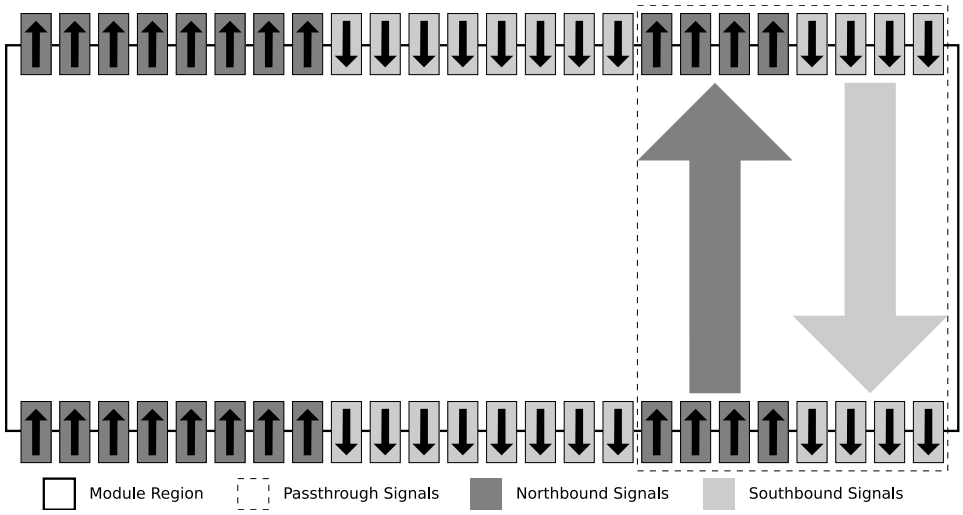

Figure 4.5: Final wrapper structure design

stating that the bus macros are used improperly. The easiest way to alleviate this error is to connect the bus macros to the I/O pins of the FPGA, however this has a negative affect on the final bitstream produced. For reasons explained in Section 4.4, the static logic needs to be optimized away during the Xilinx PR flow. To achieve this behavior, minimal static logic is generated by the preprocessor that connects to each of the input and output ports of the partial region. Instead of connecting to external I/O pins, the static logic acts as a consumer for the outgoing signals from the partial region and produces semi-random data to serve as the incoming signals for the partial region. The static logic gets optimized away because it has no communication with any other part of the static system. An example of the HDL source generated for the static logic is shown in Appendix A.5. It is also important that the static logic generated registers the signals. If there is no synchronous logic in the static design the clock nets will not be enabled and the partial region will be unclocked.

Top Level HDL

The top level module instances the wrapper structure, the static logic, and the bus macros. The top level also contains black-box definitions for each of the modules, including the

wrapper structure, the static logic, and each type of bus macro used. Black-box definitions are necessary to meet one of the requirements of the Xilinx PR flow. Another important part of the top level module is the port connections between the bus macros, the wrapper structure, and static logic. An example of the HDL source generated for the top level module is shown in Appendix A.4.

User Constraints File

By producing the user constraints file, the preprocessor insulates the designer from many low-level FPGA details. The UCF contains clocking information as defined in the template file, separate area groups for the partial region and the rest of the device, and location constraints. The wrapper structure, which contains the IP module, is placed in an area group designated as partially reconfigurable. Partially reconfigurable area groups must have strict location constraints generated in accordance with the template file specification. The static logic is placed in a separate area group that does not require location constraints. The UCF also contains a specific location constraint for each bus macro to locate the bus macro in the correct place along the module edge. Lastly, the UCF provides a clock buffer location to the Xilinx PR tools to prevent a known issue with the Xilinx flow. An example of the UCF generated is shown in Appendix A.7.

4.3.2 Generated Scripts

The preprocessor generates several script files that are necessary for automation of the compile-time tool flow. The scripts include a makefile for moving the necessary bus macros to the synthesis directory (see Appendix A.11), makefiles that automate the Xilinx PR flow (see Appendices A.12 and A.14), and a Perl script (see Appendix A.9) that creates a list of the module source files (see Appendix A.10). Additional files are generated that serve as input files to XST (see Appendix A.13). These input files are not considered source files but

supply additional information XST needs for synthesis. One of the XST input files references the list of source files generated by the Perl script so that XST will synthesize all module source files.

4.3.3 XML Description File

The preprocessor generates an XML file that describes the physical properties of the IP module and the partial bitstream. The description file includes information needed by the post-processing tool as well as the run-time tool flow. Many of the details provided in the template file are included in the description file, including the target device, the module data flow, the module name, and the horizontal clock trunk used. The location and dimensions after the addition of the wrapper structure are also included. A description file lists each special column type and location relative to the module as well as details about each module port (including passthrough signals). The ports are grouped into buses (as defined in the template file) and the direction, name, type, and bit width are provided. The exact location, relative to the module, for each bit of a port is provided. Location information consists of CLB coordinates, the slice index, and which LUT within the slice is used. An example XML description file generated by the preprocessor is shown in Appendix A.8.

4.3.4 Implementation

The preprocessor is implemented using C++ due to the object oriented nature of the language and the existence of compilers for many platforms. Another advantage to using C++ is that many libraries exist for performing common tasks.

Data Storage

The preprocessor parses the template file to gather necessary data about the IP module. Most of the data gathered can be stored in standard data structures provided by the C++ language. However, the template file includes a port list. The number of ports a module has varies between modules, so the data structure needs to be flexible enough to accommodate different size port lists. The data structure used to store the port list is a double-linked list designed specifically for storing port information. The port list class includes a pointer to the head node of the port list and a function for adding a port to the list. This function creates a new port list node and inserts the node in the correct location. The pointers are managed internally, which provides a clean interface. Each port list node contains standard C++ data structures for storing port specific information and two pointers to the previous and next nodes in the list.

Template Parsing

The format of the template file was expected to change as the development of the preprocessor progressed. Designing and implementing a parser using C++ is a tedious and error-prone process to begin with; having to constantly alter the parser would also be tedious and error-prone. Fast Lexical Analyzer (FLEX) is a tool for generating scanners. A scanner, as defined by the FLEX documentation, is a "program which recognizes lexical patterns in text" [31]. The template file parser for the preprocessor was generated using FLEX. Using FLEX reduces the effort required to update the parser with the new template file format. FLEX also introduces error checking and provides an easy platform for syntax checking.

XML Generation

The XML description file produced by the preprocessor is parsed by the postprocessor and the run-time tools. It is important that the XML format is consistent among the three tools. Xerces is a library for parsing, generating, and manipulating XML documents [32]. The preprocessor uses Xerces to generate the XML description file.

Program Flow

The preprocessor begins by parsing the template file for the necessary data describing the IP module. The data gathered from the template file is stored locally and used to calculate information needed for the remainder of the program. The additional information calculated includes port positions, partial region coordinates, and special column coordinates. Once all calculations have been made, the XML description file is generated. Next, the preprocessor generates the HDL files, starting with the top level module. The HDL files for the static logic and wrapper structure are generated next. The scripts that automate the compile-time tool flow are generated one after another when generation of the HDL files has completed. The final step of the preprocessor generates the UCF.

4.4 Postprocessor

The postprocessor was developed to perform alterations to the bitstreams produced by the Xilinx PR flow. Three bitfiles are produced by the Xilinx PR flow: a full bitstream, a partial bitstream, and a blanking bitstream. A partial bitstream includes only the configuration bits necessary to load the partial region of the design. The bus macros are placed such that half of each bus macro lies inside the partial region and the other half lies outside the partial region. A partial bitstream only includes the half of the bus macro inside the partial

region. The run-time tool flow requires the full bus macro to be part of each module, so the partial bitstream will not satisfy this requirement. A full bitstream includes the full design, including the full bus macros. However, as mentioned in Section 4.3.1, the static logic is optimized away during the Xilinx PR flow. The only remaining logic is the partial region and the full bus macros. This is exactly what is needed, but the full bitstream will still not work because it will clear the static FPGA configuration when loaded. The postprocessor was developed to extract a partial bitstream from the full bitstream that includes only the configuration bits for the partial region and full bus macros.

The postprocessor also has the capability of changing the horizontal clock trunk that drives the synchronous logic within the partial region. This is necessary because the clock trunk used in the module build is not necessarily the same clock trunk that was used in the sandbox of the static build. If the clock trunks are not consistent between the two designs, the partial module will be unclocked once placed in the sandbox region.

The run-time tools also have the capability to change horizontal clock trunks so that the same module can be used in several sandbox implementations, each of which could potentially use a different horizontal clock trunk. Before the run-time tools had the capability, the postprocessor was used to modify the clock trunks. The run-time tools also have the capability of extracting the partial bitstream, however the extracted partial bitstream requires far less storage than the full bitstream. Partial bitstreams, as observed during testing, were consistently 90% smaller than full bitstreams. By extracting the partial bitstream at compile-time, the dynamic module library requires less storage.

