Framework for Hardware Agility on FPGAs

Prabhaav Bhardwaj

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfilment of the requirements for the degree of

Master of Science

in

Computer Engineering

Peter M. Athanas, Chair

Patrick R. Schaumont

Paul E. Plassmann

December 15, 2010

Blacksburg, Virginia

Keywords: Reconfigurable Computing, FPGA, Dynamic Routing, Virtex 5

Framework for Hardware Agility on FPGAs

Prabhaav Bhardwaj

(ABSTRACT)

As hardware applications become increasingly complex, the supporting technology needs to evolve and adapt to the demands. Field Programmable Gate Array (FPGA), Application Specific Integrated Circuit, General Purpose Processor, and System on Chip are the preferred devices for solving computational problems. Each of these platforms has its own specific advantages and disadvantages, which need to be accounted for during application development. Flexible radio communications has been dominated by Software Defined Radios. However, research in industry and universities has successfully developed run-time reconfiguration tools to make FPGA designs more flexible and thus vastly reducing configuration times. Developers now have a more powerful platform with dense Digital Signal Processor resources and the flexibility of SDR. Xilinx offers tools such as partial reconfiguration, which is a special modification of the standard tool-flow that supports configuration of the selected partial regions on an FPGA. The AgileHW project improves on the Xilinx tools resource allocation and routing scheme to increase the design agility and productivity. This thesis advances the AgileHW reconfigurable platform so developers can use the newer technology to build enhanced designs.

This work received support from the DARPA and the Harris Corporation.

Acknowledgements

Thanks to Dr. Peter Athanas, my advisor, for giving me the opportunity to work in the CCM lab and providing me with the guidance needed for achieving my goals.

Thanks to Dr. Patrick Schaumont for conducting one of the most enriching classes I have taken at Virginia Tech, and for serving on my committee.

Thanks to Dr. Paul Plassmann for all the entertaining conversations and guidance.

Thanks to Adolfo Recio, Tony Frangieh, Ali Sohangpurwala, Jacob Couch, and all my friends at CCM Lab for sharing your knowledge and making my time at the lab memorable.

Thanks to Katherine Hunter for being such a great companion during my time in Blacksburg.

Thanks to my family for being so supportive throughout my academic career. Without my family I would not be where I am today.

Contents

List of Figures

List of Tables

Acronyms

ASIC	Application Specific Integrated Circuit
BPSK	Binary Phase Shift Keying
BRAM	Block RAM
CAD	Control Address Data
CLB	Configurable Logic Block
DCM	Digital Clock Manager
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FIFO	First In First Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
HDL	Hardware Description Language
ICAP	Internal Configuration Access Port

IEEE	Institute of Electrical and Electronics Engineers
LUT	Look Up Table
MPGA	Mask Programmable Gate Array
NCD	Native Circuit Description
PAR	Place and Route
PIP	Programmable Interconnect Point
PR	Partial Reconfiguration
RF	Radio Frequency
SDR	Software Defined Radio
SOC	System On Chip
WoD	Wires on Demand
XML	eXtensible Markup Language
XDL	Xilinx Device Language

Chapter 1

Introduction

1.1 Motivation

Small, flexible, and efficient has become the mantra of computer industry. Cutting edge technologies such as smart phones, portable computers, and other electronic devices require fast and agile processors. These processors need to be able to perform various tasks such as signal processing, image processing, cryptography, and data manipulation to name a few. Some of these tasks are performed in the software domain and others are better suited for the hardware domain. When dealing with the hardware to run these tasks, the developer has to choose the platform that best allows the application to perform at its maximum potential.

Choosing a hardware platform that is cost effective, power efficient and flexible is crucial to the design process. The common hardware platforms are System on Chip (SoC), Application

Specific Integrated Circuit (ASIC), Field Programmable Gate Array (FPGA), and General Purpose Processor (GPP). Each platform offer unique advantages and disadvantages. FPGAs have dense signal processing resources such as Digital Signal Processor (DSP) and memory resources. It is a homogeneous hardware development platform that can be used for many different applications such as digital signal processing, cryptography, hardware emulation and imaging. FPGAs are relatively power efficient and minute which makes them desirable for embedded systems. However, one major setback for FPGAs is lengthy implementation times. This issue is especially pertinent when large designs need to be recompiled during development.

An FPGA is programmed using a bitstream. The bitstream contains the information for each tile on the FPGA such as the Configurable Logic Block (CLB), DSP, Block RAM (BRAM), etc. Every time the developer changes the Hardware Description Language (HDL) for the design, a new bitstream reflecting the changes is created and loaded on the FPGA. Even if certain logic is unchanged, the whole design is recompiled every time. Hardware design lacks the intermediate stepping that is allowed in software for debugging and flexible implementation. However, new research concepts such as run-time reconfiguration [5] and partial reconfiguration [4] are bridging the gap between hardware and software design productivity.

Traditional hardware development involved programming the whole device when any logic changes appeared in the design. Instead, if the developer could re-program only the part of the design that changed separately, configuration time would be drastically reduced. There are two methods for this process, module based and difference based [4]. Radio designs

commonly use a modular approach. A radio developer can divide the different functions of the radio into separate modules and program them as necessary. In the difference based approach, the developer can use FPGA Editor, a Xilinx tool, to change certain logic. The tools identify the differences between the two bitstreams and accordingly update the base bitstream. Both of these methods allow the developer to divide the overall design process into incremental steps, which greatly reduces the development time.

Another advantage of run-time reconfiguration is the ability to swap the modules at run-time. Traditional FPGA implementation required the entire device to be re-programmed when a new functionality is added, which leads to lengthy down times. However, with run-time reconfiguration, the developer only re-programs certain sections of the device. The remainder of the design will be working during the module swap. This is similar to the software domain, where users can add and remove pre-compiled parts of the program during run-time.

FPGA productivity is greatly enhanced with these new features. Developers will be able to develop and implement designs incrementally. Portions of the FPGA can be reconfigured during run-time, which was only available in the software domain. Specialized tools developed by industry such as Xilinx Partial Reconfiguration (PR) and academic tools such as AgileHW [5]; allow the user to use FPGAs for applications previously restricted to software. One exciting new application for FPGAs is dynamic radios [6] similar to Software Defined Radios (SDR). These radios exploit the superior signal processing capabilities of the FPGA and the flexibility offered by run-time reconfiguration.

Tools such as Xilinx PR allow users to allocate reconfigurable regions where modules can be placed. However, one of the major disadvantage of this tool is that only one module can be placed in this region. If in the future the module size increases or decreases, the pre-allocated resources also need to be changed. In the AgileHW project [5], [7], there is one large dynamic region which has multiple modules that are routed and placed at run-time. The size of the modules is assigned by the user which allows resources to be managed properly.

As hardware platforms continue to advance, the tools used to program them need to be upgraded. With the advent of newer FPGA architectures such as Xilinx Virtex 5 and Virtex 6, the reconfiguration tools to program these FPGAs also need to be upgraded. Companies like Xilinx and Altera have started to support PR design and offer better tools to manage designs. The AgileHW was developed for Xilinx Virtex 2, Virtex 2 Pro and Virtex 4 boards. This thesis focuses on adapting and upgrading the AgileHW flow to account for the completely revamped Xilinx Virtex 5 FPGA architecture.

1.2 Objective

AgileHW is a re-configuration tool that helps the developer design and implement dynamic designs on FPGAs. It uses the Xilinx PR flow to create partial bitstreams for each module. These modules are then loaded in the FPGA through the Internal Configuration Access Port (ICAP). Many different radio modules designs were implemented on the Virtex 4 FPGA using this flow. However, with new FPGA platforms such as Virtex 5, there is a push to move to

newer platforms. The goal of this work is to update the AgileHW framework for Virtex 5 FPGAs and to provide other options to re-program the FPGA.

There are two important changes that need to be made to the framework. First, information about the revamped Virtex 5 family architecture needs to be added. Next, the run-time router used to connect the partial modules in the dynamic region needs to be updated to account for the interconnect changes. These infrastructure to AgileHW will allow future developers to design applications for the Virtex 5 FPGA family.

The move to Virtex 5 and to later generations is necessary for increasing productivity and efficiency of designs. Virtex 5 FPGAs have a revised Look Up Table (LUT) structure, faster DSPs, denser BRAMs and most importantly a far more robust interconnect [8]. Developers will be able to implement far more complex designs such as faster radios or stronger cryptographic algorithms. Another change to the framework is to gain independence from the Xilinx PR tools. In this generation of AgileHW, the developer will have the option to load partial bitstreams through the ICAP or stitch full bitstreams together which will not require the Xilinx PR toolkit. With this new upgraded infrastructure, the developer will have the same agility seen in previous generations of the tool in addition to some new flexibility.

Summary of Contributions This thesis aims to contribute the following:

- Extend the existing AgileHW framework to include Virtex 5
- Upgrade the run-time router to use the new Virtex 5 interconnect
- Develop new bitstream manipulation tools to use with future framework

- Incorporate existing AgileHW flow for Virtex 5

1.3 Organization

The thesis is organized in the following way. Chapter 2 provides the background research conducted in the FPGA reconfiguration domain. Chapter 3 explains the AgileHW project contributions and previous related work. Chapter 4 provides an in-depth analysis of the architectural differences that were made in the reconfiguration framework for Virtex-5 FPGA. Chapter 5 describes the modifications to the run-time router and the new Virtex 5 interconnect database. Chapter 6 explains the implementation results. Chapter 7 summarizes the contributions of the thesis and ideas for future work.

Chapter 2

Background

FPGAs are a popular hardware platform because they offer design versatility and powerful resources. They are pre-dominantly used in signal processing application platforms such as set-top boxes and radios. However, development issues such as long compilation and implementation time have held back FPGAs. Software is commonly substituted to perform these applications at the price of latency and throughput because of its faster development cycle.

2.1 Traditional FPGA Programming

FPGAs were initially developed as a hardware platform that could be customized quickly to perform various applications. They were not as fast or dense as Mask Programmable Gate Arrays (MPGA), but had significantly lower set up time and cost. With the advancements

of transistor technology, FPGAs have become much more than simple glue logic systems. While the architecture advanced, bigger issues arose such as inept logic mapping and routing software tools [9].

2.2 Shift to Dynamic FPGA Programming

As FPGAs increased in size and complexity, development time also increased. To address this issue, the idea of dynamic FPGA reconfiguration was developed as a means to allow the FPGA adapt to the changing demands of the application. There are different abstraction levels for reconfiguration; system level, functional level, and RTL level [10] [11].

Function level abstraction is the most popular method used because it offers the optimal combination of granularity and logic reconfiguration size. Flexible reconfigurable FPGA design offers important advantages such as power and cost optimization in designs by only using functions when necessary and removing them otherwise [12]. After the initial learning curve of PR, implementation and development times are significantly reduced because only parts of the design is changed at a time.

2.3 Dynamic Framework

Improving FPGA agility has been a burgeoning topic in research due to the platform's immense potential. There are two areas of research for dynamic FPGA development, framework

and communication architecture. The framework area involves establishing the information necessary to add and remove module logic to the existing design. And the communication area involves routing the new logic changes so data can pass through.

The framework for reconfigurable FPGA design has a common standard, dynamic and static region. Xilinx offers two different methods of programming the dynamic region, difference based and module based. The difference based flow allows for only minor changes in the bitstream to be made to the design quickly. However, module based designs have partial bitstreams that are added to the full bitstream [4].

Although this paradigm offers the designer flexibility to add and remove modules to the design, it restricts the developer to only one module per dynamic region. This issue arises because partial reconfiguration is a special capability of the current tools, and thus not well supported. When the developer assigns a region as a dynamic region, the tools cannot place routes in this area. Thus, if the developer wants to place multiple modules in one dynamic region, the tools are not capable of routing the datapath for the modules. Thus if the module size changes, the allocated resources also needs to be revamped.

In [13], the author states the common FPGA use cases such as System on Chip, co-processor, and stand-alone. Each of these use cases utilize the dynamic region in a different manner and thus it is partitioned accordingly. Although these best use scenarios are helpful, they still do not address the issue of static dynamic region size.

There are other simpler approaches offered to curb this issue such as having pass-through

LUTs at the boundaries of the modules shown in Figure 2.1. In [1] the author uses this method to place multiple modules in the same dynamic region. Although this method circumvents the issue presented earlier, the user needs to always be aware of placement and does not offer true flexibility.

