Improving Field-Programmable Gate Array Scaling Through Wire Emulation

Ryan Joseph Lim Fong

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

Masters of Sciences
In
Computer Engineering

Dr. Peter Athanas – Co-Chair
Dr. Mark Jones – Co-Chair
Dr. Cameron Patterson

September 3, 2004
Blacksburg, VA

Keywords: wire, scaling, FPGA, Xilinx, Virtex-II, self-reconfiguration, ICAP, emulation
Copyright 2004, Ryan Joseph Lim Fong

Improving Field-Programmable Gate Array Scaling Through Wire Emulation

Ryan Joseph Lim Fong

Abstract

Field-programmable gate arrays (FPGAs) are excellent devices for high-performance computing, system-on-chip realization, and rapid system prototyping. While FPGAs offer flexibility and performance, they continue to lag behind application specific integrated circuit (ASIC) performance and power consumption. As manufacturing technology improves and IC feature size decreases, FPGAs may further lag behind ASICs due to interconnection scalability issues. To improve FPGA scalability, this thesis proposes an architectural enhancement to improve global communications in large FPGAs, where chip-length programmable interconnects are slow. It is expected that this architectural enhancement, based on wire emulation techniques, can reduce chip-length communication latency and routing congestion. A prototype wire emulation system that uses FPGA self-reconfiguration as a non-traditional means of intra-FPGA communication is implemented and verified on a Xilinx Virtex-II XC2V1000 FPGA. Wire emulation benefits and impact to FPGA architecture are examined with quantitative and qualitative analysis.

Acknowledgements

First, I would like to thank my Co-Advisors, Dr. Peter Athanas and Dr. Mark Jones, for supporting me in completing my Master's degree, and for providing a great experience in the Configurable Computing Laboratory (CCM Lab). Dr. Athanas, who graciously lent his time during the writing of this thesis, has provided excellent guidance and encouragement throughout the entire process. Dr. Jones first introduced me to world of configurable computing systems, and I've been hooked ever since. In addition, I would like to equally thank Dr. Cameron Patterson, who has provided great insight into Xilinx FPGA architectures and some of the mysteries of partial-reconfiguration.

My family and I will always be indebted to the Bradley Department of Electrical and Computer Engineering and the Via family, who have financially supported me throughout the majority of my career at Virginia Tech with a Bradley Fellowship and a Bradley Scholarship. To know that an entire department believes in your ability, allowing one to pursue one's educational interests, without the worries of financial issues, is phenominal. This thesis would not have been possible without support of the Bradley Fellowship.

Thanks to all past and present members of the CCM Lab that I've had the privilege to work with. There have been countless times where we've lent each other a hand, providing each other with creative ideas and support in times of difficulty. This thesis is a culmination of many ideas that I have improved upon because of your listening.

A tremendous amount of thanks go to my loving family, who has relentlessly supported me in reaching my goals. To my friends – thank you for making sure I that I relax outside of the lab.

To my colleagues at EVI, thank you for allowing me to take the time off to complete my degree.

Last but not least, I would like to thank all of you who made a stand, and were committed that I complete this degree, even if I was not willing to kindly take your advice.

Table of Contents

Table of Contents iv
List of Figures vi
List of Tables vii
Introduction 1
1.1 Contribution 4
1.2 Thesis Organization 4
Background and Related Work 5
2.1 Wire Scaling 5
2.2 FPGA Architecture 7
2.2.1 Architectures 8
2.2.2 Interconnection and Logic Overhead 10
2.3 FPGA Scaling 11
2.4 Wire Emulation 13
2.4.1 FPGA Self-Reconfiguration 13
2.4.2 Network-on-Chip 14
FPGA Architecture Enhancement 17
3.1 Goals 17
3.2 Concept 18
3.3 Methodology 19
Wire Emulation Prototype 22
4.1 Overview 22
4.2 Concept 23
4.2.1 Configuration Memory 25
4.2.2 ICAP 26
4.3 System Design and Implementation 27
4.3.1 System FSM 28
4.3.2 Transports 29
4.3.3 ICAP Frame Reader 34
4.3.4 ICAP Frame Writer 35
4.3.5 ICAP Frame Copier 36
4.4 ICAP Multi-Frame Write 36

Results and Analysis 40
5.1 Prototype Wire Emulation System 40
5.1.1 Verification 42
5.1.2 Performance 43
5.1.3 Power 46
5.1.4 Multi-Frame Write Experiment 47
5.1.5 Improvements to Performance 48
5.2 Wire Emulation Architecture Analysis 49
5.2.1 Effects on FPGA Layout 56
5.2.2 Effects on FPGA Design Tools 58
Conclusions 61
6.1 Summary 61
6.2 Future Work 63
6.3 Conclusions 64
Bibliography 65
Vita 69

List of Figures

Figure 1.1 - General FPGA architecture.. 2
Figure 1.2 - Original FPGA Hop Count 3
Figure 1.3 - Enhanced FPGA Hop Count.. 3
Figure 2.1 - 2003 ITRS scaling comparison.. 7
Figure 2.2 - Xilinx Virtex-II architecture 8
Figure 2.3 - Xilinx Virtex-II switch interconnections 9
Figure 3.1 - Conceptual wire emulation system 19
Figure 4.1 - Prototype concept 24
Figure 4.2 - Transport user interface options 25
Figure 4.3 - Virtex-II configuration architecture.. 26
Figure 4.4 - Prototype system design 27
Figure 4.5 - Virtex-II CLB and slice structure 30
Figure 4.6 - Register emulation using a single LUT 32
Figure 4.7 - Multi-frame write experimentation system.. 37
Figure 5.1 - Prototype implementation in XC2V1000 41
Figure 5.2 - Potential FPGA interconnect latency.. 51
Figure 5.3 - Simplified wire emulation system 53
Figure 5.4 - Potential effective global wire throughput.. 54
Figure 5.5 - Traditional FPGA user global routing congestion... 55
Figure 5.6 - Enhanced FPGA user global routing congestion.. 56

List of Tables

Table 2.1 - Xilinx Virtex-II extrapolated die size 12
Table 2.2 - Xilinx Virtex-II long wire delay 12
Table 4.1 - ICAP readback sequence 35
Table 4.2 - ICAP frame write sequence 36
Table 4.3 - ICAP frame writer with multi-frame write 39
Table 5.1 - Calculated performance of Xilinx Virtex-II family operating at 46.7 MHz 44
Table 5.2 - FPGA user storage and frame utilization 45
Table 5.3 - Prototype (no debug HW) current consumption 47
Table 5.4 - Estimated broadcast performance on XC2V1000 operating at 46.7 MHz........ 48

Chapter 1

Introduction

Field programmable gate arrays (FPGAs) are an excellent solution for deploying custom logic in low to medium quantities, and have a time-to-market advantage over application specific integrated circuits (ASICs). Static random access memory (SRAM) based FPGAs have a further advantage over ASICs in that they are reprogrammable, allowing for rapid system prototyping and field upgradability. ASIC designs are fixed and highly optimized for a particular solution while FPGAs are flexible, incorporating features that meet the demands of a wide audience. Flexibility comes with cost; ASIC designs are typically faster and consume less power than an equivalent design implemented on an FPGA. Performance improvements and power consumption reduction in FPGAs is paramount, as programmable logic devices are seen as the predominant processor of the future, due to their reconfigurable flexibility and computational efficiency [1]. With FPGAs being less efficient than ASICs, FPGA architecture must evolve in order to close the performance and power gap.

General FPGA architecture, as shown in Figure 1.1, consists of a large sea of configurable logic blocks (CLBs) and an array of input/output blocks (IOBs). A CLB consists of lookup tables (LUTs), flip-flops, and arithmetic carry logic. IOBs interface the configurable logic to the external pins of the FPGA. More recent FPGAs have also included dedicated RAM, multipliers, microprocessors, and specialized high-speed serial transceivers. CLBs, IOBs, and other FPGA resources can communicate through an interconnection network consisting of configurable switches and wires of various lengths. The function to be performed by all FPGA resources, including the interconnection network, is defined by configuration memory.