Chapter 5

Results

5.1 Overview

The WoD flow for flexible run-time reconfiguration has been demonstrated successfully. This chapter presents a test application that uses the WoD flow. The results focus on aspects that demonstrate valid operation of the compile-time tool flow presented in this thesis.

5.2 Test Application

A development board with a Virtex-4 XC4VLX60 FPGA, an Analog to Digital Converter (ADC), and an ARM processor was used as the test platform. Figure 5.1 shows the layout of the test platform. An antenna and an analog low-pass filter were external components to the development board. The ARM processor communicated with external storage and an audio playback client through an Ethernet connection.

The platform was used to implement Amplitude Modulation (AM) radio envelope detection [33]. The modules necessary to implement envelope detection were built using the

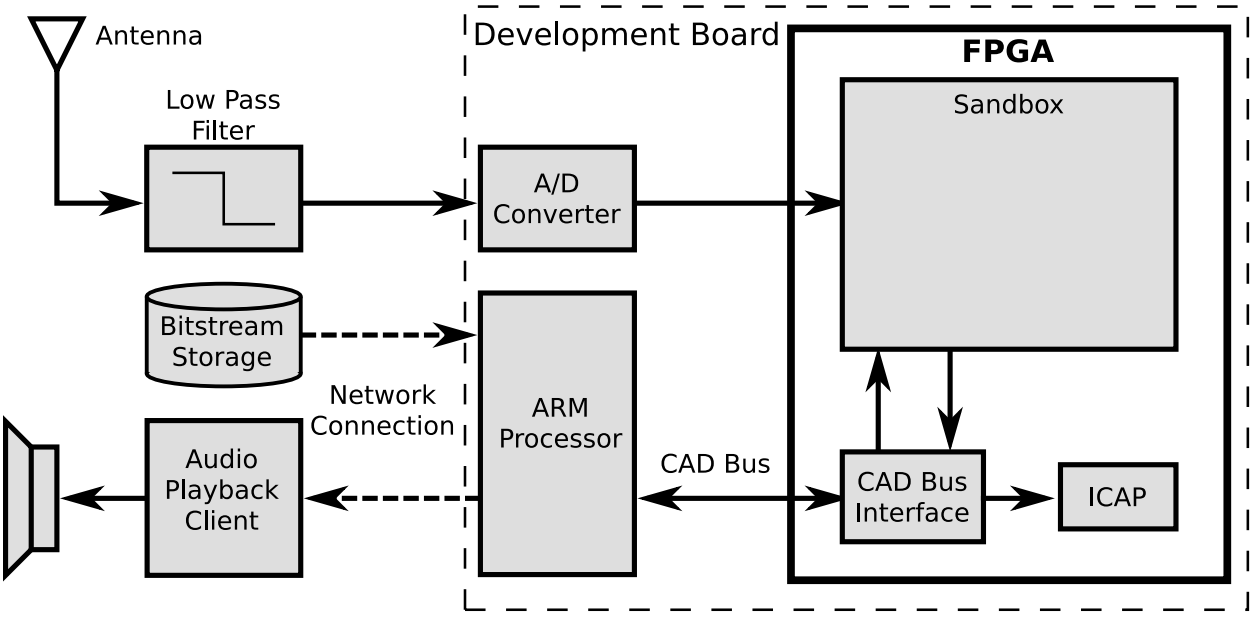

Figure 5.1: Test platform for the AM radio application

automated compile-time tool flow and were used to construct the AM radio at run-time. The AM radio application was chosen to demonstrate the use of the WoD flow for a data flow application.

The analog low-pass filter, shown in Figure 5.1, was used to reduce the bandwidth of the signal into the ADC. The AM spectrum lies between 500 kHz and 1700 kHz; anything outside of that range is not needed. The cut-off frequency of the analog low-pass filter was 1.9 MHz, so the entire AM spectrum was available to the ADC. The output of the ADC, at 12 million samples per second, was connected directly to the FPGA and to the input port of the sandbox region, as shown in Figure 5.1. The AM signal was processed using the FPGA to produce an audible signal of the selected station.

A CAD Bus interface was used to transfer the processed signal from the envelope detector to the ARM processor. The ARM processor was used to broadcast the signal over Ethernet for the audio playback client. The audio playback client, in this case a PC, received the signal and played the audio for verification that the system worked properly.

The WoD run-time tools provided an interface for constructing and modifying a data path using the IP modules available in the dynamic module database. The ARM processor on the development board was used to execute the run-time tools. As shown in Figure 5.1, the run-time tools used network storage to retrieve IP module bitstreams and used the CAD Bus interface to configure the FPGA through the ICAP. A separate program, used to choose the station and broadcast the processed audio over Ethernet, executed on the ARM processor as well. Choosing a station affected control registers for the envelope detector but did not influence the run-time tools. The CAD Bus interface was used to update the control registers for the envelope detector.

5.2.1 AM Radio Design

The first stage of the envelope detector was a band-pass filter that narrowed the frequency, as shown in Figure 5.2. The frequency range of interest was from 500 kHz to 1500 kHz. To reduce the sample frequency from 12 MHz to 4 MHz, the filter also decimated the signal by a factor of three. The purpose of the next stage of the envelope detector was to divide the spectrum at 1200 kHz. This was necessary to prevent lower frequencies from causing interference when a high frequency signal was mixed to the center frequency of the station band-pass filter. A high-pass filter removed lower bands to receive AM stations above 1200 kHz and a low-pass filter removed higher bands to receive stations below 1200 kHz.

A sinusoid generator was used to choose the desired AM station within the available spectrum by changing the frequency of the sinusoid. In the next stage of the envelope detector, the signal from the sinusoid generator was mixed with the AM signal. Mixing the signals moved the desired signal to the center frequency of the station band-pass filter. The next stage of the envelope detector used a decimating low-pass filter to reduce the sampling frequency to 500 kHz. This was done to reduce the requirements of the station band-pass filter in the next stage. With the lower sampling frequency, fewer taps were required and the band-pass filter

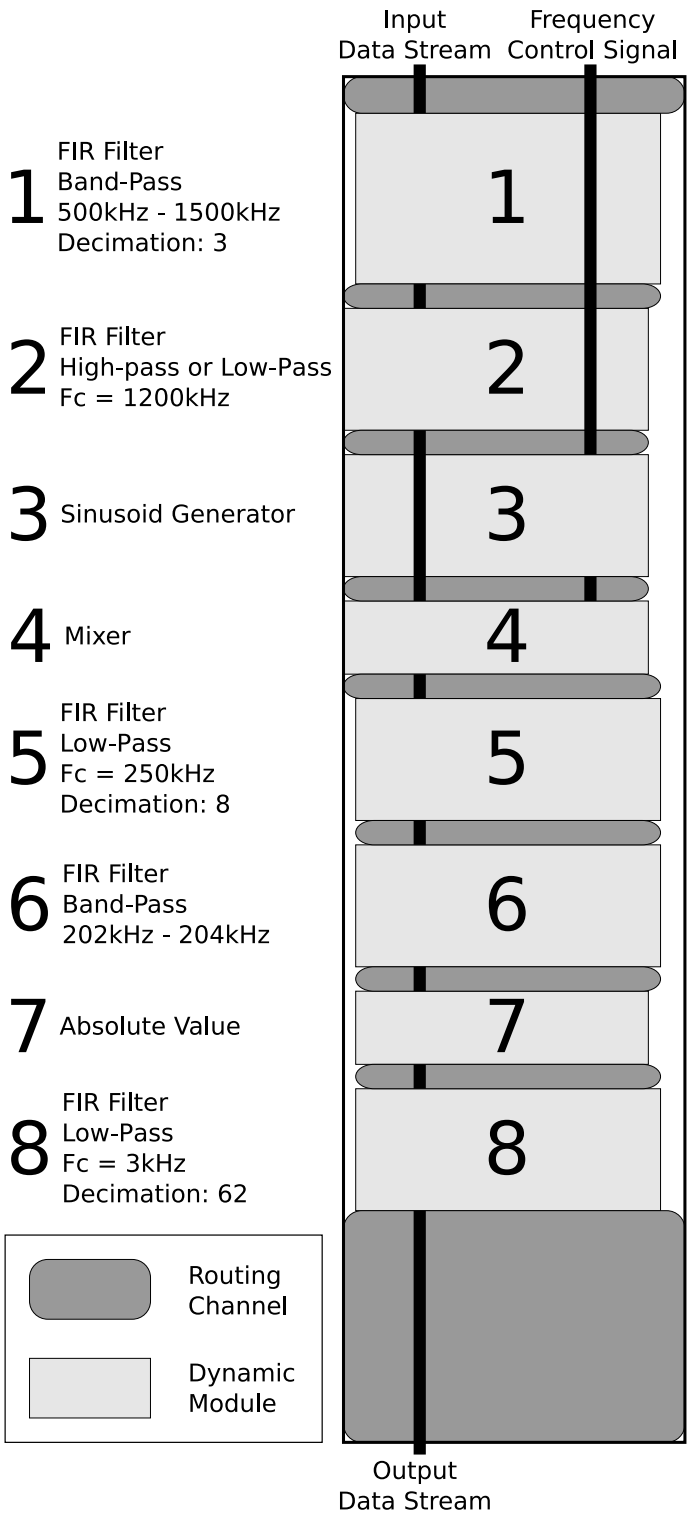

Figure 5.2: FPGA sandbox region and for AM radio application

could take longer to process each sample. The station band-pass filter isolated the signal associated with the desired station by only passing frequencies from 202 kHz to 204 kHz. The signal was brought to baseband in the next stage by taking the absolute value.