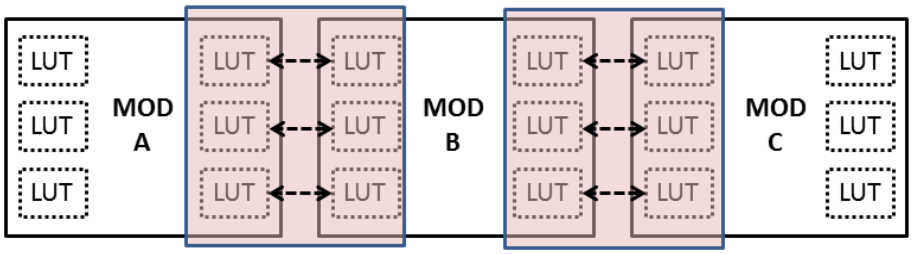

Figure 2.1: LUT pass through method from [1].

A truly dynamic system needs to have control over all the aspects of the module implementation. The AgileHW framework is a derivative of Wires on Demand (WoD) [5]. In WoD, there is a single dynamic region which spans the left half of the FPGA due to the location of the DSPs. In this region, partial modules are placed and routed during run-time. This allows the developer to change module size and placement without having to worry about resource allocation or wastage.

2.4 Reconfigurable Communications

The communication interface for the modules placed in the dynamic region is an important research issue [2], [14], [3]. In the traditional setup of one module per dynamic region, data is passed using bus-macros placed at the boundaries. However, routing inside the dynamic

region is a completely new problem. There are different approaches to solve this issue such as a bus communication scheme, network on chip, and run-time routers.

The bus communication architecture is the simplest reconfigurable communication method to implement and works well for systems with two modules. However, scalability on bus architectures is poor and thus limits the design [14]. A variation of the bus architecture is a split bus architecture that allots each module its own bus. The communication is then channeled through a unified bus macro, which is handled by an external arbiter as shown in Figure 2.2 [2].

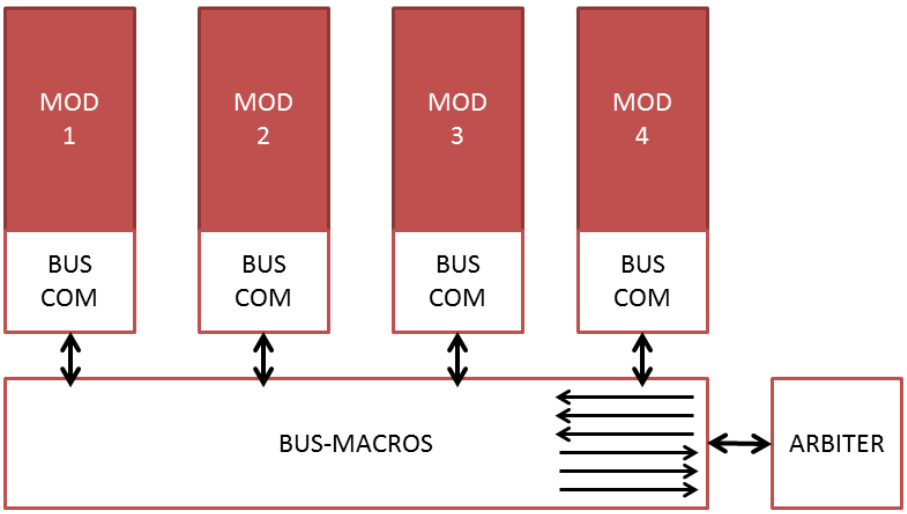

Figure 2.2: Split bus architecture from [2].

A more complicated approach to connecting modules is utilizing a dynamic network on chip. There are router elements placed in a 2-D grid pattern in the dynamic region that are used as module interconnect shown in Figure 2.3. When a module is placed in the dynamic region, it can easily be mapped to the routers. Furthermore, unused router elements are later used

for other purposes such as pass through [3].

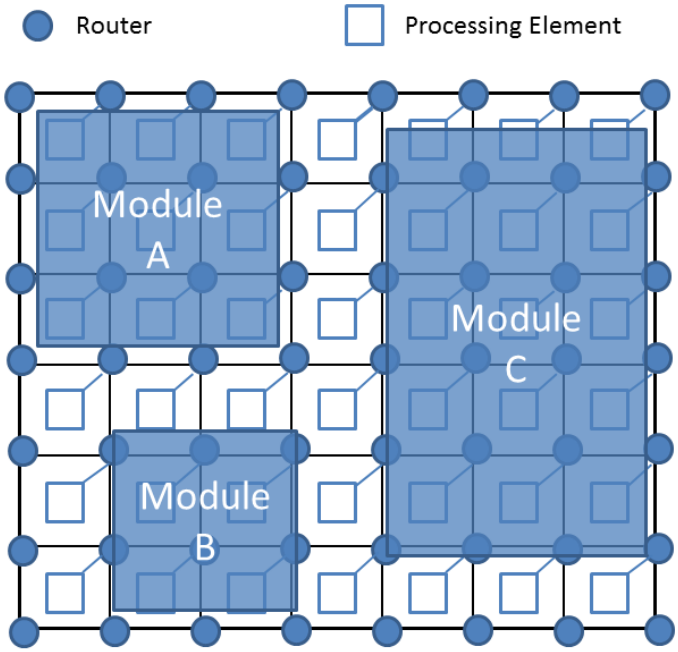

Figure 2.3: Dynamic Network on Chip from [3].

On-The-Fly routing is another methodology for dynamic communications. Unlike previous methods which used pass throughs, bus architectures and network on chip, on-the-fly routing uses the dense interconnect fabric of the FPGA to make simple routes during run-time. At compile time, the module has designated inputs and outputs, assigned by the developer, mapped to bus-macros. When the modules are placed at run-time, the router connects the corresponding inputs and outputs of the adjacent modules [7].

2.5 Applications

Truly dynamic reconfigurable applications are revolutionizing the FPGA industry. There are numerous applications for dynamic FPGA designs such as image processing, systems on chip applications, and flexible radios. The AgileHW project used the WoD framework in combination with the run-time router to build high speed and low speed radios for multiple data manipulation applications [6].

Chapter 3

AgileHW

AgileHW is a form of PR that allows developers design applications such as dynamic radios. PR allows select regions of an FPGA to be modified without disturbing the rest of the design. However, AgileHW allows the user to place multiple modules in one dynamic region. Similar to SDR, which offers flexibility by using software that can be quickly modified, a PR system offers additional flexibility through the ability to quickly swap out modules. A module in our case is a set of similar data processing steps or techniques that fit well into a singular unit of computation. Our solution will include a large dynamic region that supports the placement of modules during run-time and an overlaying static system that supports the interaction with the dynamic region and the external host machine.

3.1 Compile-time

Compile-time is the first step in the Agile Hardware flow that happens before run-time. It is responsible for establishing communication between the static and dynamic region, and creating partial modules. The FPGA is divided into two halves with the left half dedicated for PR, and the right half dedicated for static logic. The static region, which includes the clocking, I/O and other supporting modules, is built with the idea of PR in mind and has support from the tools. The other responsibility of compile-time is to build the partial modules. These modules act as the building blocks of the radio placed in the dynamic region. Xilinx provides a basic environment that supports PR, but some custom tools have been designed along the way to help in designing our unique system.

The Xilinx Partial Reconfiguration Early Access Software Tools for ISE 9.2i (here on referred to as the Xilinx PR Toolkit) has been a useful contribution to the reconfigurable computing community, and has served as the basis for many research projects. This tool kit, while not formally released with the ISE tool kit, was intended to augment the modular design flow. The underlying principle is that the tool kit enables slot-based partial reconfiguration. A "slot" in this case is a fixed geometric region – fixed in size and fixed in position for a selected device. Multiple partial designs can be created with this tool kit that can one-by-one populate a given slot.

For the domain of SDR, it is easy to show why a slot-based partial reconfiguration model is not desired. The slot locations and sizes are allocated at design time, which make them a

compile-time constant. This means if there is a need to add an additional module, an FFT for instance, there may not be an available slot based on the current design. The size of the slots are also fixed, which can lead to wasted space if the modules allocated to the region exhibit a high degree of footprint disparity. A collection of filters designed for placement in a given slot where one filter is substantially larger than any other will make the targeted slot an inefficient use of FPGA fabric the majority of the time. Finally, the I/O ports on these regions are static, which force unnecessary constraints on the modules desired for a particular region while also hurting flexibility by disallowing a module to relocate to different slots.

The solution is to deviate from the standard Xilinx PR flow, and instead create a large reconfigurable region to house multiple modules where pre-fabricated modules are relocated as needed in the region. In addition, the interconnectivity for these newly placed modules is performed in a fraction of a second outside of the ISE tool suite, eliminating the costly Place and Route (PAR) process. In the context of the work done in this project, there are two regions where both the transmit and receive chains have dedicated regions for incoming or outgoing radio processing. Each region will need the ability to dynamically remove specific radio modules to maintain the productivity presented in the dynamic radio. The Wires-on-Demand (WOD) run-time environment allows for on-the-fly modifications to the reconfigurable regions giving similar granularity to that of software. Each of the modules contain a wrapper structure that provides anchor points for routing. Modules are relocated and placed based on an algorithm aimed at efficient use of the reconfigurable region and

reducing routing delays between modules. The router used for module connections is both lightweight in terms of memory usage and fast in execution. The set of tools presented will allow the design to break away from the vendor tools giving significant gains in productivity.

3.1.1 Current Static design

The static region of the FPGA has two sections; static logic and an empty sandbox where the dynamic modules will be placed. The static logic of the FPGA has the clock management functions, data I/O modules, ICAP, and bus macros.

Clock Management : The modules in this section use the FPGA clock or external clock and modulate it with Digital Clock Manager (DCM) to generate the desired frequency needed for the radios.

Data I/O : Modules such as the Control Address and Data Bus (CAD Bus), used for local communications; First In First Out (FIFO), to process data; and other functions that are necessary to properly create the data packets that are transmitted and received by the radios.

ICAP : The ICAP is a specialized data I/O port designed for loading partial bitstreams during run-time. Its module logic is placed in the static region and is clocked separately with a faster ICAP clock generated using a Digital Clock Manager (DCM). It has reset, read, and write functionality which are called when a partial bitstream is loaded.

Bus Macros : These Xilinx cores act as the connection between the dynamic and static regions of the design. Bus macros ensure that signals crossing the PR region are consistent with both the static and dynamic regions.

3.1.2 Dynamic Region

The dynamic region is the length of the FPGA and is placed on the left hand side where the DSPs are located. It is an empty region without any pre-existing routes. Partial modules are placed and routed at run-time to create the dynamic applications. The following sub-sections explain the placement and creation of partial modules.

Placement

In this project, the reconfigurable region was placed on the left half of the Virtex 4 FPGA because the DSP resources are only available here. The location of these regions is determined based on the layout of the special resources in a given FPGA (DSPs, BRAMs, etc.). It is not necessary to limit the user to a specific Xilinx FPGA within a family, yet all have slightly different resource layouts. The resolution is to automate this process with the use of PlanAhead, which has knowledge of many FPGA layouts even throughout different families. The layouts are stored in eXtensible Markup Language (XML) files, which can be parsed for information relative to the task of floor planning. This automation ensures the user will not be required to have in-depth knowledge of the FPGA of their choosing. This tool will only

be used for determining the placement of the large reconfiguration regions and does not play a roll in the run-time aspects of library component placement and routing.

Partial Module Creation

The partial modules make up the parts of the radio transmitter and receiver. These modules are placed in the dynamic region and connected on-the-fly to create the radios. The tool flow to build the partial modules is depicted in Figure 3.1.

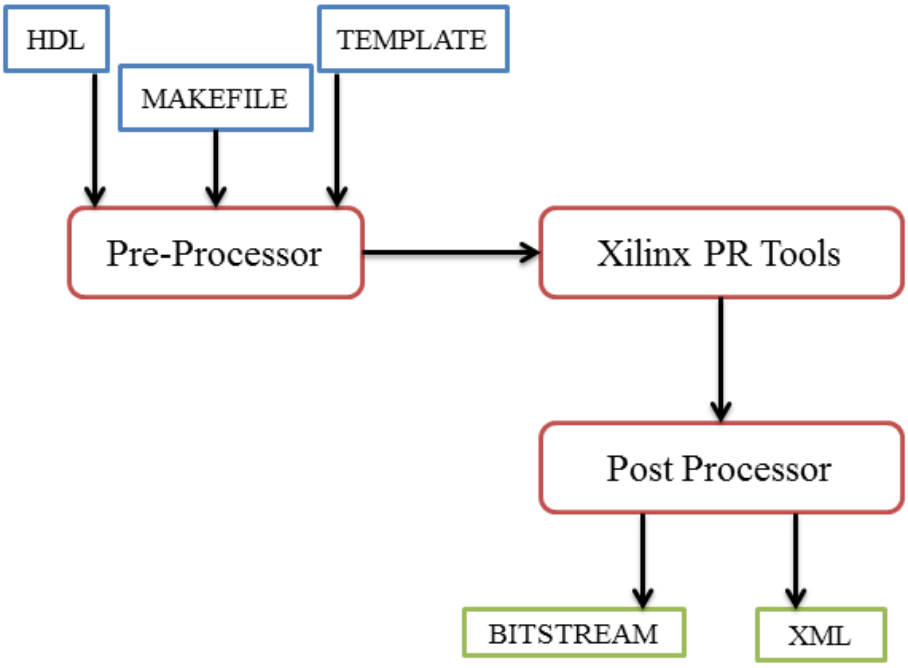

Figure 3.1: Partial Module Tool Flow.