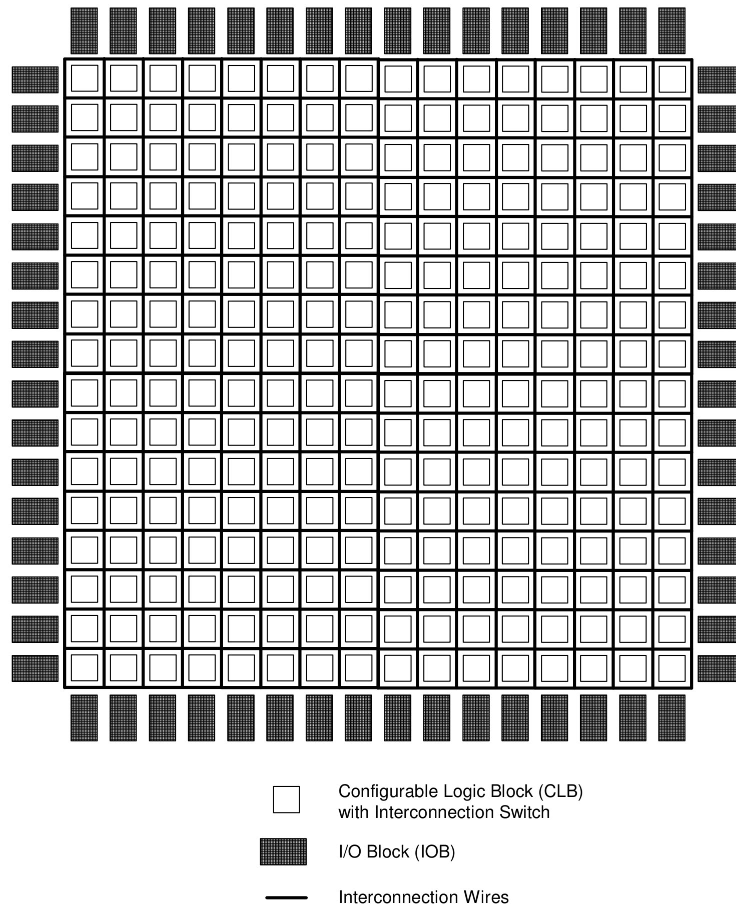

Figure 1.1 - General FPGA architecture

A predominant source of FPGA design delays comes from interconnection overhead, where interconnection historically contributes to 70% of total design delay. Interconnection overheads are found in the configurable switches that segment interconnection wires. Even though configurable switches minimize propagation delays through wire buffering, they introduce gate delays in wiring paths. These unavoidable gate delays keep FPGA performance bellow that of ASICs. To minimize interconnection delays, some FPGAs provide direct wiring connections between CLBs, bypassing the costly configurable switches for local communication. Another technique to minimize interconnection delays is to offer wires of various lengths, allowing FPGA resources to communicate non-locally through paths utilizing a reduced number of configurable switches. Some of these wires bypass several adjacent switches, while others may span the entire length of the FPGA, offering access to a select number of switches. We shall refer to these

FPGA-length wires as chip-length wires. Of the variety of interconnection wires offered, chip-length wires have the longest delays.

Leading-edge FPGAs are being produced in 90nm technology, with FPGAs manufactured with smaller feature sizes coming in the next couple of years. Knowing that FPGAs have substantial interconnection delay and that IC density increases with scaling, performance overheads may worsen if we continue to use traditional FPGA architecture in future FPGAs. Consider that interconnect can account for 70-90% of circuit delay when ASIC designs are manufactured using technologies below 0.18um [2]. One concern of FPGA scaling is chip-length wire performance. Long wires that span the full length of an IC have been known not to scale in relation to local wires. Long wires maintain full length while local wires will shorten under scaling. Wire delay is proportional to wire length, therefore the delay gap between long and local wires increases. Further discussion on how technology scaling may impact FPGA performance is included in Chapter 2. Another interesting issue to investigate is FPGA architecture power scalability, a topic beyond the scope of this thesis.

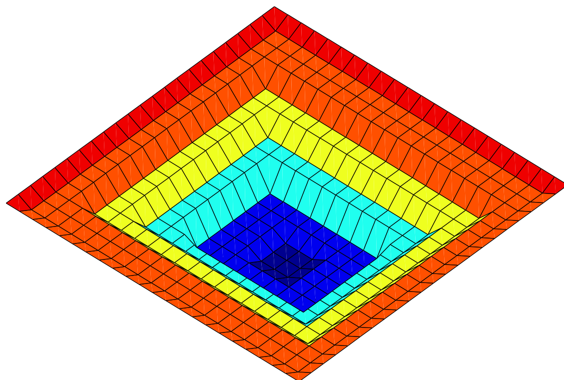

Figure 1.2 - Original FPGA Hop Count

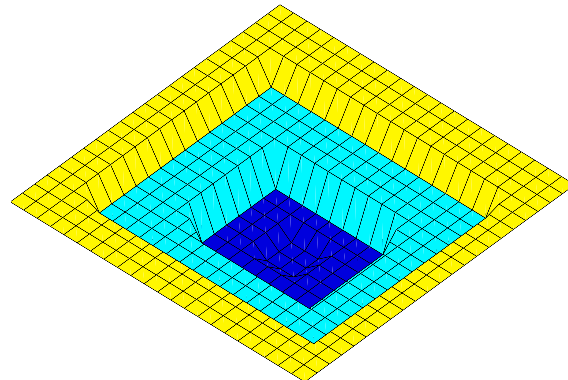

Figure 1.3 - Enhanced FPGA Hop Count

Each square represents an FPGA CLB. Hop count from the center of the FPGA is represented by color. Warmer colors imply greater hop count, where red is the warmest and blue is the coolest.

Strategies must be developed to improve the scalability of chip-length wire performance in FPGAs. A conceptual view of current FPGA interconnection delay is shown in Figure 1.2. If existing methods of FPGA long interconnects were implemented with smaller feature processing technology, edges of the FPGA would become further apart in communication delay. If we could improve the scalability of long interconnects, FPGA long interconnect delay would not grow as quickly, allowing edges of the FPGA to be within timely reach, as shown in Figure 1.3.

1.1 Contribution

This thesis describes an FPGA architecture enhancement that accelerates cross-chip communications and reduces routing congestion. As a result, FPGAs can increase in density with reduced communication penalties. In addition, a prototype that embodies several properties of the proposed enhancement is described.

1.2 Thesis Organization

Chapter 2 of this thesis discusses the background of this work, including trends in wire scaling, a summary of current FPGA architecture, and an investigation into FPGA architecture technology scalability. In addition, strategies to deal with wire scaling issues are discussed, with focus placed on techniques that emulate wire functionality. Chapter 3 proposes an FPGA architecture enhancement that accelerates cross-chip communications and reduces routing congestion, especially in future sub-micron technologies. A prototype that embodies several properties of the proposed enhancement is presented in Chapter 4, while results of this prototype are discussed in Chapter 5. Chapter 5 also analyzes prototype results and FPGA architecture enhancement methodology. In addition, this chapter discusses how the architecture enhancement could impact FPGA layout and FPGA CAD tools. Lastly, Chapter 6 discusses conclusions and future directions for this work.

Chapter 2

Background and Related Work

This chapter discusses background concepts and related work. As mentioned in Chapter 1, FPGA interconnect scaling is of concern. To provide more insight into this issue, trends in wire scaling and an overview of field-programmable gate array (FPGA) architectures are presented. Afterward, FPGA interconnect scalability is discussed, applying the presented wiring trends to FPGA architecture.

To reduce the penalties of FPGA interconnect scaling, this thesis proposes the use of wire emulation as a means for cross-chip communication. This thesis uses the term "wire emulation" to describe techniques that realize the functionality of conventional wire interconnection resources. To this end, previous work in wire emulation technology, both in FPGAs and in ASICs, are lastly discussed.

2.1 Wire Scaling

Semiconductor manufacturing process advancements have allowed the manufacturing of deep sub-micron features on integrated circuits. With smaller feature sizes, IC designers can implement current designs in a smaller footprint and achieve gate delay reductions, allowing for the addition of new features while maintaining the original die size. As gate performance increases with scaling, only some wires will scale, causing a mismatch in scaling performance. The following discussion presents a basic overview of wire scaling issues.

Local wires are those used to interconnect gates at the lowest logical primitive level. These wires are found on the first several metal layers of the die, and tend to be small in width and pitch (center to center distance between two wires). Since gates get smaller with scaling, the local wires between them get shorter, and therefore roughly scale with gates. On the other hand, global wires, those spanning across the length of the chip, will not scale with gates [3]; these wires tend to remain the same physical length while the density of logic that it spans increases. Therefore the delay of the global wire increases relative to gate delay under scaling. In addition, Rent's Rule [4] shows that the number of connections outside of an area of logic is a function of logic density. Therefore, more global wires will be required to connect to other areas of the IC. Global wires are typically used for clock distribution and system level interconnects, and can be typically found at the higher metal layers of a die.

In comparison to local wires, global wires must be implemented with larger pitch and wider metal. Larger wire pitch is important because it reduces the capacitive coupling and crosstalk between wires; the closer wires are together, the more susceptible they are to coupling and crosstalk noise. The crosstalk and noise scaling of long wires is discussed in [3]. Wider metal is important because cross-section size and length immediately affect wire resistance, and hence RC delay of a wire. Resistance (R) is inversely proportional to cross-section and grows quadratically with wire length [3], making unbuffered global wires undesirable. In addition, unbuffered global wires are predicted to double in delay relative to gate delay every technological progression [3]. Repeaters, typically implemented with inverters, can be used to segment a global wire into a set of smaller wires. The repeaters buffer the wire, resulting in linear delay growth in relation to wire length.

The 2003 International Technology Roadmap of Semiconductors (ITRS) report offers a prediction of gate and wire performance scaling [5]. A graphic summary of their scaling prediction is shown in Figure 2.1. It is clearly shown that wires on metal layer 1 (local wires) will roughly follow the delay of gates while global wires will not, even with intermediate repeaters. Gate delay is expressed as fan out 4 or FO4, meaning the delay through a circuit consisting of an inverter driving four identical copies of itself. F04 is a useful metric because many CMOS logic circuits can be measured as having a delay multiple of FO4, which can hold true regardless of technology, temperature, and voltage.