The final stage of the envelope detector decimated the signal by a factor of 62 using a low-pass filter. The resulting sample frequency of the signal was 8 kHz, which is a standard wave sound file frequency. This was done so the signal could be played using a standard PC sound card.

5.2.2 FPGA Implementation

The WoD automated compile-time tool flow was used to implement the envelope detection modules for a Virtex-4 XC4VLX60 FPGA. Virtex-4 FPGAs have a center column of IOBs and clock resources that limit the size of the sandbox region to half of the device. In addition, there is only one column of DSPs in the left half of the Virtex-4 part chosen. The sandbox region was placed on the left half of the device because the envelope detection modules require DSP resources. Since there is only a single column of DSPs in the sandbox area, it was not possible to place two modules side-by-side, so modules were designed to take up the full width of the sandbox region.

The majority of the modules have a single input data stream and a single output data stream, however there are two exceptions. The sinusoid generator, shown in Figure 5.2 as the third module, has a single frequency control input and a single data stream output. The frequency is set by the ARM processor each time the user chose a station. Since the sinusoid generator is the third module in the data path, the frequency control signal is routed through passthroughs of the two preceding modules. The output data stream of the sinusoid generator is mixed with the AM signal data stream by the mixer module. The mixer module, shown as the fourth module in Figure 5.2, is the other exception because it mixes two input data streams and produces a single output data stream. The AM signal data stream is

routed through passthroughs of the sinusoid generator to reach the mixer module. All other data stream connections are between neighboring modules.

5.3 Run-time Performance

The AM radio implementation represents a real-time data processing application that benefits from run-time reconfiguration. The design of the system tests all the features of the dynamic modules generated using the WoD compile-time flow. The dynamic modules also have to be efficient in order to process the real-time AM radio signal.

Initial construction of the data path tests loading IP modules from the dynamic module library, placing the modules in locations of the FPGA that are unknown at compile-time, and routing signals to the ports of the IP modules. Loading modules requires data from the XML description files to extract the portion of the partial bitstreams containing the configuration bits for the IP modules. Placing modules requires data from the XML description files to align the IP modules with BRAM and DSP columns within the FPGA fabric. Routing signals to IP module ports tests the XML port information and the bus macro port anchors of the dynamic module bitstreams.

Changing AM stations without changing the second FIR filter in the data path shown in Figure 5.2 tests passthrough signals for the first two modules of the data path. Changing stations and changing the second FIR filter in the data path tests run-time reconfiguration as well as everything previously mentioned. Lastly, initial construction and both methods of changing stations tests the correctness of the bitstreams produced by the WoD compile-time flow. Audio output from the audio playback client successfully matching the AM radio station chosen verifies that no errors have occurred.

Each of the tests described above produced correct results based on examining the output from the audio playback client. Audio playback would be noticeably distorted if an error had

occurred with the WoD flow. Change in playback speed or volume and added clicks or pops are some of the distortions that could occur if errors were made. The system successfully tuned to each AM station chosen and broadcast a live, undistorted audio feed to the audio playback client.

5.4 Compile-time Performance

Performance of the compile-time tools was measured based on execution time and reduction of designer responsibilities. Two IP module designs were used used to illustrate how the tools scale with design size. The first design was an absolute value module similar to the one used in the AM radio application. The second design was a MicroBlaze soft processor. The MicroBlaze processor required more than eight times the resources than the absolute value module required. The statistics are summarized in Table 5.1.

The platform used to measure all performance statistics has a 2.4 GHz Core 2 Quad processor and 2 GB of RAM with a network-mounted file system. As is shown in the table, execution times of the preprocessor and postprocessor are negligible when compared to the execution time of the entire WoD compile-time tool flow. The Xilinx PR flow dominates the total WoD tool flow execution time. The execution time for the preprocessor was actually less for the larger design, which is most likely a result of the network-mounted file system. With execution times far less than one second, network latency affects the execution time more than design size. The postprocesssor showed a fifty percent increase in execution time with the 700 percent increase in design size. These results show that the design size increases an order of magnitude faster than the execution times of the WoD compile-time tools.

The lines of code produced by the designer is a small fraction of the code produced by the WoD compile-time tools. The designer requirements, measured in lines of code, increased by thirty percent with the 700 percent increase in design size. Also, the lines of code produced

	Small Design (absolute value)	**Medium-sized Design (MicroBlaze)**	**Percent Increase**
Slice utilization	179	1499	737
Execution times (seconds)			
Preprocessor	0.020	0.016	N/A
Postprocessor	0.164	0.248	51
Xilinx PR Flow	362.774	551.830	52
Full WoD flow	362.958	552.094	52
User-created template file (lines)	64	85	33
Generated file sizes (lines)			
_top.ucf	28	56	100
_top.v	327	730	123
_pr.v	32	52	63
_static.v	30	36	20
.0.xml	142	366	158
top.xst	13	13	N/A
prm.xst	13	13	N/A
static.xst	12	12	N/A
prm.pl	6	6	N/A
prm.prj	2	2	N/A
pr.make	72	72	N/A
merge.makefile	9	9	N/A
total	686	1367	99

Table 5.1: Compile-time tool flow performance statistics

by the WoD compile-time tools doubled with the 800 percent increase in design size. The compile-time tools help prevent the increased design size from causing a large increase in designer responsibilities.

Chapter 6

Conclusion

6.1 Summary

This thesis presented an automated compile-time tool flow for dynamic module library generation. Background information was provided on the uses, architecture, and configuration of FPGAs and run-time reconfiguration applications. The limitations of current partial reconfiguration tool flows were described, as well as previous research efforts related to dynamic module generation.

It was necessary to describe the WoD run-time system because dynamic module library generation is a compile-time flow intended to support a flexible run-time reconfigurable system. The compile-time tool flow was discussed in detail, including automation, template file format, preprocessor and postprocessor. Finally, a test application was presented with results that show the compile-time tool flow worked properly.

6.2 Achievements

Use of the WoD compile-time flow is necessary for the run-time flexibility provided by the WoD run-time flow and also enables designers to be more productive. Productivity is increased by reducing the lines of code a designer must generate and by automating the build flow for the generation of partial bitstreams. The compile-time flow also eliminates the need for the designer to learn the low-level details of partial reconfiguration and FPGA architecture.

The compile-time tools were developed to address the design objectives described in Section 3.2. The template file format was designed to easily describe IP modules intended for data flow applications and the wrapper structure was designed to support the communication requirements of data flow applications. Designers are insulated from the low-level details of partial reconfiguration by supplying application and IP module details in the template file format. The preprocessor and postprocessor also insulate the designer by generating the necessary HDL and constraints files and by modifying Xilinx-produced bitstreams, respectively. Design automation is provided by the template file and preprocessor and build automation is provided by makefiles and scripts.

Flexible run-time placement and routing are supported by the port anchoring and passthrough connections of the wrapper structure and the XML description files that provide details about each bitfile to the run-time tools. RTR is efficient by using partial bitstreams extracted by the postprocessor. All tools and test applications target the Xilinx Virtex-4 architecture.

6.3 Future Work

The compile-time tool flow presented in this thesis is intended as a proof-of-concept flow. Future work will likely include adding desired functionality to the flow that was not considered

necessary for the proof-of-concept.

Additional functionality could include a redesign of the template file to allow additional options. Also, a more C-like format for the template file, easier support for specifying buses, and support for including information that needs to be part of the XML description file are all possible improvements. The ability to have a global template file for details that remain consistent throughout several IP modules would be a convenient addition.

The preprocessor could also be improved by validating the information gathered from the template file. Allowing more flexibility for including passthrough signals is another possible improvement. As the WoD project continues at Virginia Tech, there will likely be more updates to the compile-time design flow.