There is a set structure to build the partial modules in the AgileHW flow. The partial module tools accept two different types of files, HDL files that describe the module's logic and a template file that defines the module's size and port information. The HDL files

consist user defined logic for radio specific functions like filtering, interpolation, scrambling, de-scrambling, and any other functionality deemed necessary. The template file defines the modules port information such as the type, size, direction, and name. The template file also defines the module's size so it can have adequate resources to incorporate all the logic.

Once the HDL and template files are prepared, it is passed to the pre-processor with a Makefile. This pre-processor ensures the template file and module names correspond, parses the template file to set up the module boundaries and resources. This is then passed to the Xilinx PR tools which compile the HDL files and place and route the design on the FPGA. Finally, the post-processor receives the compiled HDL and template file information and preps the module and places it in the assigned co-ordinates of the FPGA. It also generates an XML file that is used by the run-time tools to understand the module size and port information. At the end of the tool flow, the user receives a partial bitstream and an XML file with the module port definitions used by the run-time tools.

3.1.3 Inter-module Communication

After the partial modules are built, the run-time flow connects them in the dynamic region. The current data flow in Agile Hardware is north and south, meaning that data flows vertically across the stacked modules. The run-time tools use the XML files created by the partial module tool flow to connect the modules to each other. The first module in the sandbox receives its inputs from the sandbox. When the next module is placed in the chain, the

outputs of the first module are connected to the input of the second module. The output of the final module is then connected to the output of the sandbox. The tools identify which ports are connected by the port type: data, reset, ready. So a output port of type data is connected to and input port of type data. The clock is not passed through the modules because the whole sandbox is clocked by the static logic.

3.2 Run-time

The previous section discussed how the partial modules were created, and briefly introduced on-the-fly routing. When a partial module is placed in the dynamic region, it needs to be stitched with the static bitstream so the static and dynamic logic work together. The routers job is split in two major functions, data flow control and resource allocation.

3.2.1 Resource Allocation

When the partial module is designed and built, it has specific resource requirements such as DSPs, BRAMs and CLBs. The router ensures that when the partial module is placed in its assigned location, all the links with the appropriate FPGA sites are connected properly.

3.2.2 Place and Route

After the partial module is placed in the dynamic region, the router connects the data input nets with the outputs of the module above and the data output nets with the sandbox output. It also routes the clock line used by the sandbox to the logic, so all the modules are operating at the correct frequency.

The channel routing scheme used by the router to connect the different nets and resources is very similar to the Left-Edge Algorithm [15]. A depiction of a dynamic radio transmitter on Virtex-4 is shown in Figure 3.2.

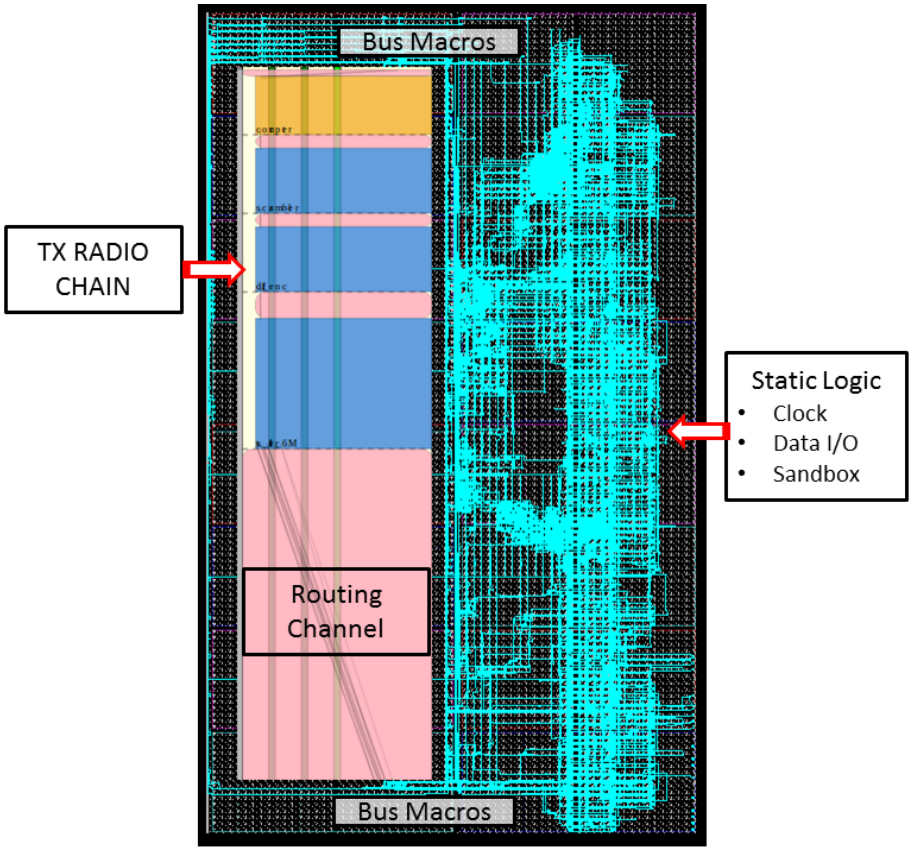

Figure 3.2: AgileHW Radio Transmitter Build

3.3 Agile Framework

The agile framework is the penultimate piece of the Agile Hardware flow. It is responsible for the on-the-fly bitstream manipulation that makes the design dynamic. The framework is responsible for placing and removing configuration data in the system, adding and removing new routing in the bitstream, and modifying designated LUTs and Programmable Interconnect Points (PIP). The current version of agile framework is capable of performing these functions on Virtex 2, Virtex 2 Pro, and Virtex 4 FPGAs. Information about Virtex 5 FPGAs added to the agile framework database to make it available across multiple platforms is explained in the next chapter. A breakdown of the framework tool flow is described in the following subsections.

3.3.1 Device & Architecture Query

The first task at hand is to identify which device the framework is modifying. The device information is used to identify the architectural changes in the device, the locations of said sites, and other information that are specific to each device.

3.3.2 Base Tasks

Once the device being manipulated is determined, the agile framework parses the base bitstream that consists of the static region. It also checks the PIP and LUT database for the site information and functions.

3.3.3 Place and Remove Configuration Data

After the information for the device has been acquired, the specified modules can be added and removed. The tools 'OR' the partial and base bitstream to add the module information to the system. There are rules that are in place to check that the module is within bounds and the resources are available. When the tools need to remove a placed module in the bitstream, they use a mask operation to write over the designated area.

3.3.4 Apply Modifications

When the partial module is added to the base bitstream, there are several PIPs and LUTs that change in the module's designated location. Once the sites are modified, new routes to the base bitstream need to be established. A list of all the modified sites is created and passed to the router.

3.3.5 Add and Remove Routes

The router receives the modified LUT and PIP database information and uses a left-edge algorithm [15] to connect them in the bitstream. The tools iterate over each of the sink and source to tie the corresponding LUTs and PIPs. When the tools need to remove the routes, they iterate over the corresponding LUTs and PIPs and simply turn the sites off.

Chapter 4

Virtex-5 AgileHW Infrastructure

When the AgileHW project was expanded to Virtex-5 FPGA family, specific information about the device architecture and interconnect specifications needed to be added to the framework database. This information provides the AgileHW framework data necessary to place and remove partial modules, modify the base bitstream, and add or remove routes. The tool flow for AgileHW is seen in Figure 4.1.

In the first step, the user loads the static bitstream into the framework. This bitstream has the information necessary to program the data I/O, clock, and ICAP logic for the design. The AgileHW framework extracts the device information from the bitstream header and loads the appropriate information necessary to manipulate the FPGA at run-time.

Figure 4.1: AgileHW Tool Flow.

4.1 Device Information

The device information class has all the specific information about the FPGAs physical layout. Since there was a complete overhaul for the Virtex-5 architecture, new information needed to be added to the database. When the base bitstream is loaded into AgileHW, it determines the specific FPGA family, platform and part. Using the FPGA's device specific information stored in the database, AgileHW is able to make changes to the design when partial bitstreams are added to the flow. The different levels of abstraction and the information added are explained here.

Family Information : this step determines family specific information such as specific site information common to all the FPGAs in the family. For example, Virtex-5 has 6-input LUTs and BRAM36 [16]; conversely, Virtex-4 has 4-input LUTs and BRAM16 [17].

Platform Information : once the correct family is chosen, the differences of the different platform need to be addressed. There are different platforms of FPGAs within a family for different applications. For example, an application group might need high performance logic applications with advanced serial connectivity so the developer might

choose a FPGA from the Virtex-5 LXT platform. Or if the application needs vast DSP resources for signal processing intensive applications, then the developer might choose a FPGA from the Virtex-5 SXT platform. Thus, different platforms have different amounts of BRAMs, DSPs and CLBs in each clock region.

Part Information : the final step of abstraction for the device information is the specific part of the platform. In each platform, there are different size FPGAs that correlate with the numbering system. Smaller FPGAs have fewer columns and rows of specific sites. Thus the database needs to store the part specific information such as number of clock regions in the device and number of columns of CLBs, BRAMs, and DSPs in each of these regions.

Device coordinates : there are different co-ordinate systems used in AgileHW. These co-ordinate systems are vital for calculating the correct site location. There are three different systems used in the framework, UCLB and XCLB. The XCLB co-ordinate system does not discriminate between the sites. It assigns the co-ordinates according to the sites location in the FPGA. On the other hand, the UCLB co-ordinates is used to easily determine the position of the CLB visually. It only accounts for the CLB interconnects and skips all other sites.

4.2 Bitstream Stitching Tools

AgileHW is different from the Xilinx PR flow because it allows the developer to place and remove multiple modules in one large dynamic region. This process is achieved by stitching partial bitstreams into the full bitstream or masking the region and using a run-time router to connect the modules. Bitstream manipulation tools are used to manipulate the individual bits of a bitstream specific to the configuration area. Each FPGA family is configured differently because of the specific site placement. The Virtex-5 FPGA bitstream information is calculated from the device information provided earlier. The bitstream is composed of three parts: header information, frame words, and end packets [18] as seen in Figure 4.2. This functionality is common in the Virtex family, and thus does not require any extra family specific information.

4.2.1 LUT Modification Tools

Each partial module has bus macros on inputs and outputs. The bus macros carry the data from the output of Module A to the input of Module B. They are also placed at the boundaries of the static and dynamic region for the same functionality. Each bus macro is a grouping of two CLBs placed on the boundary with each CLB in a different region like seen in Figure 4.3.

The AgileHW framework uses the LUTs in the bus macro slices as a pass through so that data can data can pass to the next module. The LUT equations need to be programmed

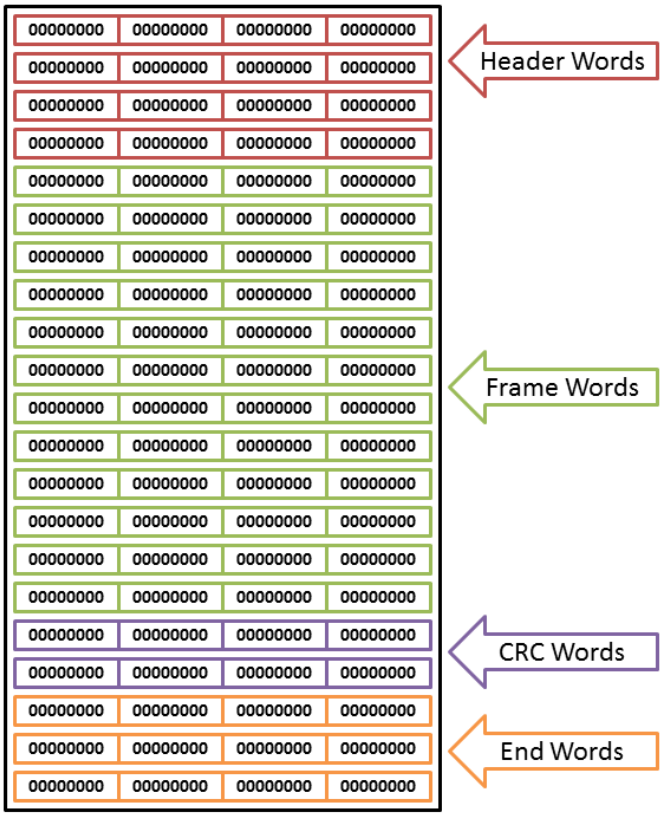

Figure 4.2: Virtex-5 Bitstream Composition

for this purpose. There are two different slice types in Virtex 5, SliceL and SliceM. The equations are specific to the LUT input and slice. The equations are acquired by configuring the LUTs using FPGA Editor and analyzing the changes in the bitstream.