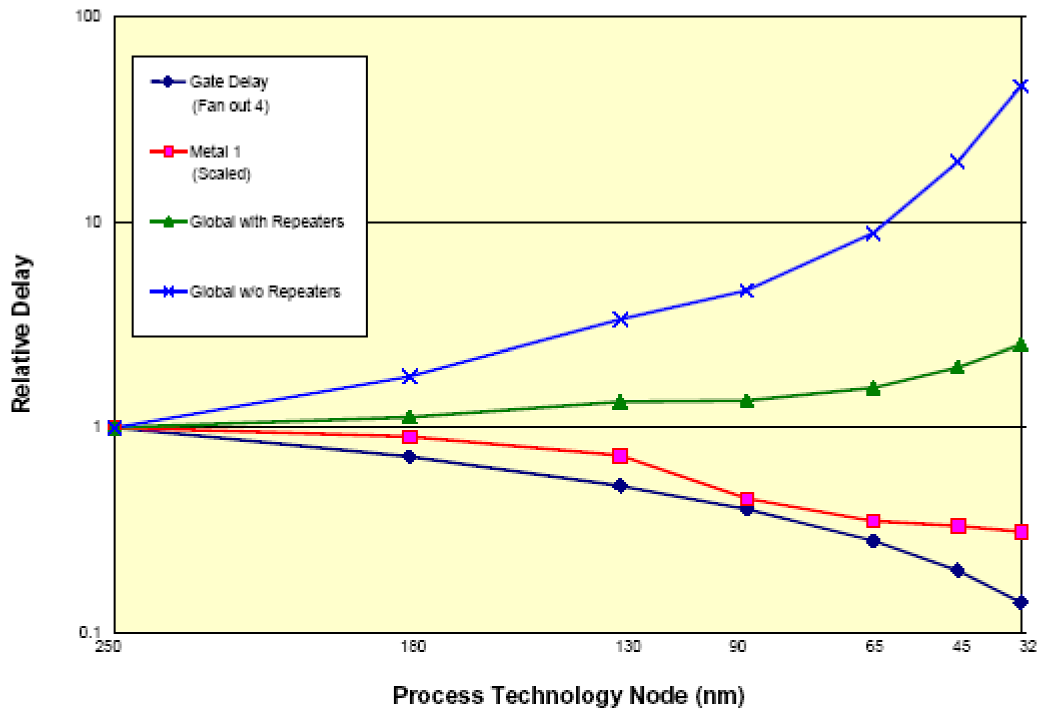

Figure 2.1 - 2003 ITRS scaling comparison

Reprinted with permission from [5]

As technology processes progress, global wire communication must occur over multiple clock cycles, where a conservative maximum clock period is approximately 16 FO4s [3]. In order to minimize the affects of global wire delay, register pipelines can be inserted to keep ICs running near maximum speed. Another method to reduce global wire delay is to use low-swing voltage techniques, such as those described in [6]. Others have proposed IC architectural designs that minimize the use of global wires to maintain performance. One such methodology is network-on-chip, which is discussed in Section 2.4.2.

2.2 FPGA Architecture

FPGAs are an excellent solution for deploying custom logic in low to medium quantities, and have a time-to-market advantage over ASICs (application specific integrated circuits). SRAM-based FPGAs have a further advantage over ASICs in that they are reprogrammable, allowing for

rapid system prototyping and field upgradability. The following section will discuss the architectures and overheads found in current FPGAs.

2.2.1 Architectures

Modern FPGAs typically consist of a matrix of configurable logic blocks (CLB), I/O blocks (IOB), RAM, and dedicated multipliers interconnected through a network of programmable switches and wires. A CLB is the building block of user logic, consisting of lookup tables (LUTs), flip-flops, and arithmetic carry logic. IOBs can be configured to operate in a variety of signaling standards. Dedicated multipliers help FPGAs compete with the computational performance of digital signal processors (DSPs). As an example, the architecture of a Xilinx Virtex-II [8] is shown in Figure 2.2 . Some FPGAs, such as the Xilinx Virtex-II Pro [9], may also integrate microprocessors and high-speed serial controllers, furthering FPGAs' push as a viable solution for system-on-chip (SoC) applications.

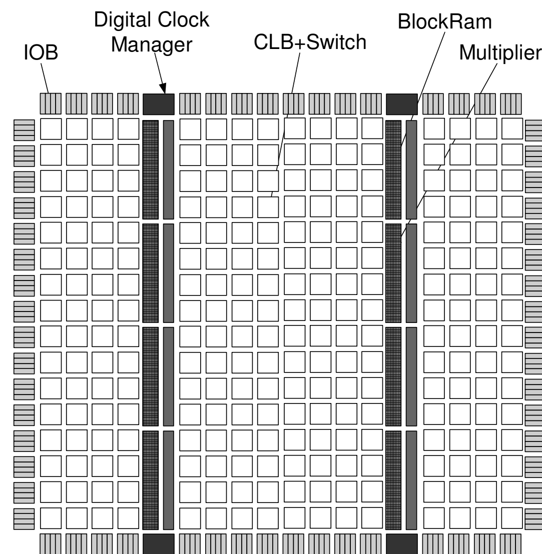

Figure 2.2 - Xilinx Virtex-II architecture

FPGA routing architectures are designed to offer a wide variety of routing paths, a necessity in mapping general logic designs. Global clock distribution and clock controllers allow for low-skew, configurable clocks. In addition, the switches and wires of the routing architecture are

designed to minimize delays and fan-out loading effects. The following discusses the routing architectures used in commercial FPGAs.

The island routing structure is used in many Xilinx FPGAs, such as Virtex [10], Virtex-II, and Virtex-II Pro. The routing wires of the Virtex-II [8] are shown in Figure 2.3. Each CLB is connected to an individual interconnection switch. Double, hex, and long wires are offered in both axes of the FPGA, and are connected through interconnection switches. Directs connect to each of the eight adjacent switches. Doubles connect to the first and second neighboring switch. Hexes connect to every 3rd and 6th distant switch. Longs run the full length of the programmable fabric, and connect to approximately every 6th switch. Carry chain connections between CLBs bypass interconnection switches. Direct, double and hex wires are heavily used in implementing logical functional blocks [7], while long wires, are mostly used to interconnect at the system level.

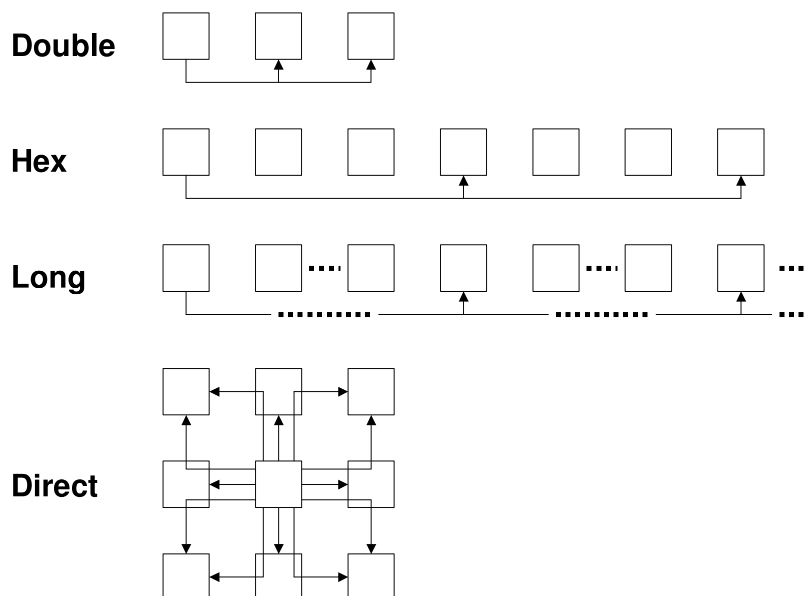

Figure 2.3 - Xilinx Virtex-II switch interconnections

Another approach to routing structure is taken by Altera FPGAs, such as the Stratix-II [11] and Stratix [12]. The Stratix-II routing scheme consists of R4, R24, C4 and C16 wires. “R” wires run horizontally, while “C” wires run vertically. Wire numbers represent the number of

interconnections that are skipped in that direction. For example, C16 represents a wire that connects to an interconnection switch 16 switches away in a vertical direction. Similar to the Virtex-II, each CLB interfaces with an individual interconnection switch. In addition, Stratix-II offers direct connections between adjacent CLBs for carry chains, bypassing interconnection switches in both the vertical and horizontal axis.

2.2.2 Interconnection and Logic Overhead

In general, FPGA performance overheads can be divided into two categories: interconnection and logic. As mentioned in Chapter 1, a predominant source of FPGA design delays is interconnection overhead, where interconnection delay generally contributes to 70% of total design delay. Logic overheads are seen in the additional gates necessary to configure logical functions through configuration memory. These additional gates cause a delay in functional speed and increase integrated circuit (IC) area. An FPGA can require up to twenty gates to accomplish the equivalent function implemented by one gate on an ASIC, with configuration overheads accounting for 70% of the transistors on an FPGA [1].

Besides performance, another FPGA architectural concern is high power consumption, limiting FPGA use in portable applications. Power consumption analysis can be separated into two categories static and dynamic. Static power refers to conditions when an IC is powered, but logic remains idle, lacking switching activity. Due to the CMOS nature of SRAM-based FPGAs, static power solely consists of leakage current, contributing to 10-20% total FPGA power consumption [7]. Dynamic power, the predominant source of total power dissipation in an SRAM-based FPGA, refers to the power consumed during switching activity. Interconnection wiring contributes 50-70% of dynamic power, while CLBs, clocking resources, and IOBs contribute roughly an equal amount of the remaining dynamic power [7]. In addition, configurable interconnection wiring contributes up to 80% of the total FPGA wiring.

2.3 FPGA Scaling

FPGA architecture performance and density have consistently improved over time, a trend demonstrated in [2]. The following section discusses the potential performance scalability issues of modern FPGA architecture; power scalability is beyond the scope of this thesis.

As feature size decreases, transistors become smaller and faster, leading us to FPGAs with an increasing number of faster CLBs. With scaling, additional logic could be packed into a CLB, since wires dominate FPGA IC area. On the other hand, the scaling of some wires becomes a problem. Local wires providing direct connections between neighboring CLBs will scale with transistor performance. Conversely, wires that provide global and long routing will become bottlenecks to performance increases in future sub-micron FPGAs, where delay will exceed multiple clock cycles. These wires do not scale because they typically run the full length of the IC, regardless of the technology used. To compound the problem, the configurable interconnection switches introduce delays and wires may have unused loads. To better understand the impact of long wire scaling in FPGAs, let us look at various FPGAs within the same family. By examining family members, we can see how long wire delays expand when chip density grows while the architecture stays constant.