6.4 Conclusion

The run-time tool flow has proven that the compile-time tool flow presented in this thesis is a necessary and successful contribution to the WoD flow. The flow addressed the lack of compile-time automation and run-time flexibility that exists in alternative run-time reconfiguration flows. The results have shown the ability to dynamically place IP modules and synthesize communication at run-time.

Bibliography

[1] C. Patterson, "High performance DES encryption in Virtex FPGAs using JBits," *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pp. 113–121, 2000.

[2] G. Rouvroy, F.-X. Standaert, J.-J. Quisquater, and J.-D. Legat, "Compact and efficient encryption/decryption module for FPGA implementation of the AES Rijndael very well suited for small embedded applications," *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, vol. 2, pp. 583–587 Vol.2, 5-7 April 2004.

[3] V. K. Prasanna and A. Dandalis, "FPGA-based cryptography for internet security," *Online Symposium for Electronic Engineers*, November 2000.

[4] S. Kumar, C. Paar, J. Pelzl, erd Pfeiffer, and M. Schimmler, "Breaking ciphers with COPACOBANA: a cost-optimized parallel code breaker," *Cryptographic Hardware and Embedded Systems - CHES 2006*, vol. 4249/2006, pp. 101–118, 2006.

[5] M. Uhm, "Software-defined radio: The new architectural paradigm," *DSP Magazine*, vol. 1, pp. 40–42, Oct. 2005.

[6] V. Seonil Choi; Govindu, G.; Ju-Wook Jang; Prasanna, "Energy-efficient and parameterized designs for Fast Fourier Transform on FPGAs," *Acoustics, Speech, and Signal*

Processing, 2003. Proceedings. (ICASSP '03). 2003 IEEE International Conference on, vol. 2, pp. II–521–4 vol.2, 6-10 April 2003.

[7] R. Scrofano, J.-W. Jang, and V. Prasanna, "Energy-efficient Discrete Cosine Transform on FPGAs," *Proc. Int'l Conf. Eng. of Reconfigurable Systems and Algorithms (ERSA '03)*, pp. 215–221, 2003.

[8] H. H.-J. Khan, U.R.; Owen, "FPGA architectures for ASIC hardware emulators," *ASIC Conference and Exhibit, 1993. Proceedings., Sixth Annual IEEE International*, pp. 336–340, 27 Sep-1 Oct 1993.

[9] D. Langen, J.-C. Niemann, M. Porrmann, H. Kalte, and U. Rckert, "Implementation of a RISC processor core for SoC designs: FPGA prototype vs. ASIC implementation," *,In Proceedings of the IEEE-Workshop: Heterogeneous reconfigurable Systems on Chip (SoC)*, April 2002.

[10] J.-W. Jang, S. B. Choi, and V. K. Prasanna, "Energy- and time-efficient matrix multiplication on FPGAs," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 13, no. 11, pp. 1305–1319, 2005.

[11] T.-Y. Yen and W. Wolf, "Communication synthesis for distributed embedded systems," in *ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 288–294.

[12] P. Athanas, J. Bowen, T. Dunham, C. Patterson, J. Rice, M. Shelburne, J. Suris, M. Bucciero, and J. Graf, "Wires on Demand: Run-time communication synthesis for reconfigurable computing," *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 513–516, 27-29 Aug. 2007.

[13] U. Meyer-Baese and U. M. Baese, *Digital Signal Processing with Field Programmable Gate Arrays with Cdrom*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001.

[14] S. Brown and J. Rose, "FPGA and CPLD architectures: a tutorial," *Design & Test of Computers, IEEE*, vol. 13, no. 2, pp. 42–57, Summer 1996.

[15] K. Morris, "FPGA BASE jump: Partial reconfiguration for SDR," *FPGA and Structured ASIC Journal*, November 2007. [Online]. Available: www.fpgajournal.com/articles_2007/pdf/20071106_basejump.pdf

[16] *Virtex-4 User Guide*, Xilinx, August 2007. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug070.pdf

[17] *Virtex-4 Family Overview*, Xilinx, September 2007. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf

[18] E. L. Horta and J. W. Lockwood, "Automated method to generate bitstream intellectual property cores for virtex FPGAs," *Field Programmable Logic and Application*, vol. 3203/2004, pp. 975–979, 2004.

[19] *Early Access Partial Reconfiguration User Guide*, Xilinx, March 2006. [Online]. Available: http://www.xilinx.com/support/prealounge/protected/docs/ug208.pdf

[20] S. Raaijmakers and S. Wong, "Run-time partial reconfiguration for removal, placement and routing on the Virtex-II Pro," in *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07)*, August 2007.

[21] *Xilinx ISE 9.2i Software Manuals and Help*, Xilinx, 2007. [Online]. Available: http://toolbox.xilinx.com/docsan/xilinx92/books/manuals.pdf

[22] J. H. Reed, *Software Radio: A Modern Approach to Radio Engineering*. Upper Saddle River, NJ: Prentice Hall PTR, 2002.

[23] W. Tuttlebee, "Software-defined radio: facets of a developing technology," *Personal Communications, IEEE*, vol. 6, no. 2, pp. 38–44, Apr 1999.

[24] D. Maliniak, "Interconnect IP steers SoC integration into the fast lane," *Electronic Design*, November 2002. [Online]. Available: http://electronicdesign.com/Articles/ArticleID/1814/1814.html

[25] R. Bergamaschi, S. Bhattacharya, R. Wagner, C. Fellenz, M. Muhlada, F. White, J.-M. Daveau, and W. Lee, "Automating the design of SOCs using cores," *Design & Test of Computers, IEEE*, vol. 18, no. 5, pp. 32–45, Sep-Oct 2001.

[26] E. Carvalho, N. Calazans, a. Eduardo Bri and F. Moraes, "PaDReH: a framework for the design and implementation of dynamically and partially reconfigurable systems," in *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*. New York, NY, USA: ACM, 2004, pp. 10–15.

[27] S. Koh and O. Diessel, "COMMA: a communications methodology for dynamic module reconfiguration in FPGAs," *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pp. 273–274, April 2006.

[28] E. L. Horta and J. W. Lockwood, "PARBIT: a tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (FPGAs)," Washington University, Tech. Rep., July 2001.

[29] P. Sedcole, B. Blodget, J. Anderson, P. Lysaghi, and T. Becker, "Modular partial reconfigurable in Virtex FPGAs," *Field Programmable Logic and Applications, 2005. International Conference on*, pp. 211–216, 24-26 Aug. 2005.

[30] M. Ullmann, M. Hbner, B. Grimm, and J. Becker, "On-demand FPGA run-time system for dynamical reconfiguration with adaptive priorities," *Field Programmable Logic and Application*, vol. 3203/2004, pp. 454–463, 2004.

[31] "flex: The fast lexical analyzer." [Online]. Available: http://flex.sourceforge.net/

[32] "Xerces-C++ parser." [Online]. Available: http://xerces.apache.org/xerces-c/

[33] S. A. Tretter, *Communication system design using DSP algorithms.* New York, NY, USA: Plenum Press, 1995.

Appendix A

A.1 Absolute Value Module Verilog Definition

```
// AbsVal.v
module AbsVal (clock,ce,reset,dataIn,inWrite,dataOut,outWrite);
   input clock;
   input ce;
   input reset;
   input [15:0] dataIn;
   input inWrite;
   output [15:0] dataOut;
   output outWrite;
endmodule
```

A.2 Absolute Value Module Template File

```
# template.tmp

global {
   part        xc4vlx60-ff668-10
   bufg        BUFGCTRL_X0Y0
   iostandard  LVCMOS25
   timespec    [TIMESPEC "TS_clock" = PERIOD "clock" 10.00 ns HIGH 50%;]
   hTrunkLine  0
}
module {
   modulename     AbsVal
   filename       AbsVal.v
   dimensions     X16Y8
   dataflow       SOUTH
   version        0
   passthroughs   include
}
restrictions {
   location    X4Y9
}
ports {
   port {
      name     clock
      type     clock
      wrap     exclude
      bits     1
      dir      input
   }
   port {
      name     ce
      type     ce
      bits     1
      dir      input
      wrap     include
      dataflow with
   }
   port {
      name     reset
```

```
      type     reset
      bits     1
      dir      input
      wrap     include
      dataflow with
   }
   port {
      name     dataIn
      type     data
      bits     16
      dir      input
      wrap     include
      dataflow with
   }
   port {
      name     inWrite
      type     write
      bits     1
      dir      input
      wrap     include
      dataflow with
   }
   port {
      name     dataOut
      type     data
      bits     16
      dir      output
      wrap     include
      dataflow with
   }
   port {
      name     outWrite
      type     write
      bits     1
      dir      output
      wrap     include
      dataflow with
   }
}
```