4.2.2 PIP Database

The PIP database is inherited from the XDL report for the interconnect. It holds all the PIP connections available in the Virtex 5 switch matrix.

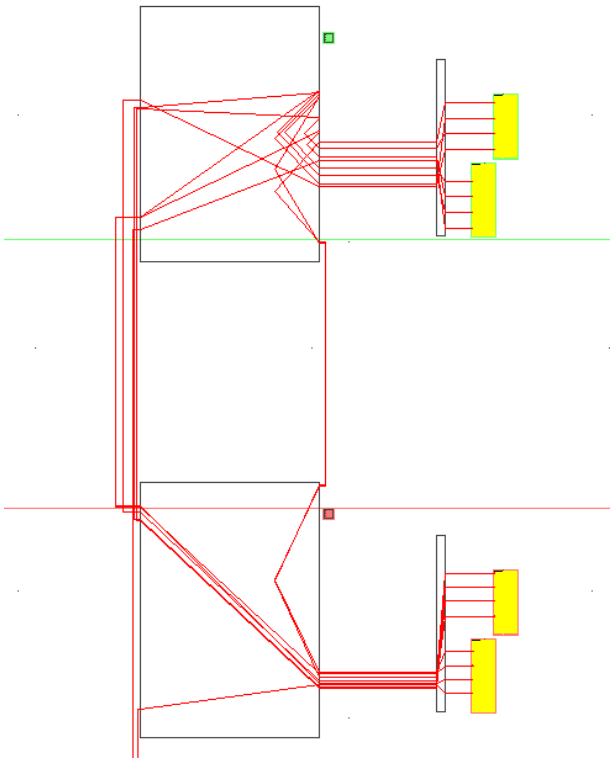

Figure 4.3: Virtex-5 Custom Bus Macro

4.3 Agile Router

As radio designs become increasingly complex, using an agile system for development and deployment is crucial. FPGAs have the ability to be configured to perform a certain task effectively, and traditionally FPGAs need to be re-programmed completely to perform different tasks. However, with the advent of Xilinx PR, run-time reconfiguration has made FPGA design flexible.

The core idea of AgileHW is to use FPGAs in a non-traditional fashion for increased agility and productivity. AgileHW is an environment well suited for the design of flexible modular

radios similar to SDR. An SDR system using software modulation provides productivity and agility at the sacrifice of computational complexity. AgileHW gives the designer the choice to offload signal processing tasks to hardware while retaining productivity and agility. One of the key design element of AgileHW is modularity of components which aids in ease of use and extensibility.

Once the modules are placed in the reconfigurable sandbox region, the run-time router connects the corresponding inputs and outputs. There are bus macros placed on the boundaries of the module used to make connections for the inter-module data flow. The upgraded router is still performing this task using a channel routing approach; however, since the resources such as the wires and slices have been upgraded, the new run-time router needs to be adapted to account for thee new features of the Virtex-5 architecture.

While developing the routing infrastructure for the Virtex-5 architecture, it is obvious that there are a multitude of changes in this generation of the Xilinx family. The interconnect fabric, which is an integral part of the router, has been completely revamped so the router can make connections with fewer hops. The CLB design has also been remodeled to allow for denser logic design and lower routing delays. These changes allow the routing to be far more efficient and flexible.

4.3.1 Routing Resources

There are two important resources used by the router, double wire segments and CLB slices. Double wires are the primary resource used for the module data flow shown in Figure 4.5. The CLB slices are used by the bus macro anchor points on the boundaries of the modules shown in Figure 4.3. Since, the two major changes in the Virtex 5 architecture involved these resources, there were a plethora of changes that needed to be made for the router.

Interconnect

Double wires are capable of connecting three interconnect tiles and travel east, west, north, south, north-east, north-west, south-east, south-west, east-north, east-south, west-north and west-south. In Virtex-4 and previous generations, double segments were restricted to four directions; east, west, north, and south and did not have the capability of moving diagonally. Thus, OMUX wires were used to make up for this capability. OMUX wires have limited connectivity, but they offer a quick way to connect directly to the eight neighboring tiles, which is extremely useful when changing directions or making turns [7]. Since Virtex-5 double wires already have the capability to move diagonally, OMUX is no longer necessary and has been eliminated from the interconnect architecture. The new interconnect uses a diagonally symmetric pattern which allows for better connectivity. With the improved interconnect, there are lower routing delays and efficient and uniform routing patterns as explained in Figure 4.4 where each box represents a CLB.

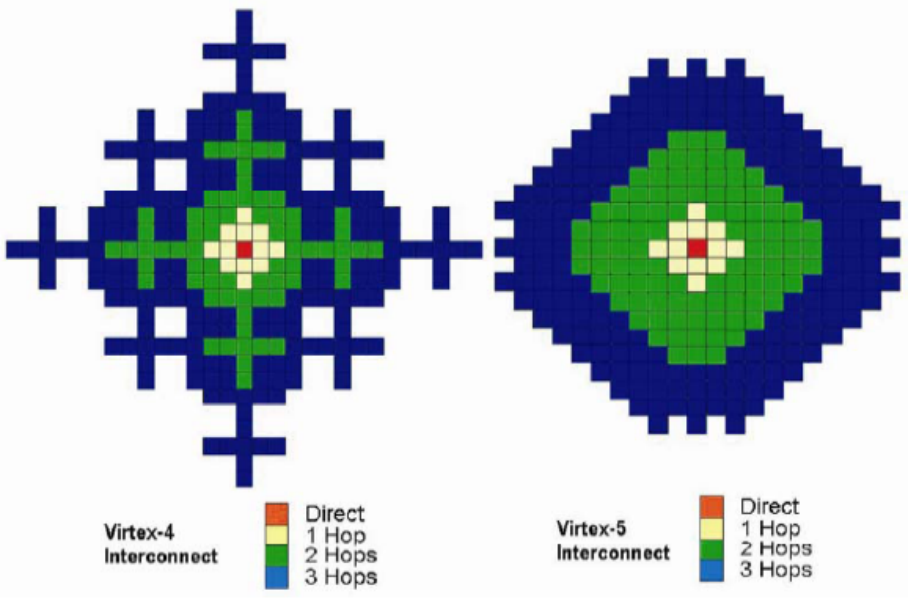

Figure 4.4: Virtex-4 vs. Virtex-5 Interconnect from [4].

CLB

Another key change in the V5 architecture is the new CLB architecture. In previous generations, a CLB consisted of four slices (0, 1, 2, 3) with two 4-input LUTs (F and G). However, now the new CLB has two slices (0, 1) and four 6-input LUTs (A, B, C and D). The CLBs are used for the bus macros placed on the module boundaries. The run-time router uses these bus macros to pass the data between modules. Thus with the upgraded LUTs the router has more inputs and LUTs to pass data through a single slice making the logic denser.

4.3.2 Routing Database

As discussed earlier, the run-time router is used to connect the modules placed in the sandbox region. It does this by connecting the source and sink bus macros of the modules to each other. The router uses a user generated wire database to make these connections. The database has the information described below for the double wire segments.

Connections : each wire is able to connect to other wires using the switch matrix. Since the routing database only uses a subset of the interconnect, unused wires such as *long* wires need to be pruned. The connections of each wire is stored in a data structure.

Special Segments : there are special double segments present in Virtex-5 that can be used alternatively if routes are sparse. Unlike the traditional double segments that travel to two neighboring CLBs in a designated direction, special segments have unique properties. For example, SR2BEG0SO, is a double segment that travels three CLBs south like seen in Figure 4.5.

CLB Extensions : the database is using only double wires to make connections between CLBs. Each double connects to the two neighboring CLBs, and this information is stored in the corresponding array. In diagonal segments, each extension travels in a different direction. Thus picking the mid or end extension influences the router greatly. Therefore, diagonal segments are connected to each extension separately.

Directions : Figure 4.5 shows the double wires available from each interconnect block. There are twelve directions possible for the wires - east, west, north, south, east-north,

east-south, west-north, west-south, northeast, northwest, southeast, southwest. This information is determined by the first two letters of the double name that are acronyms for the corresponding direction.

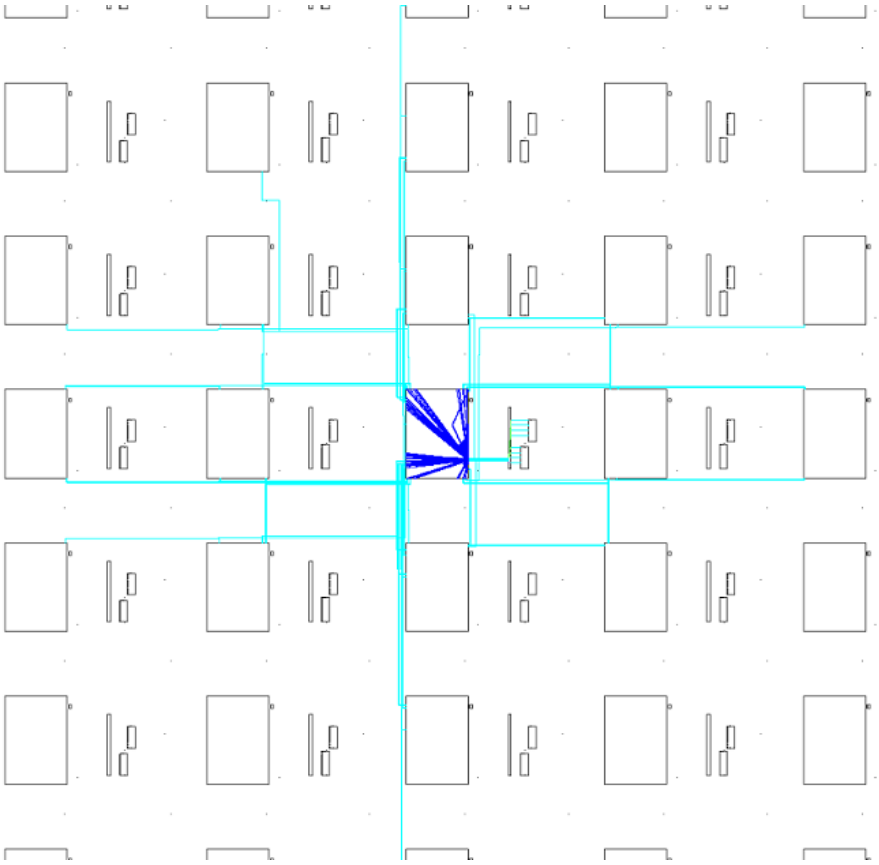

Figure 4.5: Virtex-5 Double Segment Interconnect.

Point : there are three points for a wire - beginning, middle, and end. This describes the three CLBs connected by the double wire. The beginning point corresponds to the originating CLB, the mid point is the second CLB, and finally the end point refers to the third CLB.

Index : each double segment has two other sister segments with the same properties that

can be used for other connections.

Type : the wire types fall in two categories, connecting type or I/O type. Double wires are the only connecting type wires used in the routing database. For the I/O, the current scheme allows the router to use four LUT (A, B, C, D) input and output lines. This can be further expanded to use the flip-flop I/O, but the current router does not need these additional options.

4.3.3 Routing Algorithm

Initially the existing channel routing algorithm was believed to be adequate for the Virtex-5 router; however, the new architecture forced some modifications to the algorithm. Previous Virtex architectures had dense interconnect in each of the major directions, and the router algorithm exploited this information. However, with the addition of diagonal segments in Virtex-5, the interconnect in the four major directions was reduced. The Virtex-4 routing algorithm and updated algorithm are shown in Figure 4.6.