The Xilinx Virtex-II family was chosen for this exploration due to die information availability. Table 2.1 was compiled using XC2V1000 die information provided in [13]. After calculating the approximate dimensions of a single CLB and an IOB array, die and core dimensions of other FPGAs within the Virtex-II family were extrapolated using known CLB matrix sizes given by [8]. While these extrapolations do not account for multiplier and BRAM width, the use of CLB matrix sizes is good enough to approximate core and die sizes. It was also safe to assume that the IOB array ring consumes the same cross-sectional area among family members. Table 2.2 lists the long wire delays of several members of the Virtex-II family. In addition, Table 2.2 includes dimension and delay ratios when using the XC2V1000 part as a reference. These delays were obtained from a simple FPGA user design that isolated one long wire. Xilinx *trace* was then used to extract the delay of the net associated with the long wire. Note that several parts are missing from Table 2.2. This was due to the infeasibility of isolating a long wire in these parts; connecting pins on opposite ends of the chip required additional use of a double or hex wire.

Table 2.1 - Xilinx Virtex-II extrapolated die size

*XC2V1000 data is taken from [13]

FPGA	Die Height (mm)	Die Width (mm)	I/O Pad (mm)	Die Area (mm^2)	Core Height (mm)	Core Width (mm)	Core Area (mm^2)	CLB Rows	CLB Cols	CLB Height (mm)	CLB Width (mm)
*XC2V1000	10.6	9.6	1.2	101.76	8.2	7.2	59.04	40	32	0.205	0.225
XC2V1500	12.24	11	1.2	139.54	9.84	9	88.56	48	40	0.205	0.225
XC2V2000	13.88	13	1.2	183.22	11.48	10.8	123.98	56	48	0.205	0.225
XC2V3000	15.52	15	1.2	232.8	13.12	12.6	165.31	64	56	0.205	0.225
XC2V4000	18.8	19	1.2	349.68	16.4	16.2	265.68	80	72	0.205	0.225
XC2V6000	22.08	22	1.2	490.18	19.68	19.8	389.66	96	88	0.205	0.225
XC2V8000	25.36	26	1.2	654.29	22.96	23.4	537.26	112	104	0.205	0.225

Table 2.2 - Xilinx Virtex-II long wire delay

*XC2V1000 is the reference for long delay and core width ratios

FPGA	Long Delay (ns)	Max Clk. Freq. (MHz)	Core Width (mm)	Long Delay Ratio	Core Width Ratio
*XC2V1000	1.866	536	7.2	1	1
XC2V6000	4.344	230	19.8	2.3	2.75
XC2V8000	5.729	175	23.4	3.1	3.25

Table 2.2 demonstrates that long wire delay increases linearly with chip width within the same family. Linear delay growth hints at the use of repeaters to avoid the quadratic growth of unbuffered, long wire delay. Consider that the Virtex-II is capable of operating at frequencies around 500 MHz, but as the FPGA gets larger, long communication can be limited to as low as 175 MHz. With technology scaling, this problem only gets worse – long wires are growing in delay with respect to logic.

Ultimately, FPGA architecture must evolve. In the future, we might see application specific logical blocks replacing general purpose CLBs [14], such that reductions of configuration memory overheads and local interconnection switch use will lead to greater performance. In addition, new architectures may reduce routability requirements, reducing IC wiring overheads and power consumption. Another change in FPGAs will come in the area of configuration memory, where there is a push for a non-volatile memory that is more power, area, and speed efficient than SRAMs [14]. These architectural changes not only improve performance and power, but may increase manufacturing process scalability.

2.4 Wire Emulation

The previous section showed that cross-chip wire communication scaling in FPGAs is of concern. As mentioned before, one way to address this issue is to design an architecture that avoids use of cross-chip interconnects. By minimizing the use of cross-chip wires, we minimize the time spent designing, testing, and verifying them. Wire emulation is one such method of avoidance. While the term "wire emulation" has been used in other areas, such as networks [15] and FPGA external interconnects [16], this terminology has not been used in the area of IC intra-communication. This thesis refers to wire emulation in the context of IC intra-communication.

Wire emulation can be implemented in a wide variety of ways, but the underlying motivation is to implement wire functionality without using a dedicated wire. For example, some have proposed the use of intra-IC RF communication [17], while others have proposed optical links [18]. This thesis explores ideas that are limited to traditional IC fabrication methods, and therefore limits this investigation to ideas utilizing conventional wires. Two methods that are discussed are FPGA self-reconfiguration and network-on-chip.

2.4.1 FPGA Self-Reconfiguration

Since the goal of this work is to enhance FPGA architecture, it would be prudent to examine previous work on wire emulation within FPGAs. Some SRAM-based FPGAs support active partial reconfiguration, where the contents of configuration memory can be written to while the FPGA user design is operating, allowing parts of the user design to change with minimal downtime. A subset of active partially reconfigurable FPGAs also support self-reconfiguration, where the user design can internally manipulate configuration memory, and therefore modify itself. One wire emulation technique on an FPGA uses self-reconfiguration [19][20].

The motivation behind [19] was to efficiently support runtime reconfiguration, where computation accelerator cores were swapped and replaced frequently in an FPGA. These cores would interface on a common bus to a centralized processor realized on the FPGA. The centralized processor, URISC [19], supports a single instruction – move word. The problem with this approach was that it required runtime routing, where rerouting was required to connect and disconnect swapped cores. Unfortunately, runtime routing is a computationally intensive

problem, usually requiring an external host processor[1]. Even with a host processor, runtime routing is costly, where the overhead time outweighs the benefits of runtime reconfiguration.

The solution posed by the authors of [19] was to not route a bus of wires between the cores and the processor, but to read and write FPGA configuration memory in order to access registers within cores. A prototype was implemented in a Xilinx XC6200 FPGA [20]. The URISC processor memory mapped FPGA configuration memory through the XC6200 FastMap interface, a memory-style interface accessible to the configurable fabric of the XC6200. The URISC operation was directed by a sequence of instructions stored in design RAM. Communication was serialized, but since the standard bus architecture would have been used otherwise, serialization would still have been a limiting performance factor. Configuration memory access time was constant [20], and the speed of this access might have rivaled delays imposed by programmable wires [19]. Additional benefit was seen in the capitalization of existing physical resources that otherwise would have been unused.

2.4.2 Network-on-Chip

Typical SoC implementations use a system bus to interconnect the various cores on the chip. These busses are expensive because they must be connected to involved cores with relatively long wires. Busses do not scale well when adding cores on the bus, because additional capacitive loads and increased latency are introduced. Bus wires are inefficient because only one core can drive it at once, thusly restricting cores to communicate in a serialized process. Moreover, if all but two cores are involved in any bus transfer, the non-involved cores serve as additional capacitive loads on the bus, making for a power inefficient system. Also, bus performance does not scale when migrating to a smaller technology feature size, since long wires do not scale with gate delays. To compound this problem, designers are inclined to add additional cores on the chip to take advantage of new die area when migrating an existing SoC design to a smaller technology process.

---

[1] ADB Routing [21] may enable systems lacking a host to compute run-time routing with acceptable delays.

Some researchers in the area of VLSI and SoC have been developing strategies for Network-on-Chip (NoC) [22][23][24][25][26][27][28][29][30] to reduce the use of global wires, and to alleviate the overheads associated with system busses. As its name implies, NoCs form a communication network over the logic cores of the chip. Implementing networks with shorter, segmented links reduces the need for long wires, and thusly scales with technology process. Links are interconnected with routers and network interfaces. Network interfaces bridge core logic with the NoC. Network interfaces can be designed to create a consistent interface to the network, aiding in core reusability. Network interfaces can also interface to I/O pins of the chip. Cores communicate over the network using packets.

NoC differs from traditional networks in that their limitations are different. Typical networks, such as multiprocessor shared memory systems, inter-chip networks, and LANs, are limited in the number of pins on their packages, and hence the number of wires per link. To compensate, large buffers are implemented in the nodes of these networks to handle the reassembling of smaller width data. The constraints are reversed for NoCs [22]. The number of wires per link is much less restricted, allowing for faster and wider link transfers, but silicon area for buffers becomes a large concern. In addition, power and network reliability become important. Network links are predicted to dominate NoC power dissipation, requiring four times as much power compared to NoC logic [31]. In addition, communication between cores in a chip cannot tolerate errors or dropped packets, because cores are designed with the assumption that communication is error-free when using traditional wires. Another interesting characteristic of NoC is that all network objects (routers and network interfaces), can be synchronized together, due to the small area of the network. Global synchronization allows for routing algorithms that guarantee performance: throughput and latency of network traffic [32][33].

As mentioned before, core intercommunication is realized through packetized data, replacing the dedicated wiring and busses once required. Logical channels can be established on the network to describe a collection of dedicated wires or buses. Wire reuse is an advantage of packet communication over dedicated wiring schemes because multiple logical channels can be transmitted over the same network link. Some dedicated wires can idle for a large percentage of time, and can be seen as an overhead from a layout perspective since these wires may cause layout congestion.