A.3 Absolute Value Module Makefile

```
# Makefile

MOD_NAME = AbsVal
TEMPLATE = template.tmp

PreP = preprocessor
PostP = modPrep
BIN_PATH = /project/defrag/modules/build/bin
ADDL_SRC =

###########################################################
# makefile is the same for all modules below this point
###########################################################################

PreP_EXE = $(abspath $(BIN_PATH)/$(PreP))
PostP_EXE = $(abspath $(BIN_PATH)/$(PostP))

# Identify desired source file types in the specified directories
SRC_HDL = $(foreach dir,$(ADDL_SRC),$(wildcard $(dir)/*.v))
SRC_NETLISTS = $(foreach dir,$(ADDL_SRC),$(wildcard $(dir)/*.edn $(dir)/*.ngc))


##########
# make all
all: organize process bitstreams postprocess

##########
# create source files and move to correct directories
process: organize
cp source/$(MOD_NAME).v hdl/pr
-cp source/*.v hdl/pr
-cp source/$(TEMPLATE) $(PreP)/
-cp source/*.ngc synth/pr/
-cp source/*.edn synth/pr/
-cp -p -t hdl/pr/ $(SRC_HDL)
-cp -p -t synth/pr/ $(SRC_NETLISTS)
cd $(PreP) && $(PreP_EXE) $(TEMPLATE)
-mv $(PreP)/$(MOD_NAME)_pr.v hdl/pr
```

```
-mv $(PreP)/$(MOD_NAME)_static.v hdl/static
-mv $(PreP)/$(MOD_NAME)_top.v hdl/top
-mv $(PreP)/pr.make scripts
-mv $(PreP)/static.xst scripts
-mv $(PreP)/top.xst scripts
-mv $(PreP)/prm.xst scripts
-chmod 777 $(PreP)/prj.pl
-mv $(PreP)/prj.pl scripts
-cd hdl/pr && ../../scripts/prj.pl > ../../scripts/prm.prj
-mv $(PreP)/makefile merges
-mv $(PreP)/$(MOD_NAME)_top.ucf data
-mv $(PreP)/$(MOD_NAME)_top_bb.ucf data
-mv $(PreP)/bm.make .
-make -f bm.make

##########
# create directory structure
organize:
test -d data || mkdir data
test -d hdl || mkdir hdl
test -d hdl/pr || mkdir hdl/pr
test -d hdl/static || mkdir hdl/static
test -d hdl/top || mkdir hdl/top
test -d merges || mkdir merges
test -d pr || mkdir pr
test -d scripts || mkdir scripts
test -d static || mkdir static
test -d synth || mkdir synth
test -d synth/busmacros || mkdir synth/busmacros
test -d synth/chipscope || mkdir synth/chipscope
test -d synth/pr || mkdir synth/pr
test -d synth/static || mkdir synth/static
test -d synth/top || mkdir synth/top
test -d $(PreP) || mkdir $(PreP)
test -d $(PostP) || mkdir $(PostP)
test -d final || mkdir final

##########
# generate bitstreams (using xilinx tools)

CP_XML = $(patsubst $(PreP)/%.xml,%.ncd,$(wildcard $(PreP)/*.*.xml))
```

```
bitstreams: process
cd scripts && make -f pr.make
cp $(PreP)/*.xml final
-cp merges/system_static_full.ncd final/$(CP_XML)
cp $(PreP)/*.xml $(PostP)
cp merges/system_static_full.bit $(PostP)

XML_FILE = $(notdir $(wildcard $(PostP)/*.*.xml))
RENAME_BIT = $(patsubst $(PreP)/%.xml,%.bit,$(wildcard $(PreP)/*.*.xml))

postprocess: bitstreams
cp $(BIN_PATH)/*.csv $(PostP)
cd $(PostP) && $(PostP_EXE) -i $(XML_FILE) system_static_full.bit
cp $(PostP)/system_static_full_prepped.bit final/$(RENAME_BIT)


##########
# delete generated files/folders
clean:
-rm -r data
-rm -r hdl
-rm -r merges
-rm -r pr
-rm -r scripts
-rm -r synth
-rm -r static
-rm -r $(PreP)
-rm -r $(PostP)
-rm bm.make
```