Track Allocation : once a source and sink CLB is designated, the router determines the direction and tracks available to make the connections.

Start Segment : after a start segment is selected, the router references the routing database to determine the interconnect PIPs. Then it iterates over the connections to find the correct segment.

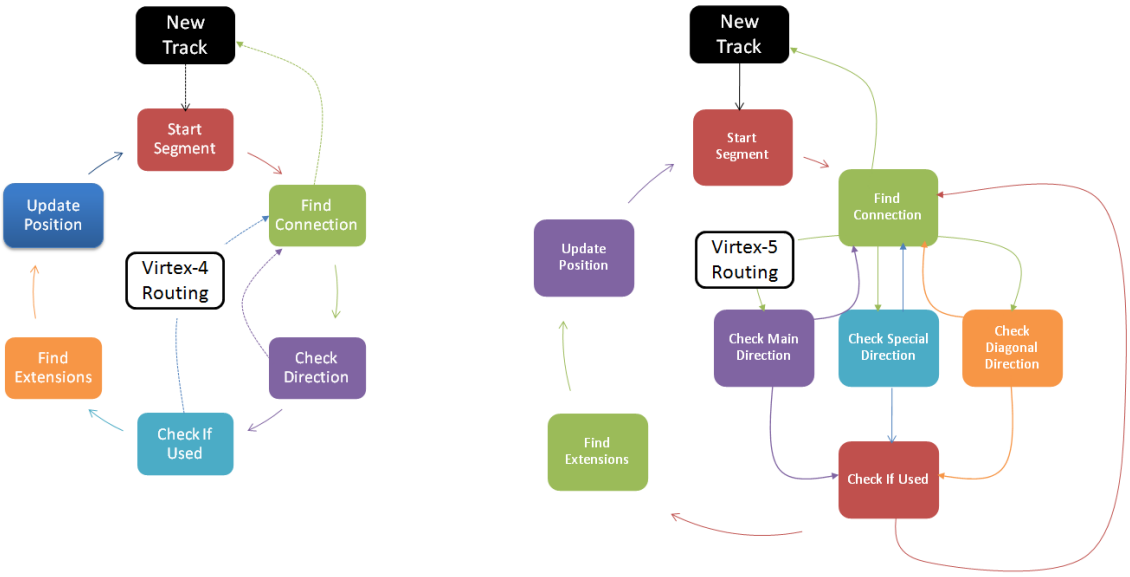

Figure 4.6: Virtex-4 vs. Virtex-5 Routing Scheme.

Main Direction : first the router checks if there are segments that travel in the main direction. For example, if the router intends to travel to west, it first checks for double segments that travel only west.

Special Direction : if there are no segments that meet the criteria, the router checks for segments in special directions. Sometimes, these segments can lead the router out of the designated channel, therefore, a channel check ensures such segments are skipped.

Diagonal Direction : finally, if there are no non-diagonal segments, then the router checks for diagonals going in the direction of the sink. For example if we are going east and the sink is below the source y-coordinate, then only south-east and east-south segments are used.

Another important aspect of the diagonal segments is the separation of the mid and

end segments. Instead of having one segment with two extension, the router uses separate segments for each extension. This is important because it gives the router more options. For example, the diagonal segment SE2BEG0 from CLBX5Y10 has a mid extension at CLBX5Y11 and an end extension at CLBX6Y11. If the router is unable to find a route from the mid CLB, it still has an option of finding a route from the end extension. Unlike regular segments which have the same interconnect at each extension, diagonal segments have varying interconnect at each extension. Figure 4.7 shows the three different double segments.

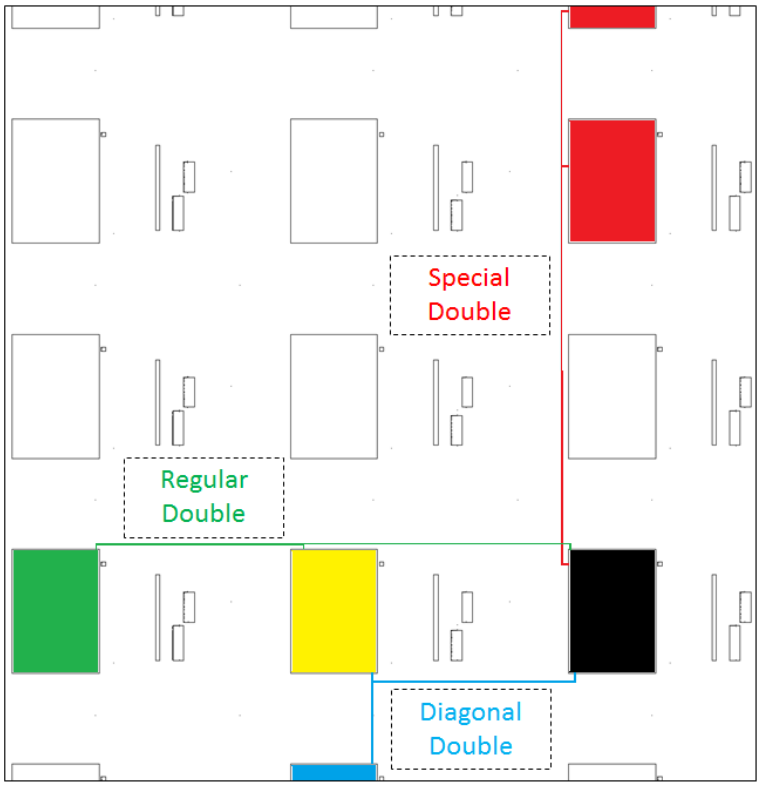

Figure 4.7: Double Segment Types.

Check Used : the last check before the router picks a segment is to see if another router

is already using the PIP in the switch matrix. This ensures that multiple sources are not using the same interconnect which can cause conflicts in the future.

Extension : once the segment passes the above tests, it is selected by the router. Then depending upon the distance, the router selects the extension.

Update List : the router then updates the current interconnect and continues the process until the sink CLB is found.

4.3.4 XDL

Xilinx Device Language (XDL) is a programming tool used by developers to read and write Native Circuit Description (NCD) files [19]. In the AgileHW flow, XDL is used to verify the connections made by the router. When the Agile Router makes the connections for the modules, the designer needs to verify these routes. AgileHW has a XDL function that writes a output file with connection information in XDL. This file can be read and converted into a NCD file, which is then used to verify the routes validity using FPGA Editor.

Verification is a crucial step for the router because the user can find mistakes such as wrong PIP connections, wires, LUT connections, or CLB. Another advantage of this step is the ability to perform a Design Rule Checking (DRC) in FPGA editor. DRC is a built-in tool that checks for logical and physical errors in the design. The XDL file has a detailed description of each CLB, slice, LUT, pip, and wire used for the connection. Using this information, the designer can verify if the module is placed in the correct region, and if the

input and output sources connected using the correct wire segments.

The XDL file has three sections, design name, instance description, and pin connections. Using this information the developer can either verify the information syntactically or by converting the XDL file into an NCD file which can be viewed in FPGA Editor. An example of the XDL code output can be seen in the appendix.

- Design Name: this section contains the name of the design and the device information such as architecture, family, and part.
- Instance Description: this section provides the CLBs and Slices used. Since the router is only being used to connect CLBs, other device instances such as IOB or DSP will not appear. When the developer is verifying the design, this section will show if wrong CLBs are being connected.
- Pin Connections: this is the most important part of the XDL file. This section describes the nets being connected by the router. Each net has an input and output pin which correspond to the LUTs of the source and sink CLBs. Then the pip connecting the interconnect wires between the CLBs are printed out. The developer can look at the wires used to make the connections and ensure that only the wires from the routing database are being used.

If the errors in the XDL file are not immediately obvious, FPGA Editor can be used to get a visual representation of the routed nets and thus making it easier for the developer to

debug. Another advantage of using FPGA Editor is the fine grain debugging tools available to the user such as the ability to check the logic for each net and CLB. And, FPGA Editor also allows the developer to modify the design and update the NCD. The developer can then convert the new NCD into XDL and determine what changes need to be made for the router. The XDL tool kit provided by Xilinx makes verifying the router design a manageable task. For an example of the XDL output of the router, see Appendix A.

4.3.5 Bitstream Stitching Tools

The bitstream tools created for the test flow have two major tasks, find the changed frames and add the functionality to the main bitstream. In order to do this, two different methods were devised. The first was to find the changed frames in the test bitstream and 'or' them with the corresponding frames in the main bitstream. In the second method, once the changed frames in the test bitstream are found, they are appended after the last frame of the main bitstream. Both of these tools are able to stitch the two full bitstreams together for the purpose of the test flow.

Bitstream 'OR' Bitstream ORing is a concept already explored in the Xilinx PR flow [4]. However, instead of making incremental changes in FPGA Editor and ORing the changes with the existing bitstream, this tool intends to add two bitstreams together. The user has the same static bitstream used in the AgileHW flow with the empty left hand side. However, full bitstreams are used in the place of partial bitstreams. The full

bitstreams have the module logic according to the empty region in the static bitstream. The program finds the changed frames and 'OR's them with the corresponding frames in the static bitstream. This process is shown in Figure 4.8.

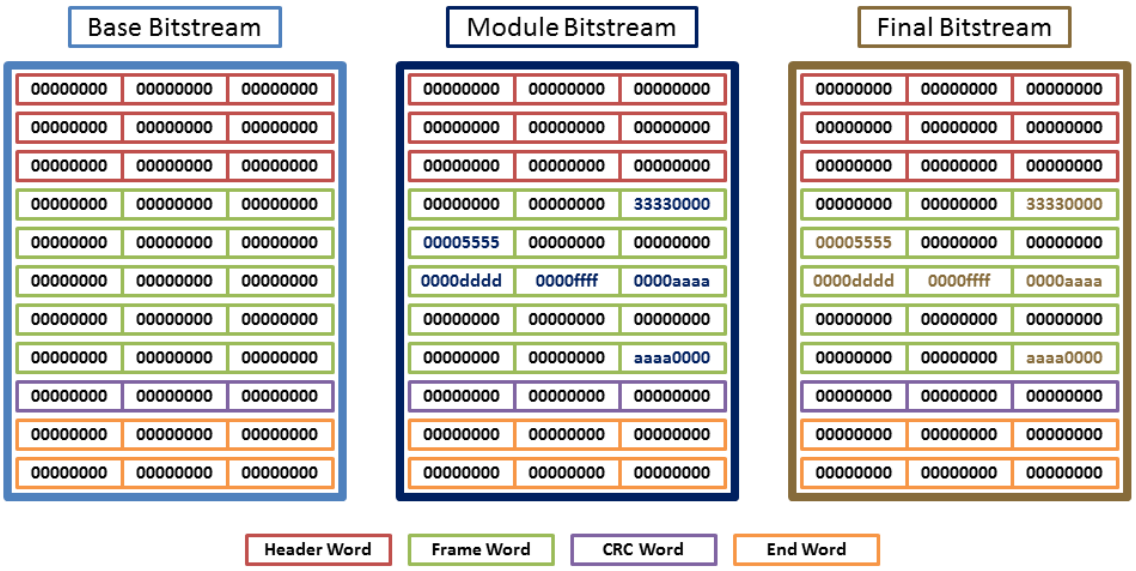

Figure 4.8: Bitstream 'OR'

Bitstream Append The other concept explored for bitstream stitching was bitstream appending. This process is a bit more complicated than the previous method because the tool needs to track the Frame Address Register (FAR). Each frame has a particular address that describes the location of the frame in the bitstream. When the tool parses the module bitstream for changed frames, it stores that frame data and the corresponding FAR. After this process is completed, the stored frames and FAR are added after the last frame of the static bitstream. This process is shown in Figure 4.9.