Another advantage of NoCs is that they are scalable. Additional links, routers and network interfaces can be added without changing the electrical properties of an existing NoC. This is especially true when networks have allocated space in a layout. Such is the case in NoCs where logic cores are stuffed into rectangular areas, with network links and routers placed between edges of adjacent logic cores. Network routers and network interfaces are easily replicated. With NoCs using regularly patterned topologies, links between new network objects can be added easily with minimal design effort. Adding links with minimal effort is important because wire design and analysis can be time consuming. If link wires are placed in consistent environments where die stack-up characteristics are predictable, then designing the wire for delay characteristics and noise isolation must be done only once. The opposite is true when connecting cores through long, dedicated wiring, because each wire must be individually designed and analyzed. Timing closure in the design process can be difficult in large VLSI designs because long wires have a tendency to be falsely characterized by design tools [3]. In addition, it is time consuming to fix these issues, and there is more room for error when many unique, long wires are involved. Therefore, link wires in NoC are less costly to design and offer more predictable performance than layouts involving long, dedicated wiring.

NoCs can be designed with a variety of topologies, but the most commonly implemented are those using meshes [28][30][31][32], folded toruses [22], or fat trees [26]. Irregular or custom network topologies can be implemented as well. Meshes are frequently favored, due to its power consumption advantages associated with minimal, consistent routing. NoC links can support large bandwidths because they can be implemented with a large width of wires. Some designs have used up to 300 wires per link [22]. Since power is a major concern, some have described using differential signaling to minimize voltage swing. Differential signaling is power efficient, since V_{dd} is reduced and power is proportional to V_{dd}^2. In addition, wires can travel further distances, but at the cost of double the number of wires. Pseudo differential signaling [22] has been mentioned, which utilizes full V_{dd} swing, but with fewer signal transitions. Throttling network frequency has also been suggested to minimize power. Simulation tools, such as [34][35], have been developed to explore topologies, link size, operating frequency, routing and buffering strategies configurations when considering factors like power, area, and delay for a particular application.

Chapter 3

FPGA Architecture Enhancement

The previous chapter presented the motivation for FPGA architecture enhancement, with focus on addressing chip-length communication scaling concerns. In this chapter, the goals of a proposed FPGA architecture enhancement are defined. Then, a conceptual approach to using wire emulation is introduced. Finally, the impacts of this approach to FPGA IC layout and FPGA user design tools are discussed.

3.1 Goals

This thesis proposes a conceptual approach to FPGA architectural enhancement to address issues concerning cross-chip communication performance scaling. The goals of this enhancement are:

- to provide a conduit that rivals or exceeds performance and manufacturing process scalability of cross-chip communication in existing FPGA technology,
- to potentially reduce FPGA routing congestion, and
- to impose minimal increases to FPGA power consumption and IC area.

Redesigning the existing cross-chip interconnects using traditional dedicated wires would be the first solution that comes to mind. One potential solution would be to implement these interconnects using differential wires, where a low voltage swing signal could be used to minimize RC delays. This would require the number of long interconnects to double and would lead to increased layout congestion. In general, dedicated wire approaches still encounter overheads posed by the FPGA architecture, such as connections to multiple switch loads.

Additional overhead is incurred considering only a portion of interconnects are configured for use in a given FPGA user design. To meet these goals, we consider the use of wire emulation techniques, which may reduce overheads through the reuse of resources.

3.2 Concept

Consider the analogy that an FPGA is like a city, where city streets, arranged in 2-D mesh, surround blocks of buildings. The FPGA programmable interconnects are the streets, programmable switches are intersections, and CLBs are buildings. Now consider that people are computational data, where people must travel along paths of city streets between buildings. The quickest method of travel between different areas of the city is by motor vehicle. The speed at which people travel is relative to the mobility and passenger capacity of their vehicles. Therefore, roadways with more lanes can handle more people simultaneously. That is why we have highways and major roads within our cities to minimize the number of traffic intersections incurred in a trip. This is analogous to the various wire lengths offered in FPGA programmable interconnections.

Unfortunately, with increasing population in this city, the time it takes to get from one side of the city to the other is increasing, with more vehicles on the road, and people willing to make longer commutes. With motor vehicle travel not as attractive due to traffic and demand, consider that we add another dimension to the city transportation resources – subway travel. Subway stations are spread throughout the city, allowing its use as an alternative to roads. With subways, larger quantities of people can be packed into a subway train, and subway trains can travel faster between subway stations than vehicles traveling between the same points, due to a reduced number of vehicles and intersections. Therefore, subways offer transportation that is less interrupted and denser than vehicles traveling on city streets. It is proposed in this thesis that wire emulation enhances FPGA cross-chip communications, much like the subway system improves city transportation, using minimally interrupted, fast communication links with increased data transfer capacity.

A conceptual view of a wire emulation system for FPGA architecture is shown in Figure 3.1. This system contains a collection of transports and an interconnection routing system. Transports are distributed throughout the FPGA and are accessible by programmable local logic.

Information transferred between transports can substitute cross-chip FPGA interconnection. The implementation of transports and the interconnection routing system is wide open, and can be independent of FPGA architecture. Therefore, the next section will describe characteristics of a desirable wire emulation system.

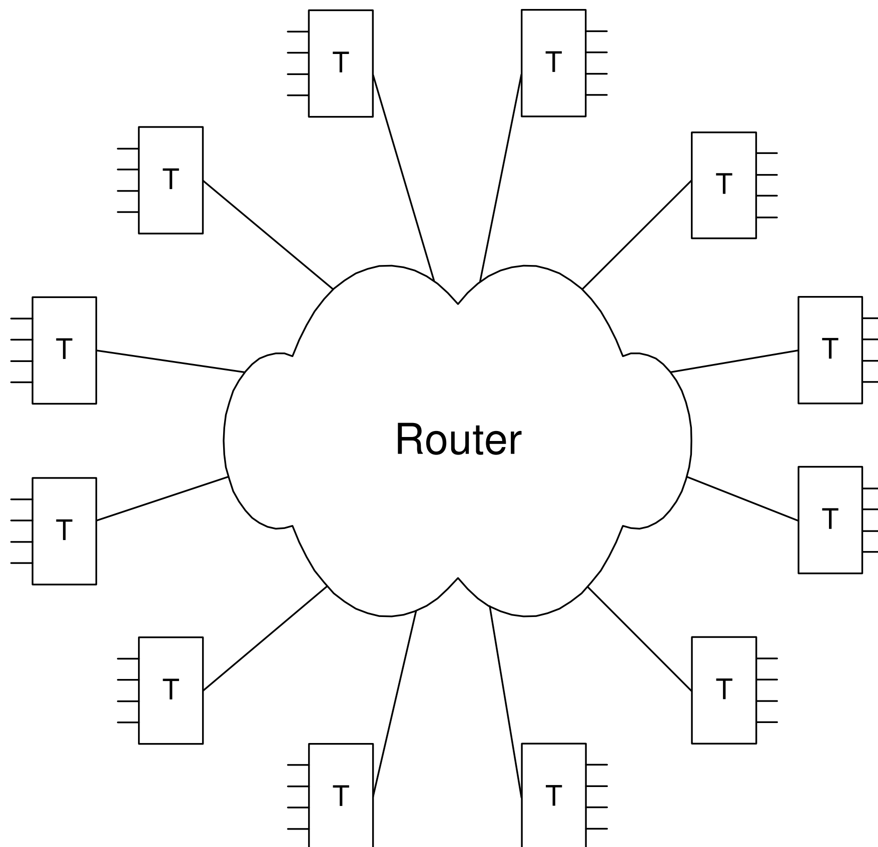

Figure 3.1 - Conceptual wire emulation system

3.3 Methodology

Wire emulation should stay true to wire behavior. From a designer's view, IC wires have deterministic delays caused by RC physics. In addition, IC wires are susceptible to noise, but should be engineered to prevent logic errors. Wire emulation can be implemented in a wide variety of ways. For this reason, it is important to discuss the desired characteristics of a wire emulation solution for FPGAs. The characteristics are as follows:

- Easy to use
- Guaranteed performance: throughput and latency
- Reliable, error-free operation
- Optimized IC implementation
- Testable

Ease of use is an important requirement because a designer will not want to use or develop for such a solution that is difficult to use. This can be considered at various levels. At the highest design level, the solution should be integrated into the FPGA design tools, and maintain the level of automation that designers expect. The use of the solution should be abstracted from the designer, unless the designer wishes to have lower-level control of FPGA resources. At the FPGA level, the solution should offer a natural, conventional interface, one that maintains generality. By keeping the interface simple, FPGA CAD tool developers will be able to integrate the solution with minimal restructuring of existing tools. One such interface could sample local wire state, and transport these states to another interface, which would then connect to another set of local wires. Such a solution provides a conduit or bridge for communications across the FPGA. In addition, high on-chip availability to this solution is desired, so that solution accessibility is not limited by logic placement.

Guaranteed performance is needed. Without guaranteed performance, the simple assumption that wires have deterministic delays will prevent a design from meeting timing requirements. In the wire emulation schemes that have been mentioned in Chapter 2, there is some form of resource reuse. Guaranteed performance can be realized if shared resources have deterministic behavior. For example, the FPGA self-reconfiguration approach used instructions to schedule the use of its shared transfer mechanism. Another example is with NoCs, where guaranteed throughput and packet latency are realized using virtual connections. Virtual connections are maintained using schedules stored in routers and network interfaces. Without virtual connections, packets would be free to enter the network randomly, which would require routers to dynamically handle contention resolution. This results in a non-deterministic time that a packet will reside in a network, and therefore non-deterministic communication latency.