A.4 Absolute Value Top HDL

```
// AbsVal_top.v

module AbsVal_top(clock);
input clock;

wire [23:0] top2busmacro_1;
wire [23:0] busmacro2module_1;
wire [23:0] busmacro2top_0;
wire [23:0] module2busmacro_0;
wire [55:0] busmacro2passthrough_0to1;
wire [55:0] passthrough2busmacro_0to1;
wire [55:0] passthrough_input_0;
wire [55:0] passthrough_output_1;
wire [47:0] busmacro2passthrough_1to0;
wire [47:0] passthrough2busmacro_1to0;
wire [47:0] passthrough_input_1;
wire [47:0] passthrough_output_0;

wire dummy_output_sig_0;
wire dummy_output_sig_1;
reg dummy_input_sig_reg_0;
reg dummy_input_sig_reg_1;

//WRAPPED MODULE INSTANCE
AbsVal_pr AbsVal_pr_inst (
.clock(clock),
.busmacro2module_1(busmacro2module_1),
.module2busmacro_0(module2busmacro_0),
.passthrough_input_0(busmacro2passthrough_0to1),
.passthrough_output_1(passthrough2busmacro_0to1),
.passthrough_input_1(busmacro2passthrough_1to0),
.passthrough_output_0(passthrough2busmacro_1to0)
);
//synthesis attribute keep_hierarchy of AbsVal_pr_inst is true;

busmacro_xc4v_t2b_sync_narrow BM_inst_output0_00 (
.ce0(1'b1),.clk0(clock),
.ce1(1'b1),.clk1(clock),
```

```
.ce2(1'b1),.clk2(clock),
.ce3(1'b1),.clk3(clock),
.input0(module2busmacro_0[0]),.output0(busmacro2top_0[0]),
.input1(module2busmacro_0[1]),.output1(busmacro2top_0[1]),
.input2(module2busmacro_0[2]),.output2(busmacro2top_0[2]),
.input3(module2busmacro_0[3]),.output3(busmacro2top_0[3]),
.input4(module2busmacro_0[4]),.output4(busmacro2top_0[4]),
.input5(module2busmacro_0[5]),.output5(busmacro2top_0[5]),
.input6(module2busmacro_0[6]),.output6(busmacro2top_0[6]),
.input7(module2busmacro_0[7]),.output7(busmacro2top_0[7])
);

busmacro_xc4v_t2b_sync_narrow BM_inst_input1_00 (
.ce0(1'b1),.clk0(clock),
.ce1(1'b1),.clk1(clock),
.ce2(1'b1),.clk2(clock),
.ce3(1'b1),.clk3(clock),
.input0(top2busmacro_1[0]),.output0(busmacro2module_1[0]),
.input1(top2busmacro_1[1]),.output1(busmacro2module_1[1]),
.input2(top2busmacro_1[2]),.output2(busmacro2module_1[2]),
.input3(top2busmacro_1[3]),.output3(busmacro2module_1[3]),
.input4(top2busmacro_1[4]),.output4(busmacro2module_1[4]),
.input5(top2busmacro_1[5]),.output5(busmacro2module_1[5]),
.input6(top2busmacro_1[6]),.output6(busmacro2module_1[6]),
.input7(top2busmacro_1[7]),.output7(busmacro2module_1[7])
);

\\ module port bus macros continue for output0_XX and input1_XX
\\ where XX = 01,02

busmacro_xc4v_b2t_sync_narrow BM_inst_passin0_00 (
.ce0(1'b1),.clk0(clock),
.ce1(1'b1),.clk1(clock),
.ce2(1'b1),.clk2(clock),
.ce3(1'b1),.clk3(clock),
.input0(passthrough_input_0[0]),.output0(busmacro2passthrough_0to1[0]),
.input1(passthrough_input_0[1]),.output1(busmacro2passthrough_0to1[1]),
.input2(passthrough_input_0[2]),.output2(busmacro2passthrough_0to1[2]),
.input3(passthrough_input_0[3]),.output3(busmacro2passthrough_0to1[3]),
.input4(passthrough_input_0[4]),.output4(busmacro2passthrough_0to1[4]),
.input5(passthrough_input_0[5]),.output5(busmacro2passthrough_0to1[5]),
.input6(passthrough_input_0[6]),.output6(busmacro2passthrough_0to1[6]),
```

```
.input7(passthrough_input_0[7]),.output7(busmacro2passthrough_0to1[7])
);

busmacro_xc4v_b2t_sync_narrow BM_inst_passout1_00 (
.ce0(1'b1),.clk0(clock),
.ce1(1'b1),.clk1(clock),
.ce2(1'b1),.clk2(clock),
.ce3(1'b1),.clk3(clock),
.input0(passthrough2busmacro_0to1[0]),.output0(passthrough_output_1[0]),
.input1(passthrough2busmacro_0to1[1]),.output1(passthrough_output_1[1]),
.input2(passthrough2busmacro_0to1[2]),.output2(passthrough_output_1[2]),
.input3(passthrough2busmacro_0to1[3]),.output3(passthrough_output_1[3]),
.input4(passthrough2busmacro_0to1[4]),.output4(passthrough_output_1[4]),
.input5(passthrough2busmacro_0to1[5]),.output5(passthrough_output_1[5]),
.input6(passthrough2busmacro_0to1[6]),.output6(passthrough_output_1[6]),
.input7(passthrough2busmacro_0to1[7]),.output7(passthrough_output_1[7])
);

\\ passthrough bus macros continue for passin0_XX and passout1_XX
\\ where XX = 01,02,03,04,05,06

busmacro_xc4v_t2b_sync_narrow BM_inst_passin1_00 (
.ce0(1'b1),.clk0(clock),
.ce1(1'b1),.clk1(clock),
.ce2(1'b1),.clk2(clock),
.ce3(1'b1),.clk3(clock),
.input0(passthrough_input_1[0]),.output0(busmacro2passthrough_1to0[0]),
.input1(passthrough_input_1[1]),.output1(busmacro2passthrough_1to0[1]),
.input2(passthrough_input_1[2]),.output2(busmacro2passthrough_1to0[2]),
.input3(passthrough_input_1[3]),.output3(busmacro2passthrough_1to0[3]),
.input4(passthrough_input_1[4]),.output4(busmacro2passthrough_1to0[4]),
.input5(passthrough_input_1[5]),.output5(busmacro2passthrough_1to0[5]),
.input6(passthrough_input_1[6]),.output6(busmacro2passthrough_1to0[6]),
.input7(passthrough_input_1[7]),.output7(busmacro2passthrough_1to0[7])
);

busmacro_xc4v_t2b_sync_narrow BM_inst_passout0_00 (
.ce0(1'b1),.clk0(clock),
.ce1(1'b1),.clk1(clock),
.ce2(1'b1),.clk2(clock),
.ce3(1'b1),.clk3(clock),
.input0(passthrough2busmacro_1to0[0]),.output0(passthrough_output_0[0]),
```

```
.input1(passthrough2busmacro_1to0[1]),.output1(passthrough_output_0[1]),
.input2(passthrough2busmacro_1to0[2]),.output2(passthrough_output_0[2]),
.input3(passthrough2busmacro_1to0[3]),.output3(passthrough_output_0[3]),
.input4(passthrough2busmacro_1to0[4]),.output4(passthrough_output_0[4]),
.input5(passthrough2busmacro_1to0[5]),.output5(passthrough_output_0[5]),
.input6(passthrough2busmacro_1to0[6]),.output6(passthrough_output_0[6]),
.input7(passthrough2busmacro_1to0[7]),.output7(passthrough_output_0[7])
);

\\ passthrough bus macros continue for passin1_XX and passout0_XX
\\ where XX = 01,02,03,04,05

dummy_mod dummy_mod_inst (
.dummy_input_sig_0(dummy_input_sig_reg_0),
.dummy_output_sig_0(dummy_output_sig_0),
.dummy_input_sig_1(dummy_input_sig_reg_1),
.dummy_output_sig_1(dummy_output_sig_1),
.busmacro2top_0(busmacro2top_0),
.passthrough_input_0(passthrough_input_0),
.passthrough_output_1(passthrough_output_1),
.passthrough_input_1(passthrough_input_1),
.passthrough_output_0(passthrough_output_0),
.top2busmacro_1(top2busmacro_1)
);
//synthesis attribute keep_hierarchy of dummy_mod_inst_0 is true;

always @(posedge clock) begin
dummy_input_sig_reg_0 <= dummy_output_sig_0;
dummy_input_sig_reg_1 <= dummy_output_sig_1;
end

endmodule

//DUMMY MODULE BLACK BOX
//==============================
module dummy_mod(
dummy_input_sig_0,dummy_output_sig_0,
dummy_input_sig_1,dummy_output_sig_1,
busmacro2top_0,passthrough_input_0,passthrough_output_1,
passthrough_output_0,passthrough_input_1,top2busmacro_1
);
input  dummy_input_sig_0;
```

```
output  dummy_output_sig_0;
input dummy_input_sig_1;
output dummy_output_sig_1;
input [23:0] busmacro2top_0;
output [55:0] passthrough_input_0;
input [55:0] passthrough_output_1;
input [47:0] passthrough_output_0;
output [47:0] passthrough_input_1;
output [23:0] top2busmacro_1;
endmodule

//BUS MACRO BLACK BOXES
//==============================
module busmacro_xc4v_t2b_sync_narrow(
ce0,clk0,
ce1,clk1,
ce2,clk2,
ce3,clk3,
input0,output0,
input1,output1,
input2,output2,
input3,output3,
input4,output4,
input5,output5,
input6,output6,
input7,output7
);
input ce0, ce1, ce2, ce3, clk0, clk1, clk2, clk3;
input input0, input1, input2, input3, input4, input5, input6, input7;
output output0, output1, output2, output3, output4, output5, output6, output7;
endmodule

module busmacro_xc4v_b2t_sync_narrow(
ce0,clk0,
ce1,clk1,
ce2,clk2,
ce3,clk3,
input0,output0,
input1,output1,
input2,output2,
input3,output3,
input4,output4,
```

```
input5,output5,
input6,output6,
input7,output7
);
input ce0, ce1, ce2, ce3, clk0, clk1, clk2, clk3;
input input0, input1, input2, input3, input4, input5, input6, input7;
output output0, output1, output2, output3, output4, output5, output6, output7;
endmodule

//WRAPPED PR MODULE BLACK BOX
//============================
module AbsVal_pr(
clock,
busmacro2module_1,module2busmacro_0,
passthrough_input_0,passthrough_output_1,
passthrough_input_1,passthrough_output_0
);
input clock;
input [23:0] busmacro2module_1;
output [23:0] module2busmacro_0;
input [55:0] passthrough_input_0;
output [55:0] passthrough_output_1;
input [47:0] passthrough_input_1;
output [47:0] passthrough_output_0;
endmodule
```

A.5 Absolute Value Static HDL

```
// AbsVal_static.v

module dummy_mod(
dummy_input_sig_0,dummy_output_sig_0,
dummy_input_sig_1,dummy_output_sig_1,
busmacro2top_0,passthrough_input_0,passthrough_output_1,
passthrough_output_0,passthrough_input_1,top2busmacro_1
);
input  dummy_input_sig_0;
output  dummy_output_sig_0;
input dummy_input_sig_1;
output dummy_output_sig_1;
input [23:0] busmacro2top_0;
output [55:0] passthrough_input_0;
input [55:0] passthrough_output_1;
input [47:0] passthrough_output_0;
output [47:0] passthrough_input_1;
output [23:0] top2busmacro_1;
wire result_0, result_1;
assign result_0 = &({busmacro2top_0,passthrough_output_0,dummy_input_sig_1});
assign result_1 = &({passthrough_output_1,dummy_input_sig_0});

assign passthrough_input_0 = 56'd3 + dummy_input_sig_1;

assign top2busmacro_1 = 24'd3 + dummy_input_sig_0;
assign passthrough_input_1 = 48'd3 + dummy_input_sig_0;

assign dummy_output_sig_0 = result_0;
assign dummy_output_sig_1 = result_1;

endmodule
```

A.6 Absolute Value Wrapper Structure HDL

```
// AbsVal_pr.v

module AbsVal_pr(
clock,
busmacro2module_1,module2busmacro_0,
passthrough_input_0,passthrough_output_1,
passthrough_input_1,passthrough_output_0
);

input clock;
input [23:0] busmacro2module_1;
output [23:0] module2busmacro_0;
input [55:0] passthrough_input_0;
output [55:0] passthrough_output_1;
input [47:0] passthrough_input_1;
output [47:0] passthrough_output_0;

assign passthrough_output_1 = passthrough_input_0; // passthrough 0to1
assign passthrough_output_0 = passthrough_input_1; // passthrough 1to0

wire ce_int;
wire reset_int;
wire [15:0] dataIn_int;
wire inWrite_int;
wire [15:0] dataOut_int;
wire outWrite_int;

assign ce_int = busmacro2module_1[0];
assign reset_int = busmacro2module_1[1];
assign dataIn_int = busmacro2module_1[17:2];
assign inWrite_int = busmacro2module_1[18];
assign module2busmacro_0[15:0] = dataOut_int;
assign module2busmacro_0[16] = outWrite_int;

AbsVal AbsVal_inst (
.clock(clock),
.ce(ce_int),
.reset(reset_int),
```

```
.dataIn(dataIn_int),
.inWrite(inWrite_int),
.dataOut(dataOut_int),
.outWrite(outWrite_int)
);

//synthesis attribute keep_hierarchy of AbsVal_inst is true

endmodule
```