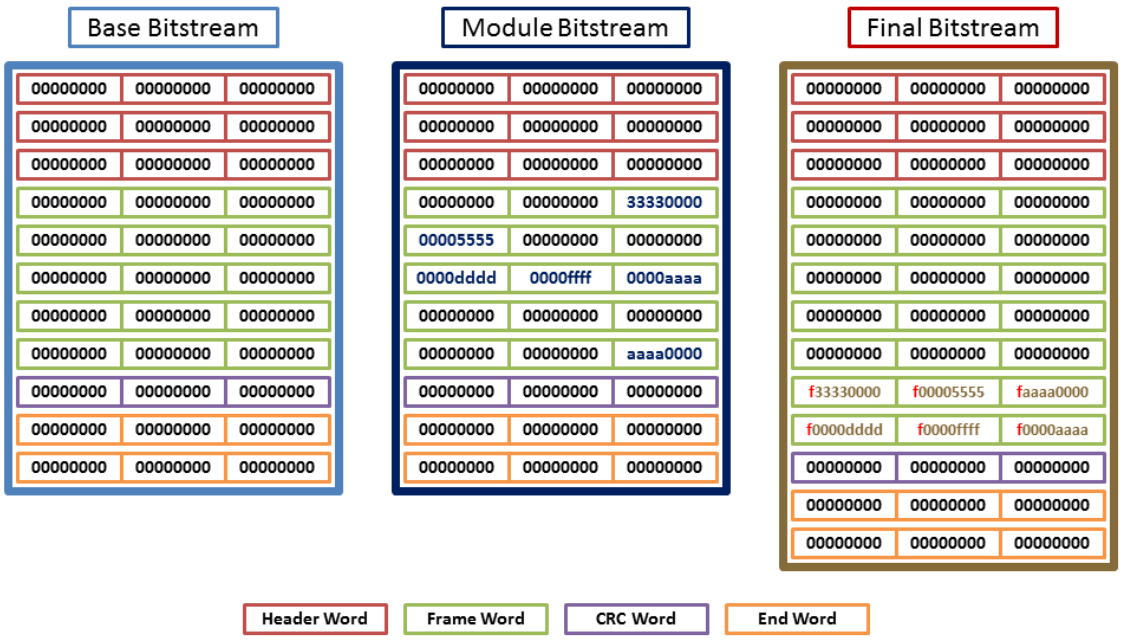

Figure 4.9: Bitstream Append

Chapter 5

Implementation & Results

This chapter presents and analyzes the results of the implementation of the Virtex-5 AgileHW framework. The implementation cycle of the static and dynamic design are explained. The results of the static build, partial module build, router, and framework changes are examined. Finally, the numerous challenges faced during this process are discussed.

The platform was tested and implemented on a Digilent XUPV5 development with a Xilinx Virtex-5 LX110T FPGA. The HDL designs were created using Xilinx ISE 9.2 PR toolkit. Other built-in Xilinx tools such as FPGA Editor and XDL conversion tools were used for the router database and error checking.

5.1 Static Design

The static portion of the FPGA consists of basic test mechanisms such as LEDs and switches. It also contains an empty sandbox region that will be used to place and remove modules. For more complicated synchronous designs, the static part of the FPGA will have other parts such as ICAP, Micro-blaze, clocking and other logic that do not change during run-time. The flow diagram shown in Figure 5.1 describes the steps for the sandbox bitstream development. The designer prepares the top-level HDL file with the static logic and the proper instances. The makefile invokes the Open PR tools and automates the bit generation process. OpenPr was used instead of Xilinx PR because it offers better support for Virtex-5 FPGAs. The Open PR tools produce the static bitstream. The XML file for the current framework is user-made. The framework places the bitstream file, and the router uses the XML file to make the connections between the source and sink CLBs. The test top-level design created for the implementation is shown in Figure 5.2.

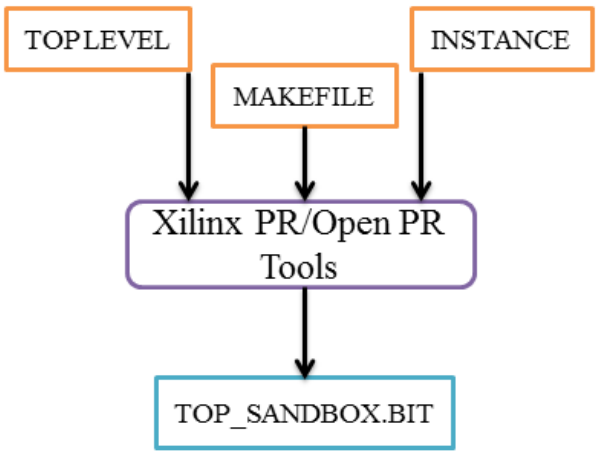

Figure 5.1: Tool flow for Virtex-5 Sandbox build

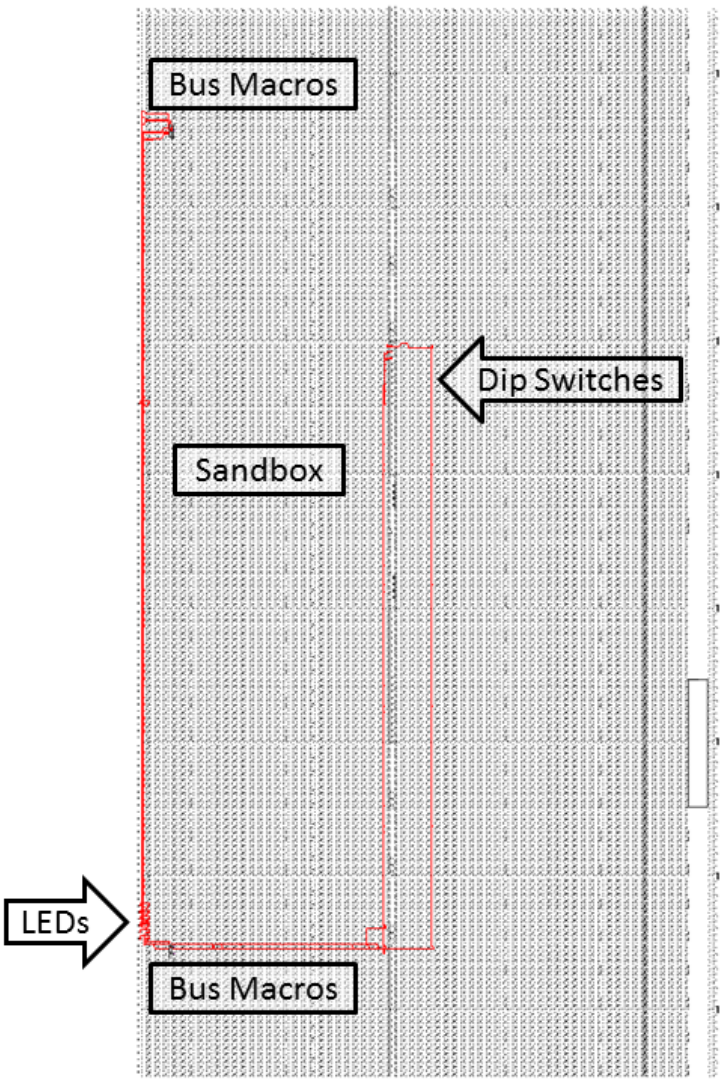

Figure 5.2: Empty Sandbox Region on Virtex-5

5.2 Dynamic Design of Partial Modules

The dynamic design section details the implementation of the partial module, which involves bus macro development, XML and template file changes, and test cases for the AgileHW framework. The partial modules for the framework were made using the Xilinx 9.2 PR tools.

Although the tool flow is the same as shown earlier in Figure 3.1, there were changes made to bus macros and XML files to accommodate for the architecture changes.

5.2.1 Bus Macros

The bus macro design and placement has been changed for the Virtex-5 family to a single slice design. In earlier revisions, a bus-macro involved two CLBs. Although custom macros can be used for Virtex-5 that follow the two CLB model as seen in Figure 4.3, the partial modules were made using standard single slice macros as seen in Figure 5.3. Placement is another key change for bus macros. Because of the two CLB design of earlier bus macros, they needed to straddle the PR boundary regions. However, in Virtex-5 the single slice bus macros are placed inside the PR region.

5.2.2 XML File

The XML files for the Virtex-5 follow the same protocol as before. They contain the information about device type, module name, data flow, dimensions, site offsets, and input/output information. The offsets are calculated from the origin CLB X,Y co-ordinates.

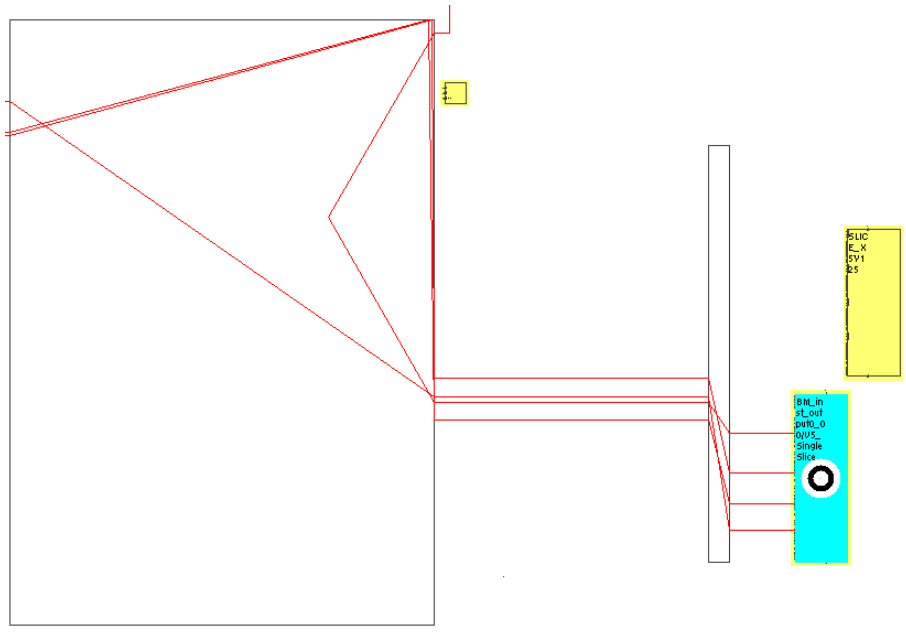

Figure 5.3: Virtex-5 Single Slice Bus Macro

Inverter

The inverter is another partial module used to test the AgileHW flow. Once a pass through module is placed in the dynamic region, the inverter is placed after it. This test case helps demonstrate the agile router's capabilities for run-time connectivity. The inverter module takes the inverts the input to the LEDs and can be seen in Figure 5.4.

5.3 Routing

The routing of the modules was handled by the router described in Chapter 4 and 5. The router was used to connect the source and sink bus macros placed around the modules. Once

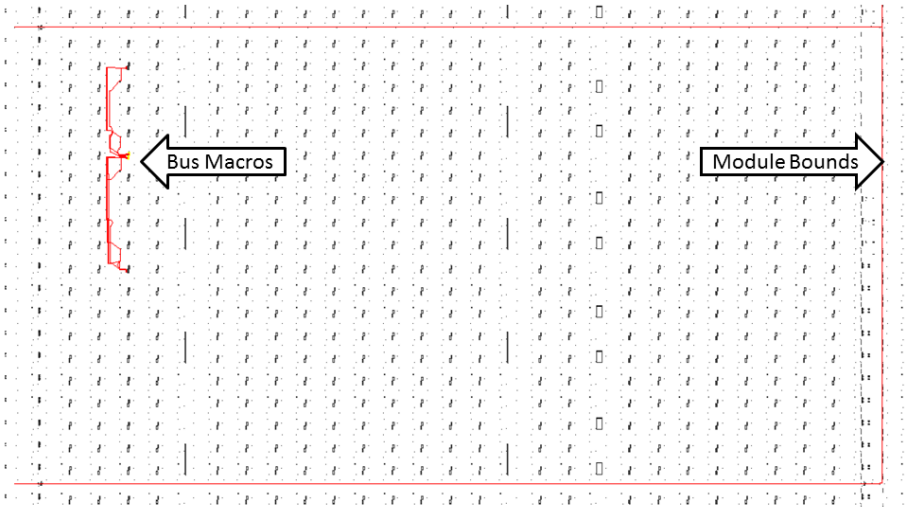

Figure 5.4: Inverter Partial Module

the router creates the routes, it also outputs an XDL dump of the PIPs used. Using XDL to NCD conversion tool, the routes can be verified in FPGA Editor. Figure 5.5 shows the routes between a source CLB in the east corner of the dynamic region to a sink in the west corner. Figure 5.6 shows the routes between a source CLB in the north corner of the dynamic region to a sink CLB in the south corner.