Reliable, error-free operation is necessary, since wires are engineered to prevent logical errors. Therefore, error detection and correction should be considered in a wire emulation scheme. But, the policy of error detection is difficult. Dropping errors is undesirable, which would be

equivalent to a temporary open circuit in the wire connection. Since packets are used in NoC, packet retransmission can be used in the detection of an error. Retransmission is costly since the retransmission causes an undesirable latency. Therefore, error correction codes should be used if the wire emulation scheme is susceptible to errors, with the errors corrected at interfaces.

The wire emulation scheme investigated here can operate independent of FPGA operational environments, allowing the scheme to be implemented with different signaling techniques, logic levels, clock domains and techniques. Additionally, the wire emulation scheme can operate without the overheads imposed by programmable wires within the FPGA fabric, such as unused loads and extraneous, non-configured interconnects. By fully utilizing VLSI design techniques, this wire emulation scheme could operate at a level that improves chip-length communication, and potentially reduce the penalties of manufacturing process scaling. The IC implementation should be scalable.

This enhancement to the FPGA should be testable, both at the IC level and at the user design level. A majority of FPGAs already include JTAG boundary scan capabilities, which allow users to examine and control features within an IC. The boundary scan chain should be extended to include control over this architectural enhancement, so that this circuitry can be verified at the IC manufacturer. In the case of a system resembling an NoC, network link continuity could be tested using router pin diagnostics. Performance of the NoC could be evaluated through the use of test bench network configurations that would fully exercise the network, enabling operational verification. These test benches could also utilize the FPGA configurable fabric to measure and verify timing requirements of the NoC.

Chapter 4

Wire Emulation Prototype

While the previous chapter discussed the qualities and effects of wire emulation technology designed for FPGAs, this chapter explores potential wire emulation technology in existing FPGAs. This exploration identifies features in existing FPGAs that can be brought together to demonstrate concepts of wire emulation technology. In addition, a prototype demonstration is described. Results and analysis of this prototype are discussed in Chapter 5.

4.1 Overview

Current FPGA technology can be used to implement concepts of wire emulation. One such strategy uses self-reconfiguration, which was presented in Chapter 2 and as well as in [19][20]. Utilizing self-reconfiguration for wire emulation is a double-edge sword. On the positive side, it enables the reuse of FPGA resources, and thus reduces the need for additional IC area. On the negative side, an implementation of self-reconfiguration depends on FPGA programmable fabric, and therefore relies on resources with performance and area overheads. Chapter 3 discussed that a wire emulation system using FPGA-independent circuitry can achieve good performance and area efficiency. While self-reconfiguration relies on FPGA resources, components implementing self-reconfiguration could be improved upon or adapted to provide a more ideal solution, one that meets the goals and characteristics presented in Chapter 3. It is expected that newer strategies can be developed after understanding the shortcomings of existing technology.

Self-reconfiguration has been explored on the Xilinx Virtex-II family of FPGAs in [36][37][38][39]. By developing a wire emulation prototype utilizing self-reconfiguration on the

Virtex-II FPGA, it is hoped that concepts of reuse, ease of use, and guaranteed performance can be demonstrated. It is expected that performance will be limited by serialization, as seen in [19][20], but it does allow one to investigate the necessary improvements needed to become a viable solution. The next section will discuss a prototype system that investigates wire emulation on the Xilinx Virtex-II.

4.2 Concept

The goal of this prototype, as shown in Figure 4.1, is to demonstrate wire emulation, where self-reconfiguration is used to realize the communication between regions of FPGA programmable fabric. This prototype will demonstrate how two or more cores can communicate within a Virtex-II FPGA without FPGA programmable interconnects between them. These cores will communicate with each other using transport modules implemented in FPGA programmable logic. Transport modules also lack interconnections between each other. Communication channels are realized with data transfer between transport modules. These data transfers are accomplished using self-reconfiguration, which is a combination of configuration memory readback and partial-reconfiguration. In addition, this prototype attempts to minimize self-reconfiguration performance overheads in Xilinx Virtex-II.

An application may contain a variety of cores, each having communication requirements between one another. For the purposes of demonstration, this prototype focuses on the interaction between two cores. Furthermore, this prototype emulates one-way communication, characteristic of typical wire usage; an exception to this occurs when a wire is connected to multiple drivers, such as in the case of a bus. This prototype could be expanded to support additional communication channels, which would be required in a real application, where each core may require one or more send and receive transport modules.

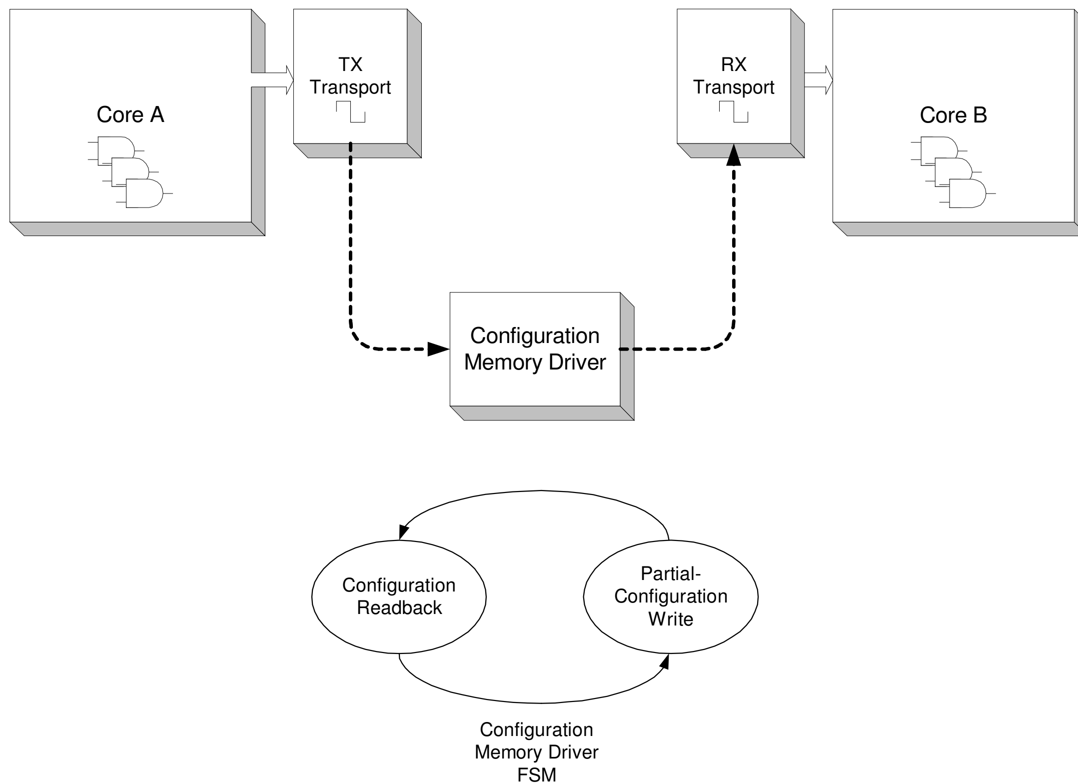

Figure 4.1 - Prototype concept

The two major components in the prototype are the transports and the configuration memory driver. First, the transport has a RAM-like interface and is implemented with memory buffers. A memory buffer has a standard interface, including ports for address, data, and read/write control. While a memory buffer does not behave like a set of wires, a memory buffer could serve as transport region for data at a higher abstraction layer, i.e. packets or messages. Another reason for using memory buffers is that their mapping to configuration memory allows for minimal overhead in addressing and transferring of Virtex-II configuration memory. More details on Virtex-II configuration memory mapping are discussed in Section 4.2.1. Registers [19][20] would have provided a better transport interface, since it naturally maps to wires, but Virtex-II does not allow configuration memory writes to modify register contents. Alternatively, the memory buffer could be divided into many small memory buffers, with individual buffers acting as registers. Such a scheme is discussed in Section 4.3.2.3. A summary of transport interfaces is shown in Figure 4.2.

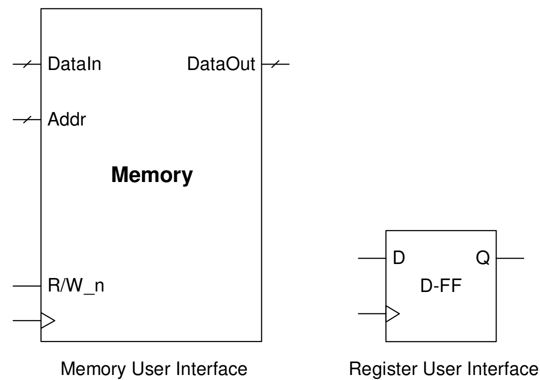

Figure 4.2 - Transport user interface options

Second, the configuration memory driver is implemented with a finite state machine (FSM), as shown in Figure 4.1, that drives the Virtex-II internal configuration access port (ICAP), a version of the external Virtex-II Select Map configuration memory interface that is accessible to the FPGA programmable fabric. The configuration memory driver FSM continuously copies configuration memory mapped to the TX transport to configuration memory mapped to the RX transport. Configuration memory copy is implemented by performing readback and partial-reconfiguration writes through ICAP. Other self-reconfiguration applications utilizing ICAP have been described in [36][37][38]. Further discussion on ICAP is found in Section 4.2.2.

4.2.1 Configuration Memory

The configuration memory of a Xilinx Virtex-II is organized into frames: major and minor [42]. A minor frame spans the length of a CLB column; therefore, the size of a minor frame is FPGA dependent and grows with FPGA size. Major frames consist of a collection of minor frames. Accessing a minor frame requires the address of both the major and minor frame. The smallest accessible unit of configuration memory is a minor frame. A configuration memory minor frame write takes one configuration clock cycle to commit. A majority of frames contain programmable fabric and IOB configuration information, while a few frames only contain IOB configuration information (frames on the left and right edges of the FPGA).