A.7 Absolute Value UCF

```
// AbsVal_top.ucf

NET "clock" IOSTANDARD = LVCMOS25;
NET "clock" TNM_NET = "clock";
TIMESPEC "TS_clock" = PERIOD "clock" 10.00 ns HIGH 50%;

AREA_GROUP "AG_PR_MODULE" GROUP=CLOSED;
AREA_GROUP "AG_PR_MODULE" MODE=RECONFIG;
AREA_GROUP "AG_PR_MODULE" RANGE = SLICE_X8Y16:SLICE_X39Y35;
INST "AbsVal_pr_inst" AREA_GROUP = "AG_PR_MODULE";

AREA_GROUP "AG_PR_MODULE" RANGE = RAMB16_X1Y2:RAMB16_X1Y3;

AREA_GROUP "AG_PR_MODULE" RANGE = DSP48_X0Y4:DSP48_X0Y7;

AREA_GROUP "AG_DUMMY" RANGE = SLICE_X8Y40:SLICE_X39Y43;
INST "dummy_mod_inst" AREA_GROUP = "AG_DUMMY";

INST "BM_inst_output0_00" LOC = "SLICE_X8Y16";
INST "BM_inst_output0_01" LOC = "SLICE_X10Y16";
INST "BM_inst_output0_02" LOC = "SLICE_X12Y16";
INST "BM_inst_input1_00" LOC = "SLICE_X8Y36";
INST "BM_inst_input1_01" LOC = "SLICE_X10Y36";
INST "BM_inst_input1_02" LOC = "SLICE_X12Y36";
INST "BM_inst_passin0_00" LOC = "SLICE_X14Y16";
INST "BM_inst_passout1_00" LOC = "SLICE_X14Y36";
INST "BM_inst_passin0_01" LOC = "SLICE_X16Y16";
INST "BM_inst_passout1_01" LOC = "SLICE_X16Y36";
INST "BM_inst_passin0_02" LOC = "SLICE_X18Y16";
INST "BM_inst_passout1_02" LOC = "SLICE_X18Y36";
INST "BM_inst_passin0_03" LOC = "SLICE_X20Y16";
INST "BM_inst_passout1_03" LOC = "SLICE_X20Y36";
INST "BM_inst_passin0_04" LOC = "SLICE_X22Y16";
INST "BM_inst_passout1_04" LOC = "SLICE_X22Y36";
INST "BM_inst_passin0_05" LOC = "SLICE_X24Y16";
INST "BM_inst_passout1_05" LOC = "SLICE_X24Y36";
INST "BM_inst_passin0_06" LOC = "SLICE_X26Y16";
INST "BM_inst_passout1_06" LOC = "SLICE_X26Y36";
```

```
INST "BM_inst_passin1_00" LOC = "SLICE_X28Y36";
INST "BM_inst_passout0_00" LOC = "SLICE_X28Y16";
INST "BM_inst_passin1_01" LOC = "SLICE_X30Y36";
INST "BM_inst_passout0_01" LOC = "SLICE_X30Y16";
INST "BM_inst_passin1_02" LOC = "SLICE_X32Y36";
INST "BM_inst_passout0_02" LOC = "SLICE_X32Y16";
INST "BM_inst_passin1_03" LOC = "SLICE_X34Y36";
INST "BM_inst_passout0_03" LOC = "SLICE_X34Y16";
INST "BM_inst_passin1_04" LOC = "SLICE_X36Y36";
INST "BM_inst_passout0_04" LOC = "SLICE_X36Y16";
INST "BM_inst_passin1_05" LOC = "SLICE_X38Y36";
INST "BM_inst_passout0_05" LOC = "SLICE_X38Y16";

##########
# CHANGE BUFG LOC IN TEMPLATE FILE IF ERROR DURING BUILD PROCESS
INST "clock_BUFGP/BUFG" LOC = "BUFGCTRL_X0Y0";
```

A.8 Absolute Value XML Description File

```
// AbsVal.0.xml

<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Module Device="xc4vlx60" Flow="SOUTH" Height="12" Name="AbsVal"
     OriginX="4" OriginY="7" Width="16" YOffset="3" hTrunkLine="0">

  <SpecialColumn Type="BRAM" XOffset="4"/>
  <SpecialColumn Type="DSP" XOffset="8"/>

  <Port Direction="input" Name="clock" Type="CLOCK" Width="1"/>

  <Port Direction="input" Name="ce" Type="ce" Width="1">
    <Bit Index="0" Lut="G" Slice="3" XOffset="0" YOffset="11"/>
  </Port>

  <Port Direction="input" Name="reset" Type="RESET" Width="1">
    <Bit Index="0" Lut="F" Slice="3" XOffset="0" YOffset="11"/>
  </Port>

  <Port Direction="input" Name="dataIn" Type="data" Width="16">
    <Bit Index="0" Lut="G" Slice="2" XOffset="0" YOffset="11"/>
    <Bit Index="1" Lut="F" Slice="2" XOffset="0" YOffset="11"/>
    <Bit Index="2" Lut="G" Slice="1" XOffset="0" YOffset="11"/>
    ...
    <Bit Index="15" Lut="F" Slice="3" XOffset="2" YOffset="11"/>
  </Port>

  <Port Direction="input" Name="inWrite" Type="write" Width="1">
    <Bit Index="0" Lut="G" Slice="2" XOffset="2" YOffset="11"/>
  </Port>

  <Port Direction="output" Name="dataOut" Type="data" Width="16">
    <Bit Index="0" Lut="G" Slice="3" XOffset="0" YOffset="0"/>
    <Bit Index="1" Lut="F" Slice="3" XOffset="0" YOffset="0"/>
    <Bit Index="2" Lut="G" Slice="2" XOffset="0" YOffset="0"/>
    ...
    <Bit Index="15" Lut="F" Slice="0" XOffset="1" YOffset="0"/>
  </Port>
```

```
  <Port Direction="output" Name="outWrite" Type="write" Width="1">
    <Bit Index="0" Lut="G" Slice="3" XOffset="2" YOffset="0"/>
  </Port>

  <Port Direction="input" Name="passthrough_north_in"
       Type="passthrough_north" Width="56">
    <Bit Index="0" Lut="G" Slice="3" XOffset="3" YOffset="0"/>
    <Bit Index="1" Lut="F" Slice="3" XOffset="3" YOffset="0"/>
    <Bit Index="2" Lut="G" Slice="2" XOffset="3" YOffset="0"/>
    ...
    <Bit Index="55" Lut="F" Slice="0" XOffset="9" YOffset="0"/>
  </Port>

  <Port Direction="output" Name="passthrough_north_out"
       Type="passthrough_north" Width="56">
    <Bit Index="0" Lut="G" Slice="3" XOffset="3" YOffset="11"/>
    <Bit Index="1" Lut="F" Slice="3" XOffset="3" YOffset="11"/>
    <Bit Index="2" Lut="G" Slice="2" XOffset="3" YOffset="11"/>
    ...
    <Bit Index="55" Lut="F" Slice="0" XOffset="9" YOffset="11"/>
  </Port>

  <Port Direction="input" Name="passthrough_south_in"
       Type="passthrough_south" Width="48">
    <Bit Index="0" Lut="G" Slice="3" XOffset="10" YOffset="11"/>
    <Bit Index="1" Lut="F" Slice="3" XOffset="10" YOffset="11"/>
    <Bit Index="2" Lut="G" Slice="2" XOffset="10" YOffset="11"/>
    ...
    <Bit Index="47" Lut="F" Slice="0" XOffset="15" YOffset="11"/>
  </Port>

  <Port Direction="output" Name="passthrough_south_out"
       Type="passthrough_south" Width="48">
    <Bit Index="0" Lut="G" Slice="3" XOffset="10" YOffset="0"/>
    <Bit Index="1" Lut="F" Slice="3" XOffset="10" YOffset="0"/>
    <Bit Index="2" Lut="G" Slice="2" XOffset="10" YOffset="0"/>
    ...
    <Bit Index="47" Lut="F" Slice="0" XOffset="15" YOffset="0"/>
  </Port>

</Module>
```

A.9 Perl Script to Generate Source File List

```
# prj.pl

#!/usr/bin/perl
@list = `/bin/ls -1`;
$prefix = "verilog work ../hdl/pr/";
foreach $entry (@list)
{
chomp($entry);
print "$prefix$entry
";}
```