Figure 5.5: Routing from a source to sink CLB horizontally

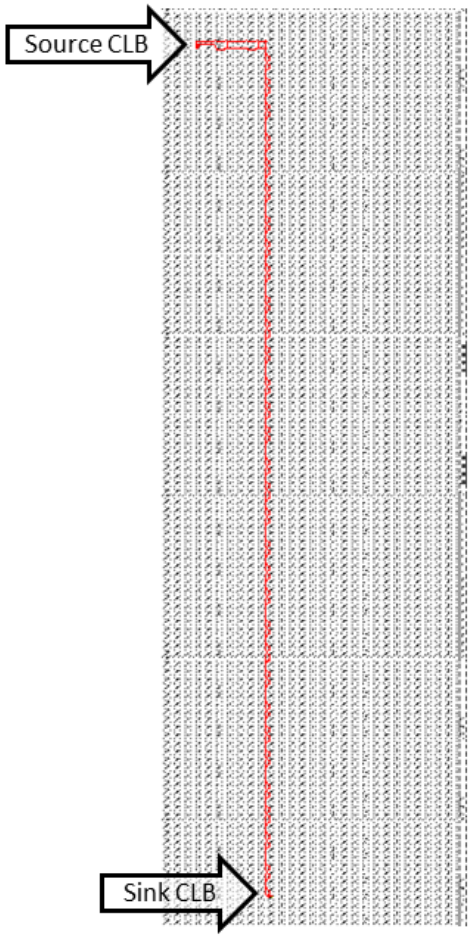

Figure 5.6: Routing from a source to sink CLB vertically

5.4 Results

In this section, a pass through test is implemented and explained. Also, the router's performance is analyzed according to routing time, delay of routes, and number of segments for varying distances.

5.4.1 Pass Through

The pass through test is a basic test case used to help demonstrate the functionality of the router and static region. The logic for module takes the sandbox inputs and pass the information through to the sandbox output. In this test case, the XUPV5 LEDs were used to ensure that data was flowing properly.

The router is used to connect the source and sink bus macros according to the sandbox XML file. The router's XDL information is then combined with the static design's XDL to create the full NCD. Then using Bitgen (Xilinx bitstream generator tool), the full bitstream is created. After which, Bitgen is used again to produce the difference bitstream of the full bitstream and the route bitstream. This is the partial bitstream of the pass through test. Figure 5.7 describes this process.

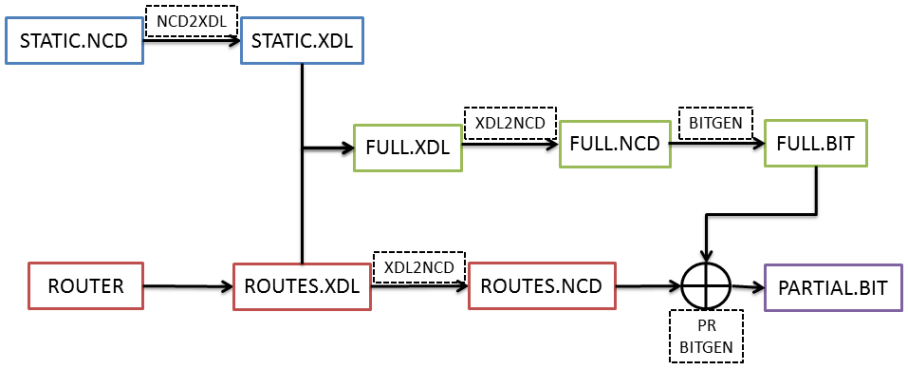

Figure 5.7: Process of creating partial bitstream for pass-through test.

The experiment was done on a host machine with Intel Core 2 Duo, 4 GB ram, and running Ubuntu OS. A Virtex-5 LX110T board was programmed with the JTAG chain using Xilinx Impact. Figure 5.8 shows the setup of the experiment.

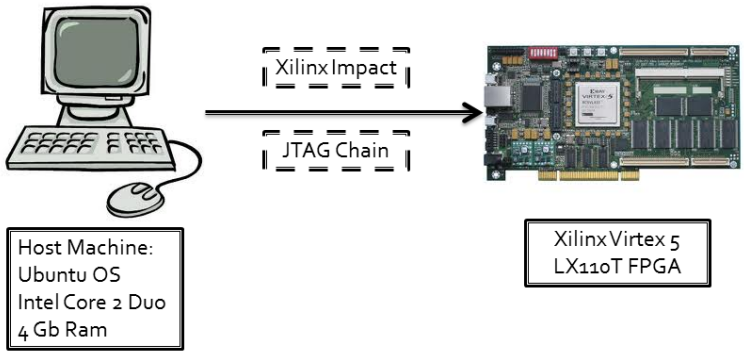

Figure 5.8: Setup for the pass through test.

The partial module of the routes connecting the source and sink bus macros is loaded to test that pass through is operational. A depiction of this is seen in Figure 5.9. The static full bitstream is 3,889,955 bytes long, and the partial bitstream is a menial 9,185 bytes. Impact takes approximately 6-10 seconds to program the full bitstream, and less than 1 second to program the partial bitstream. When the partial bitstream is loaded, the corresponding bits in the full bitstream are programmed with the partial bits. The results were verified by checking if the LEDs to the switches.

5.4.2 Routing Results

The routing time of the AgileHW router and FPGA Editor was determined by using the function *gettime* from the *time.h* library. The XDL output of the route was used to determine the number of segments used to make the route. The converted NCD file was used to analyze the delay using FPGA Editor.

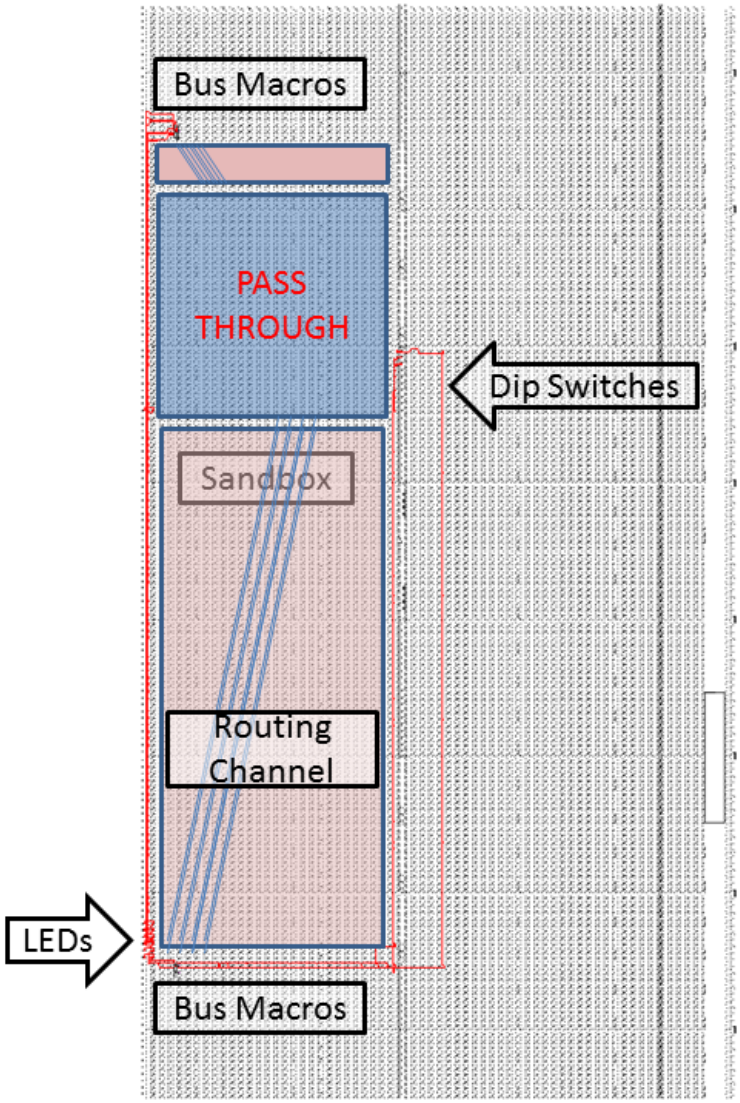

Figure 5.9: Pass through bitstream test result.

Table 5.1 shows the information for the vertical routes made using the AgileHW router. The router was forced to check routes the length of the sandbox. The routing time decreases as the distance between the source and sink CLBs decrease. This is rational because the router is doing fewer computations and finding fewer segments as the distance is reduced. A very uniform pattern emerges for the number of segments due to the homogeneous FPGA

Table 5.1: Vertical Routing with AgileHW Router for the length of the sandbox region

Distance (CLBs)	120	100	80	60	40	20
Routing Time (S)	0.192	0.182	0.179	0.170	0.160	0.155
Average Delay (NS)	10.93	9.13	7.38	5.70	3.83	2.15
Average of Segments	72	60	48	36	24	12

Table 5.2: Vertical Routing with FPGA Editor Router for the length of the sandbox region

Distance (CLBs)	120	100	80	60	40	20
Routing Time (S)	0.094	0.092	0.096	0.091	0.091	0.090
Average Delay (NS)	4.33	3.65	3.14	2.75	1.97	1.45
Average of Segments	11	10	9	8	7	6

interconnect.

Table 5.2 shows the data derived from making the exact routes in FPGA Editor. The delays and number of segments used is significantly less than the AgileHW router. This can be attributed to the lack of *pent* segments and long lines in the AgileHW routing database. Also the routing time is lower due to a more robust program employed by FPGA Editor. The delays are far lower than the AgileHW router because FPGA Editor uses significantly fewer segments to make the connections.

Although the routing channel shown in Table 5.1 stress the AgileHW router's capabilities, when modules are placed in the PR region, the routing channel is significantly smaller. Table 5.3 and Table 5.4 shown above, emulate the routing paradigm seen in real time designs. The

Table 5.3: Vertical Routing with AgileHW Router for the expected channel length

Distance (CLBs)	20	15	10	8	6	3
Routing Time (S)	0.146	0.144	0.143	0.143	0.145	0.142
Average Delay (NS)	2.13	1.79	1.27	1.28	0.90	0.61
Average of Segments	12	10	6	5	4	3

Table 5.4: Vertical Routing with FPGA Editor Router for the expected channel length

Distance (CLBs)	20	15	10	8	6	3
Routing Time (S)	0.090	0.091	0.091	0.090	0.091	0.090
Average Delay (NS)	1.45	1.25	1.01	0.97	0.72	0.62
Average of Segments	6	5	4	4	3	2.5

number of segments used by FPGA editor is lower for longer distances because of AgileHW limited database. Although the delay is still longer for the AgileHW router, it is much closer in range than before.

Table 5.5 and 5.6 have the information for horizontal routes made by the AgileHW router

Table 5.5: Horizontal Routing with AgileHW Router for the width of the sandbox

Distance (CLBs)	24	20	17	14	9	5
Routing Time (S)	0.112	0.111	0.111	0.109	0.108	0.107
Average Delay (NS)	2.971	2.573	2.263	1.953	1.348	0.918
Average of Segments	16	14	12	10	6	4

Table 5.6: Horizontal Routing with FPGA Editor Router for the width of the sandbox

Distance (CLBs)	24	20	17	14	9	5
Routing Time (S)	0.095	0.093	0.093	0.096	0.092	0.091
Average Delay (NS)	1.95	1.62	1.48	1.21	0.96	0.68
Average of Segments	7	6	6	5	5	3

and FPGA Editor respectively. The distance traveled is comparatively lower because the sandbox region is only restricted to the left hand side of the device. Once again, the delays are much lower for FPGA Editor due to the number of segments used.

5.5 Challenges

5.5.1 ICAP Issues

The Xilinx ICAP interface was used in previous versions of AgileHW. It was also supposed to be used with this version of the AgileHW tools. For the proposed flow, a micro-blaze was to be used to do the module processing such as placement, bitstream stitching, and routing. The bitstream information would then be passed to the sandbox through the ICAP. However, the ICAP for Virtex-5 with 9.2 Partial Reconfiguration flow is not well supported, and thus the future move to in-house bitstream loading tools covered in Chapter 5.

5.5.2 Interconnect Issues

The Virtex-5 architecture allows the router to reach tiles faster using a single diagonal path instead of two segments to achieve the same result. The interconnect was designed to allow the router to be flexible in the track allocation as seen in Figure 5.10, and the denser slice architecture gave the router more resources in a single CLB to route the data through.