4.2.2 ICAP

The internal configuration access port (ICAP) is a bridge between programmable fabric and configuration memory. ICAP is easily instantiated in an HDL language, and is understood by the Xilinx design tools as a component integrated into the Virtex-II. As mentioned before, the ICAP behaves like the Virtex-II Select Map configuration memory interface. The ICAP consists of two 8-bit data interfaces (one for input and the other for output), a read/write control signal, a chip enable signal, and handshaking signal (busy), and a clock. ICAP can operate up to 50 MHz without handshaking. The ICAP, as shown in Figure 4.3, provides access to the FPGA configuration manager [40][41][42], an FSM that allows access to FPGA configuration memory and configuration parameters. In fact, the configuration bitstreams used to load user designs onto the Virtex-II consists of a sequence of FPGA configuration manager transactions. Details on the FPGA configuration manager communication protocol are described in [41][42][37][38]. Overall, the process of accessing configuration memory is simple, but time-consuming, since the configuration manager interface was designed to minimize external pin count.

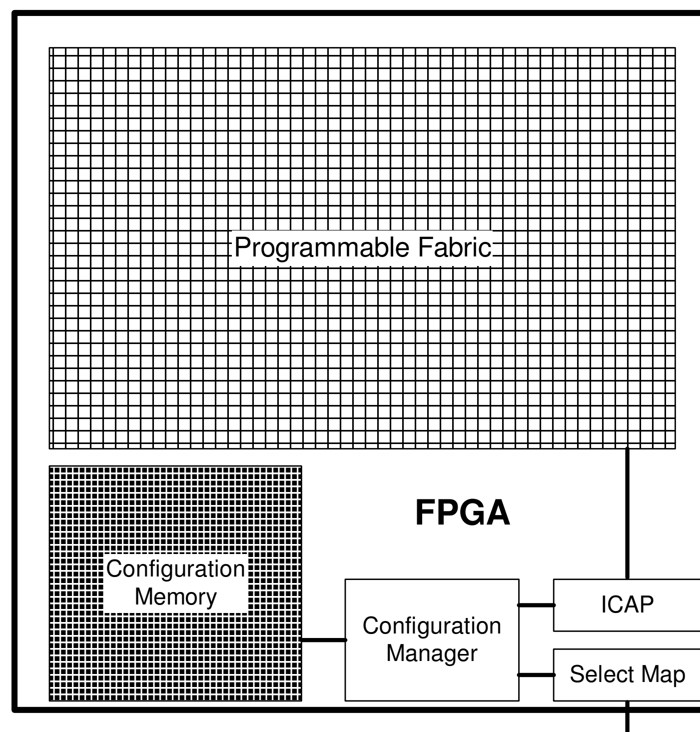

Figure 4.3 - Virtex-II configuration architecture

4.3 System Design and Implementation

The prototype system design is shown in Figure 4.4 and was targeted to run on a Xilinx Virtex-II XC2V1000 FPGA. This design can be modified to work on other Virtex-II family members. VHDL was used to implement the prototype. All components except the ICAP are implemented on the FPGA programmable fabric. A dual-port BlockRAM is used as temporary buffer during configuration data transfers. To verify correct operation, several debugging components were included to provide feedback to a PC via serial cable at 57600 BAUD. The following subsections discuss the design and implementation of several key components, including the system FSM, transports, and ICAP drivers used for accessing configuration memory.

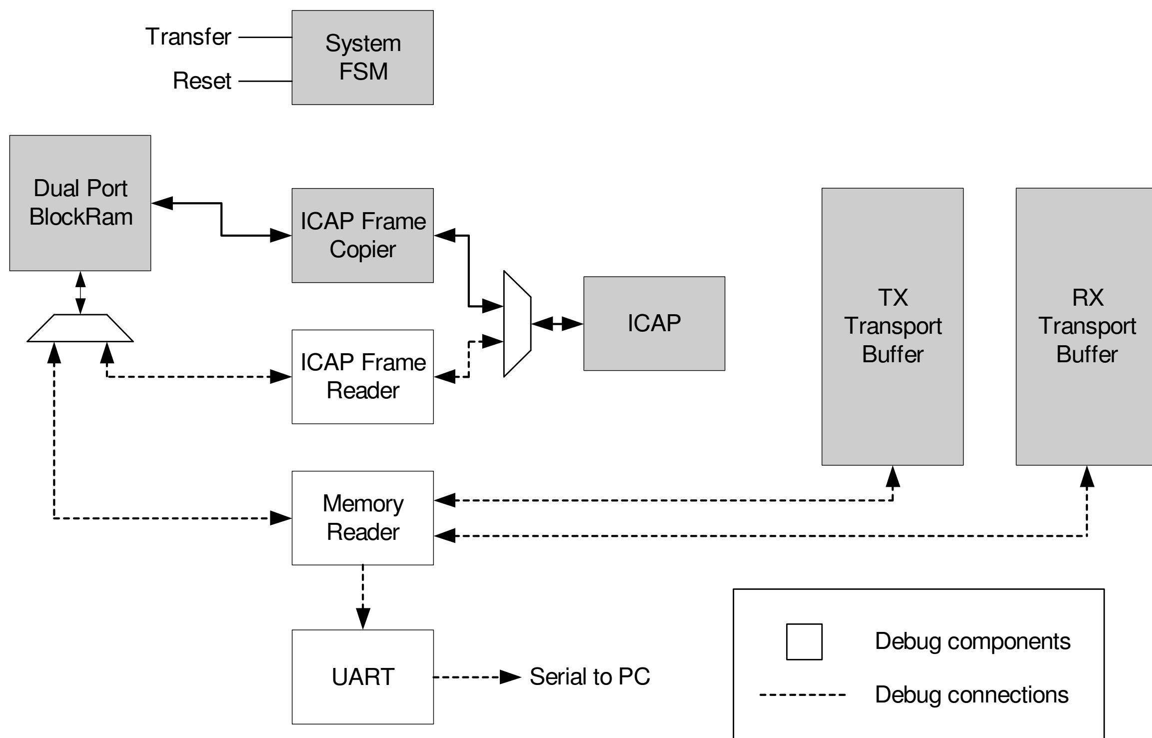

Figure 4.4 - Prototype system design

FSM connections not shown for clarity.

4.3.1 System FSM

The system FSM controls the prototype operation, implementing the configuration memory driver FSM shown in Figure 4.1, and providing diagnostic data necessary for verification by a host PC. The following list describes the states of the system FSM:

1. Readback RX Transport Buffer configuration memory frames. Send configuration frames to PC through UART.
2. Readback TX Transport Buffer configuration memory frames. Send configuration frames to PC through UART.
3. Copy TX Transport Buffer to RX Transport Buffer using ICAP Frame Copier.
4. Readback RX Transport Buffer configuration memory frames. Send configuration frames to PC through UART.
5. Send contents of TX Transport Buffer to PC through UART, reading the transport buffer through its native user interface.
6. Send contents of RX Transport Buffer to PC through UART, reading the transport buffer through its native user interface.
7. Increment TX Transport Buffer contents.

A full cycle of this FSM is performed when the “Transfer” signal is asserted. Many of the states in the system FSM are for debugging purposes only, most of which involve sending data to the PC for verification. At the end of a full FSM cycle, a program running on the connected PC verifies that the contents of RX and TX Transport Buffers match. This match is performed at the configuration frame level and at the logical data level (data accessed through native user interface). Since this prototype involves only one pair of transport buffers, the addresses of involved configuration memory frames were hard-coded into the design. In an application environment, the ICAP Frame Copier would be scheduled to continually run with various source and destination configuration frame addresses, servicing multiple pairs of transport buffers in a serialized fashion.

4.3.2 Transports

As mentioned in Section 1.1, transports are implemented with memory buffers, with transport interfaces modeling a simple RAM. This memory buffer could have been implemented with either BlockRAMs or distributed RAM/ROM blocks. While BlockRAMs are denser than distributed RAM/ROM blocks and do not utilize general programmable fabric, they are unreliable when being modified through self-reconfiguration; therefore, distributed RAM/ROM blocks must be used.

4.3.2.1 Configuration Memory Mapping

While a RAM interface is non-ideal for a general wire emulation solution, a memory buffer comprised of Virtex-II distributed RAM/ROM blocks efficiently map to configuration memory. This property was discovered through experimentation, since the configuration memory mappings are unpublished for most programmable fabric resources of the Virtex-II; Virtex LUT and BlockRAM configuration memory mappings can be found in [42], but these mappings differ from Virtex-II. Using the logic allocation (.ll) output file of the Xilinx *bitgen* tool, it was determined that contents of CLB LUTs, the storage units of distributed RAM/ROM, are stored in adjacently addressed minor frames. An .ll file lists the used resource configuration bits and their frame addresses, frame offsets, and their overall offsets in the configuration bitstream. In addition, it was determined that all but the first and last 96 bits of one of the meaningful bits of these configuration frames was devoted to LUT contents. For reference, a Xilinx Virtex-II XC2V1000 has a minor frame size of 3392 bits [8]. These leading and ending 96 bits of the configuration frame are probably associated with the IOB configurations, since IOBs are at the top and bottom of CLB column. To efficiently copy frames, it is preferred to transfer frames without masking bits, an extra step to prevent the overwriting of unassociated configuration bits. Therefore, IOBs in memory buffer columns should be unused, unless the corruption of these bits is safe, or if memory buffers have identical IOB configurations.