A.10 Source File List Generated by Perl Script

```
// prm.prj

verilog work ../hdl/pr/AbsVal_pr.v
verilog work ../hdl/pr/AbsVal.v
```

A.11 Makefile That Copies Bus Macro Files

```
# bm.make

all:
cp ../bus_macros/busmacro_xc4v_t2b_sync_narrow.nmc synth/busmacros/
cp ../bus_macros/busmacro_xc4v_b2t_sync_narrow.nmc synth/busmacros/
```

A.12 Makefile That Automates Xilinx PR Flow

```
# pr.make

##########
# Partial Reconfig makefile
##########

# partial module implementations
PRNAME = prm

# static implementations
STATIC_NAME = system_static

# original module name
NAME = absval

# UCF files
CONSTRAINT_FILE_BB = ../data/$(NAME)_top_bb.ucf
CONSTRAINT_FILE = ../data/$(NAME)_top.ucf

# other variables
MAKEDIR = ../scripts
SYNTH_PROJ_DIR = ../scripts
HDL = ../hdl
PART = xc4vlx60-ff668-10

# Partial module type name (not instance name)
PR_MOD_NAME = $(NAME)_pr

# Netlist files
NETDIR = ../synth
TOPNET = $(NETDIR)/top/top.ngc
STATICNET = $(NETDIR)/static/$(NAME)_static.ngc
PRNET = $(NETDIR)/pr/$(NAME)_pr.ngc

# Netlist directories
TOP_NET_DIR = $(dir $(TOPNET))
STATIC_NET_DIR = $(dir $(STATICNET))
PRNET_DIR  = $(dir $(PRNET))
```

```
# ... not technically netlists:
BUSMACRODIR = $(NETDIR)/busmacros
CHIPSCOPEDIR = $(NETDIR)/chipscope

# Phys implementation build directories
STATIC_DIR = ../static
PR = ../pr

# Merge directories
PRMERGE = ../merges

# used routes file generated by static logic implementation,
# and required by PR module implementation at PAR step
USED_ROUTES = ../static/static.used
ARCS_EXCLUDE = ../scripts/arcs.exclude

################################
all: merge

modules: $(PR)/$(PRNAME).ncd

static: $(STATIC_DIR)/$(STATIC_NAME).ncd

synth: $(TOPNET) $(STATICNET) $(PRNET)

synth_top: $(TOPNET)

static_ngd: $(STATIC_DIR)/$(STATIC_NAME).ngd

static_map: $(STATIC_DIR)/$(STATIC_NAME)_map.ncd

################################
merge: merge_copy
$(MAKE) -C $(PRMERGE)

merge_copy: $(STATIC_DIR)/$(STATIC_NAME).ncd $(PR)/$(PRNAME).ncd
cp -f -p $(STATIC_DIR)/$(STATIC_NAME).ncd $(PRMERGE)
cp -f -p $(PR)/$(PRNAME).ncd $(PRMERGE)

################################
# Partial reconfig build: PR
```

```
$(PR)/$(PRNAME).ncd: $(PR)/$(PRNAME)_map.ncd $(CONSTRAINT_FILE) $(ARCS_EXCLUDE)
par -w -ol med -uc $(CONSTRAINT_FILE) $(PR)/$(PRNAME)_map.ncd $@

$(PR)/$(PRNAME)_map.ncd: $(PR)/$(PRNAME).ngd
map $(PR)/$(PRNAME).ngd -o $@

$(PR)/$(PRNAME).ngd: $(TOPNET) $(PRNET) $(CONSTRAINT_FILE)
ngdbuild -p $(PART) -uc $(CONSTRAINT_FILE) -dd $(PR) -modular module
        -active $(PR_MOD_NAME) $(TOPNET) -sd $(PRNET_DIR) -sd $(BUSMACRODIR) $@

################################
# Static (aka base) design build

# this PAR step produces the used-routes/arcs-exclude file needed later
$(ARCS_EXCLUDE): $(STATIC_DIR)/$(STATIC_NAME).ncd

$(STATIC_DIR)/$(STATIC_NAME).ncd: $(STATIC_DIR)/$(STATIC_NAME)_map.ncd
                                                        $(CONSTRAINT_FILE_BB)
rm -f -r $(ARCS_EXCLUDE)
par -w -ol med -uc $(CONSTRAINT_FILE_BB) $(STATIC_DIR)/$(STATIC_NAME)_map.ncd $@
cp -p $(USED_ROUTES) $(ARCS_EXCLUDE)

$(STATIC_DIR)/$(STATIC_NAME)_map.ncd: $(STATIC_DIR)/$(STATIC_NAME).ngd
map $(STATIC_DIR)/$(STATIC_NAME).ngd -o $@

$(STATIC_DIR)/$(STATIC_NAME).ngd: $(TOPNET) $(STATICNET) $(CONSTRAINT_FILE_BB)
ngdbuild -p $(PART) -uc $(CONSTRAINT_FILE_BB) -dd $(STATIC_DIR) -modular initial
        $(TOPNET) -sd $(STATIC_NET_DIR) -sd $(BUSMACRODIR) -sd $(CHIPSCOPEDIR) $@

################################
# Synthesis

##############
# Top netlist

$(TOPNET): $(HDL)/top/$(NAME)_top.v $(SYNTH_PROJ_DIR)/top.xst
xst -ifn $(SYNTH_PROJ_DIR)/top.xst -ofn $(TOP_NET_DIR)/top.srp -intstyle silent

###############
# Static submodule netlists

$(STATICNET): $(HDL)/static/$(NAME)_static.v $(SYNTH_PROJ_DIR)/static.xst
```

```
xst -ifn $(SYNTH_PROJ_DIR)/static.xst -ofn $(STATIC_NET_DIR)/$(NAME)_static.srp
        -intstyle silent

###############
# Partial netlists

$(PRNET): $(HDL)/pr/$(NAME)_pr.v $(SYNTH_PROJ_DIR)/prm.xst
xst -ifn $(SYNTH_PROJ_DIR)/prm.xst -ofn $(PRNET_DIR)/$(NAME)_pr.srp -intstyle silent

###############################
clean: clean_phy clean_merges clean_synth

clean_phy:
rm -f -r $(STATIC_DIR)/*
rm -f -r $(PR)/*
rm -f -r $(ARCS_EXCLUDE)

clean_merges:
mv -f $(PRMERGE)/makefile ../
rm -f -r $(PRMERGE)/*
mv -f ../makefile $(PRMERGE)/

clean_synth:
rm -f -r $(TOP_NET_DIR)/*
rm -f -r $(STATIC_NET_DIR)/*
rm -f -r $(PRNET_DIR)/*
```

A.13 XST Scripts

XST script for modules located in the partial region: `prm.xst`

```
set -xsthdpdir ../synth/pr/
run
-ifn prm.prj
-ifmt mixed
-ofn ../synth/pr/absval_pr.ngc
-ofmt NGC
-p xc4vlx60-ff668-10
-opt_mode Speed
-opt_level 1
-keep_hierarchy YES
-iob false
-iobuf no
-top absval_pr
```

XST script for static logic modules: `static.xst`

```
set -xsthdpdir ../synth/static/
run
-ifn ../hdl/static/absval_static.v
-ifmt Verilog
-ofn ../synth/static/static.ngc
-ofmt NGC
-p xc4vlx60-ff668-10
-opt_mode Speed
-opt_level 1
-keep_hierarchy YES
-iob false
-iobuf no
```

XST script for top level module: `top.xst`

```
set -xsthdpdir ../synth/top/
run
-ifn ../hdl/top/absval_top.v
```

```
-ifmt Verilog
-ofn ../synth/top/top.ngc
-ofmt NGC
-p xc4vlx60-ff668-10
-opt_mode Speed
-opt_level 1
-keep_hierarchy YES
-iob false
-iobuf yes
-top absval_top
```

A.14 Makefile That Automates Xilinx PR Flow Merge Step

```
# makefile

##############################
# Partial Reconfig makefile (merge step)

# partial module implementations
PRNAME = prm

# static implementations
STATIC_NAME = system_static


# These file should get copied in by main makefile
STATIC_NCD = $(STATIC_NAME).ncd
PR_NCD = $(PRNAME).ncd

all: $(STATIC_NAME)_full.bit

# Assemble step
$(STATIC_NAME)_full.bit: $(STATIC_NAME).summary
PR_assemble $(STATIC_NCD) $(PR_NCD)

# Verify step
$(STATIC_NAME).summary: $(STATIC_NCD) $(PR_NCD)
PR_verifydesign $(STATIC_NCD) $(PR_NCD)
```