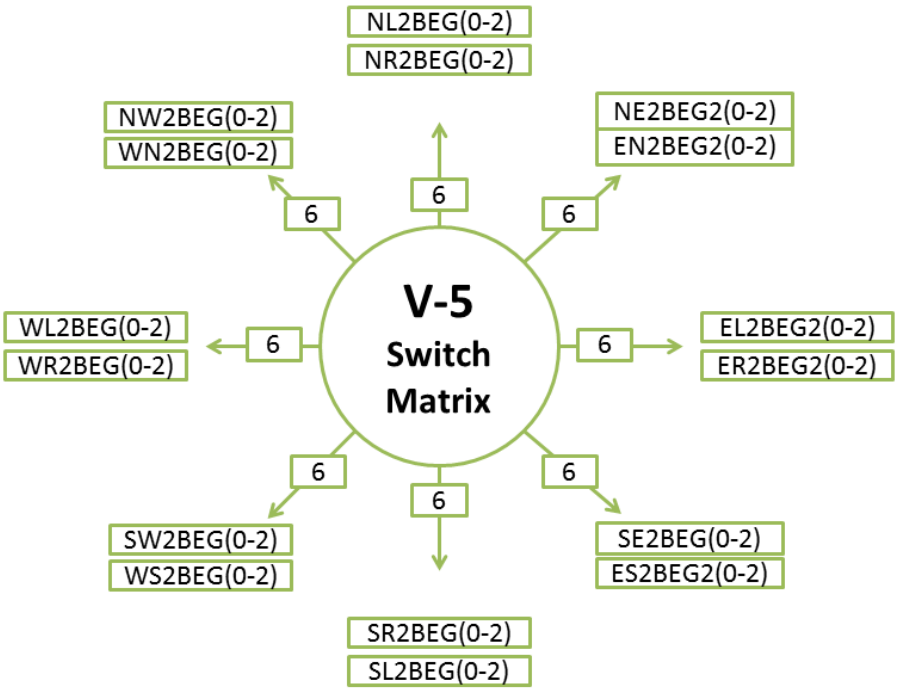

Figure 5.10: Virtex-5 Interconnect

Since the AgileHW router is restricted to double segments only, its capabilities are restricted. A connection from one tile to another with the fewest hops is desired because of its low delay and resource usage. However, this paradigm is harder to meet when using only double segments to travel a long distance. Since the router connected sources and sinks in close vicinity, timing and wire complexity was not a concern. Another factor for using only double

segments is because the Virtex-4 interconnect was tailored to move in the major directions as it is seen in Figure 5.11. There are twelve possibilities to move in a major direction unlike in Virtex-5 where there are only size possibilities. Although, Virtex-5 diagonals can be used to appease this problem, they warrant extra hops and channel errors. Thus, in future revisions of the router, *pent* segments need to be implemented for improved connectivity.

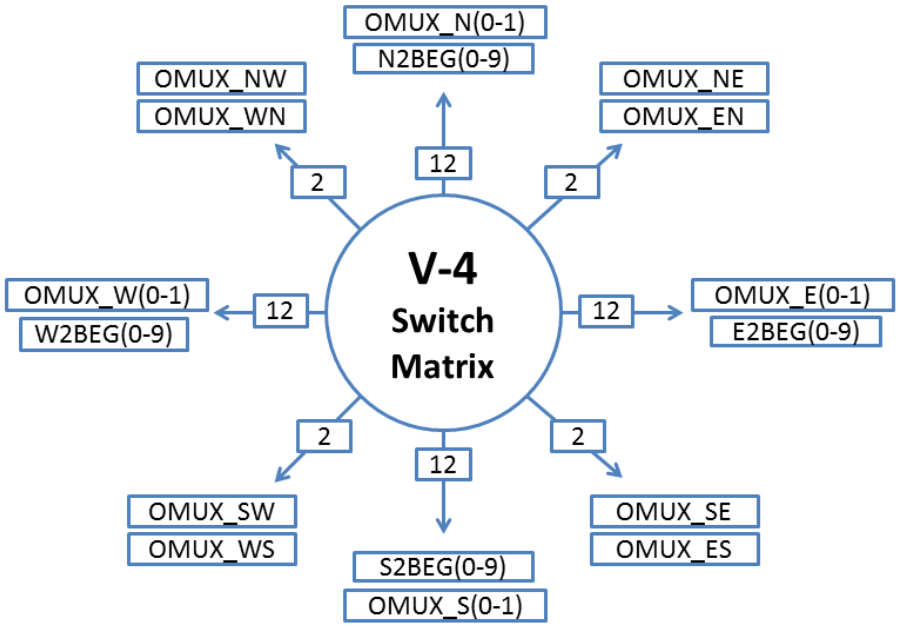

Figure 5.11: Virtex-4 Interconnect

Slice Issues

Another issue was the sparse interconnectivity from memory slices (SLICE M) as seen in Figure 5.12. Even with the addition of special segments to travel the major directions and revamped router, making connections was still a challenge. However, since Virtex-5 only uses single slice macros, the router is restricted to only Slice L's for routing.

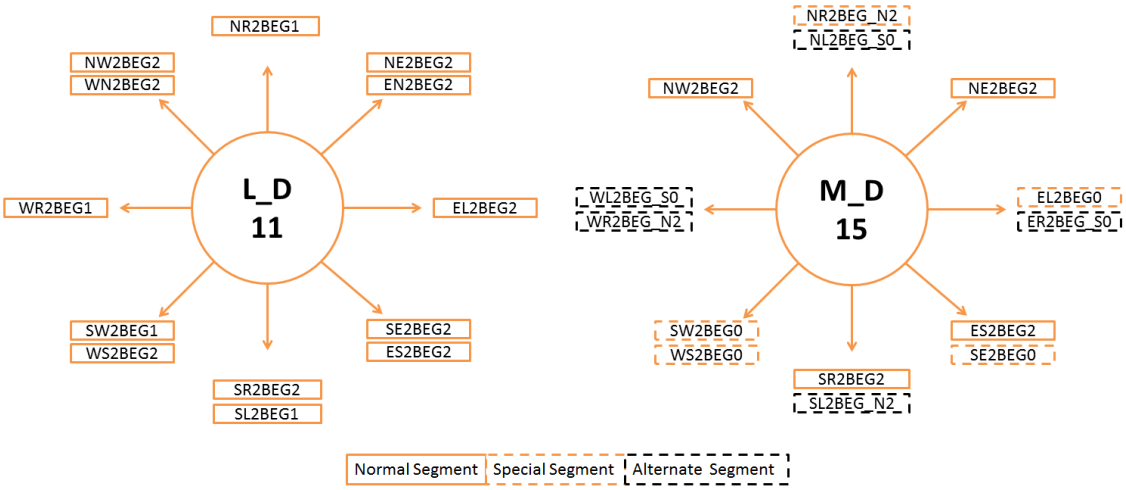

Figure 5.12: Slice L vs Slice M Interconnect Issues.

5.5.3 Unsupported Tool Kit

Most of the major issues faced during the AgileHW framework development could be traced back to poorly supported 9.2 PR tools for Virtex-5. There is sparse exposure and help documentation for Xilinx PR flow.

Chapter 6

Conclusions

FPGAs are dynamic platforms that allow users to configure various designs on a single chip. Due to this flexibility and relatively small size, they are used for many applications such as cell phones, set-top boxes, wireless communications, video filtering, and medical equipment. However, with the advancement of silicon technology, FPGAs have become extremely dense and complicated, which increases the design and implementation time drastically. Therefore, in order for FPGAs to evolve, there is a need for flexible development tools similar to software.

AgileHW is a run-time reconfigurable FPGA design tool that allows developers to quickly develop and deploy dynamic designs. Instead of using one huge design with all the logic, the developer has the option to modularize the most commonly changed logic. This allows for faster implementation and debug. This agility is a crucial advantage that software applications had over hardware platforms. However, with the advent of dynamic FPGA frameworks,

developers no longer have to sacrifice the computational capabilities of hardware in exchange for the flexibility of software design.

This thesis presents the dynamic framework's evolution from an aging FPGA family to an updated platform. With the enhancements in the interconnect, router, and FPGA architecture, future developers will be able to build vastly more complicated designs on the new hardware. Another advantage of this work is the portability to future platforms. The newer Virtex-6 family is relatively similar to Virtex-5, thus the AgileHW framework can also be quickly adapted for these platforms.

6.1 Contributions

Device Information - information about the various Virtex-5 devices (clock regions, BRAMs, DSPs, CLBs, LUTs, frames, etc.) were added to the framework

LUT Manipulation - the LUT manipulation tool information was added and tested using bitstream differentiation tools and FPGA Editor.

Interconnect Information - the Virtex-5 interconnect database was developed to allow the router to connect designated CLBs.

Enhanced Router - router was upgraded to work with the vastly different Virtex-5 interconnect. Extra checks were added to prohibit diagonal and special segments from leading the router outside the channel. Router was rigorously tested to route horizon-

tally and vertically.

XDL - the XDL tools were upgraded for Virtex-5 so developers can debug the router using FPGA Editor.

Bitstream Manipulation Toolkit - bitstream 'OR' and appending tools were developed and tested for alternate bitstream manipulation techniques.

Pass Through Test - a pass through test was performed to ensure that the framework and router was completely operational.

6.2 Future Work

6.2.1 Clock Tree

Although the framework upgrades were extensive, there is still some work to be done. One crucial addition is the clock tree. Currently, the designs are asynchronous due to the missing clock bit information for the Virtex-5. Once this information is added, synchronous module logic can be developed to aptly exploit the capabilities of Virtex-5.

6.2.2 Routing Scheme

The current application for the router requires connections to be made from one CLB to another in relatively close proximity. The router uses a channel routing algorithm. This

paradigm was very successful for Virtex-4 because it had a multitude of double segments in each of the major directions. However, using this approach for the Virtex-5 interconnect is unnecessarily complicated. One easy fix for this issue is to add pent segments to the routing database. Pent segments travel five CLBs in each direction unlike double segments which only travel two CLBs. This addition gives the router more viable options to travel from source to sink CLBs.

6.2.3 Freedom from PR Tools

Open PR is an open source partial reconfiguration toolkit developed for Xilinx FPGAs at the Configurable Computing Lab. It is a module based PR tool based on Xilinx 9.2 PR tool flow. Open PR allows the designer to develop slot based modules that can be placed and removed during run time. Such an open source tool has the support of the FPGA development community and thus would be better supported.

Bibliography

[1] M. L. Silva and J. C. Ferreira, "Creation of Partial FPGA Configuration at Run-Time," in *13th Euro*, 2010.

[2] M. Huebner, T. Becker, and J. Becker, "Real-time LUT-Based Network Topologies for Dynamic and Partial FPGA Self-Reconfiguration," *ACM*, 2004.

[3] C. Bobda and A. Ahmadinia, "Dynamic Interconnection of Reconfigurable Modules on Reconfigurable Devices," in *IEEE Design and Test of Computers*. IEEE, 2005, pp. 443–551.

[4] *Two flows for partial reconfiguration: Module based or difference based*, Xilinx Inc., September 2004.

[5] P. Athanas, J. Bowen, T. Dunham, C. Patterson, J. Rice, M. Shelburne, and J. Suris, "Wires On Demand: Run-Time Communication Synthesis for Reconfigurable Computing," *FPL*, 2007.

[6] J. Suris, M. Shelburne, C. Patterson, P. Athanas, J. Bowen, T. Dunham, and J. Rice, "Untethered On-The-Fly Radio Assembly with Wires-On-Demand," *NAECON*, 2008.

[7] J. Suris, C. Patterson, and P. Athanas, "An Effecient Run-Time Router for Connecting Modules in FPGAs," *FPL*, 2008.

[8] Xilinx, *Virtex-5 Platform FPGA Family Technical Backgrounder*, May 2006.

[9] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems," *IEEE*, 1998.

[10] F. Berthelot and F. Nouvel, "Partial and Dynamic Reconfiguration of FPGAs: a top down design methodology for an automatic implementation," in *Emerging VLSI Technologies and Architectures*, 2006.

[11] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh, "A Quick Safari through the Reconfiguration Jungle," *ACM*, 2001.

[12] J. Becker, M. Hubner, and M. Ullmann, "Run-Time FPGA Reconfiguration for Power-/Cost-Optimized Real-Time Systems," in *From Systems to Chips*, M. Glesner, R. R., and I. L., Eds. Boston:Springer, 2006, vol. 200, pp. 119–132.

[13] C. Conger, R. Hymel, M. Rewak, A. George, and H. Lam, "FPGA Design Framework for Dynamic Partial Reconfiguration," *RAW*, 2008.

[14] T. Pionteck, C. Albrecht, R. Koch, E. Maehle, M. Hubner, and J. Becker, "Communication Architectures for Dynamically Reconfigurable FPGA Designs," *IEEE*, 2007.

[15] T. Yoshimura and E. Kuh, "Effecient Algorithms for Channel Routing," *IEEE*, 1982.

[16] Xilinx, *Virtex-5 Family Overview*, Febuary 2009.

[17] ——, *Virtex-4 Family Overview*, December 2010.

[18] ——, *Virtex-5 FPGA Configuration User Guide*, August 2009.

[19] P. A. Jurgen Becker, Roger Woods, *Reconfigurable Computing: Architectures, Tools, and Applications*, F. Morgan, Ed. Springer, 2009.