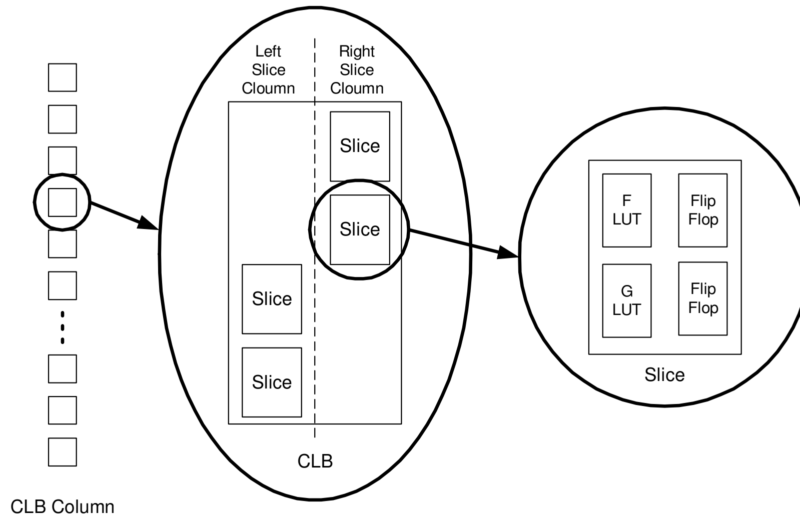

Figure 4.5 - Virtex-II CLB and slice structure

To have a better understanding of how LUTs are organized in configuration memory frames, one must understand the contents of a CLB, as shown in Figure 4.5. CLBs consist of four slices, with each slice containing two 4-input, 1-output LUTs (named G and F), two flip-flops (one for each LUT), and carry logic. In addition, slices within a CLB are organized into two columns, with two slices within each column. Each 4-input, 1-output LUT requires 16-bits of data, since there are $2^4 = 16$ possible input combinations. These LUTs can also be configured as distributed RAM, distributed ROM, or a 16-bit shift register. Since there are two LUTs per slice, each slice contains 2 x 16 bits of data that must be stored in configuration memory. With slices divided into two columns within a CLB, the LUT data from the left slices map to one configuration frame while the LUTs from the right slices map to a second configuration frame. In the LUT portion of a configuration frame, LUTs are packed together, with the highest logical slice address first. LUT data is not perfectly packed; there is an 8-bit gap of 0's between LUTs within the same slice. It was also experimentally verified that LUT contents are stored inverted in configuration memory, which was the case in Virtex [42]. Verification was done by instantiating distributed ROMs with initialized data, and comparing configuration readback results of those LUTs with the contents displayed in the Xilinx *FPGA Editor* tool.

4.3.2.2 Memory Buffer Design

The previous subsection discusses how a column of distributed RAM/ROM efficiently uses configuration memory. Since ease of use is a primary a goal, a full column of RAM/ROM was assembled to form a unified, single-ported, simple memory buffer. RAM/ROM blocks could be left independent to support multiple smaller transports, a strategy that is discussed in the next section.

When designing a memory buffer, there are two aspects to consider: access type (read-only, write-only, or read/write) and organization, driven by resulting data width and address depth. When considering access type, a read/write memory device will be suitable for all situations. Read-only and write-only memory devices could be implemented with less logic overhead, but are only applicable to particular situations. Since communication channels are one-way, a transport interface only needs to support either read-only or write-only. For completeness, all access types were implemented; the read-only buffer was implemented using ROMs while the write-only and read/write buffers used RAMs. In the end, RAMs and ROMs occupy the same number of LUTs for storage; only the need for a R/W control signal and data output port varies programmable resource usage.

Memory organization is the second aspect to consider. Distributed RAM/ROM blocks are available with various data and address widths, but all are implemented with one or more LUTs. For the purposes of this prototype, an 8-bit data width was used. The address width was determined after organizing the distributed memory. The easiest method of instantiating a full CLB column of distributed memory is to instantiate an array of 128x1 (128 addresses of 1 bit data) and 32x4 (32 addresses of 4 bit data) memory blocks, each of which consume an entire CLB. Since the target FPGA has forty CLBs in a column [8], the memory buffer was implemented with 32 128x1s and 6 32x4s to create a unified 608x8 memory buffer with a 10-bit address interface. The last two CLBs in the column were left unused so that they could be used for future needs, such as the implementation of semaphores [42].

4.3.2.3 Alternate Memory Buffer Design

The previous section describes the organization of a large, unified memory buffer that utilizes a CLB column. An alternative would have many small memory buffer blocks, and hardwire them to emulate registers, allowing for a more natural wire emulation interface. This strategy was not implemented, but is described here for future implementation.

Register emulation could be accomplished by declaring each LUT as a 16x1 RAM, tying the four address pins low, and supplying a clock. To emulate a writeable register, shown in Figure 4.6, the write-enable pin is hardwired for assertion and the DIN pin is used as the wire input. To emulate a readable register, also shown in Figure 4.6, the write-enable pin is hardwired for de-assertion and the DOUT pin is used as a wire output. Since the address pins are tied to a fixed address, the LUTs are utilized at 1/16th efficiency.

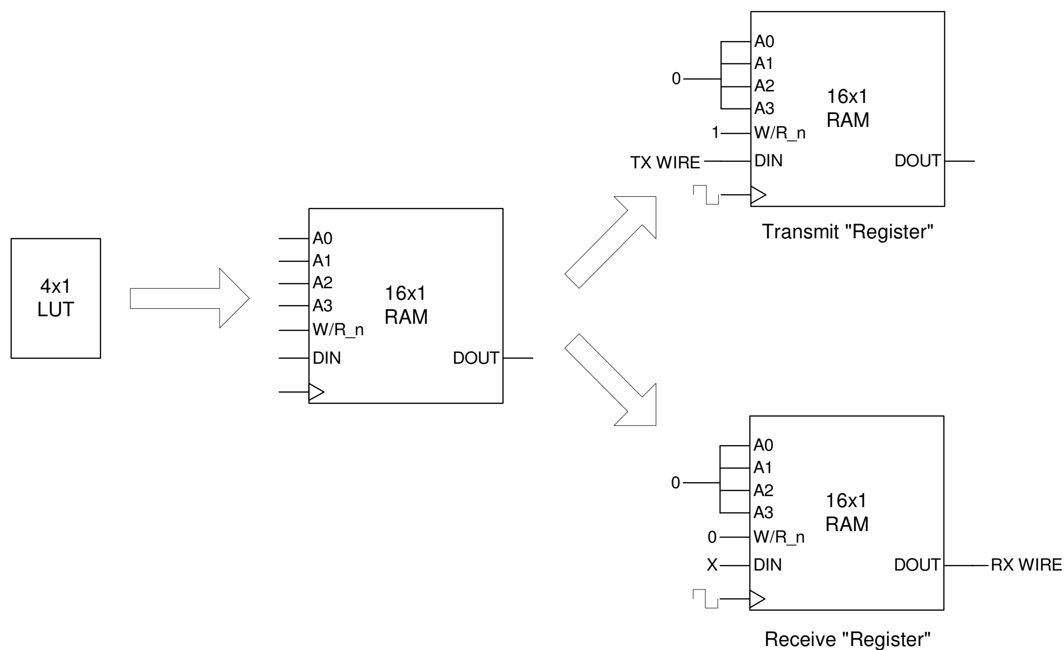

Figure 4.6 - Register emulation using a single LUT

With this strategy, each LUT can act as a transmitter or receiver interface in a wire emulation system. With the target FPGA having 40 CLBs in a column, with four slices in a CLB and two LUTs in a slice, a total of 40 x 4 x 2 = 320 wires can be emulated through the transfer of two consecutive configuration minor frames, assuming all wires are either all transmitters or all

receivers. If a variety of transmitters and receivers is needed, the CLB column usage can be split such that one slice column is used for transmitting, while the other column is used for receiving. This method allows for 160 transmitters and 160 receivers within a CLB column. Wire emulation transfer with this split scheme would require one configuration minor frame read/write for each slice column.

4.3.2.4 Consistent FPGA Implementation

The most challenging aspect of implementing the memory buffers was maintaining consistent placement, packing and pin assignment in the FPGA. FPGA design tools have the freedom to vary CLB packing and pin assignments between instantiations of the same distributed memory component, as long as the design remains logically correct. Consistent routing is not required, since routing takes place after locking the placement, packing and pin assignment of each distributed memory block. In addition, copying configuration frames associated with LUT contents does not affect programmable routing. Without consistency, transferred information would be corrupted when accessed through memory buffer user interfaces. Several methods were originally attempted, such as using hard macros and other user constraint directives, but none were found to be a viable solution. A successful methodology is now presented.

First, both the synthesis and FPGA design tools were given parameters to keep unconnected components. Since these memory buffers may have unconnected outputs and may not influence external FPGA pin outputs, the synthesis and FPGA design tools would have optimized the memory buffers out of the design.

To obtain consistent placement, a user constraint file (.ucf) was used to lock slice locations of each instantiated distributed memory block. In addition, the result of the FPGA design tool mapper needed to be modified due to inconsistent CLB packing. Inconsistencies in CLB packing were seen in 32x4 memory blocks, where there are $4! = 24$ packing possibilities. As a result, slice-data bit relationships were inconsistent. A 128x1 memory block has only one packing possibility, and therefore must not require correction. To fix packing inconsistencies, the mapped design was translated into XDL, a text representation of the physical design, using the Xilinx *xdl* utility. A perl script processed the XDL design to correct the packing problems. The corrected

design was then converted backing into the native format of the design tool, .ncd, and the design flow was continued.

The FPGA design tools have the freedom to assign address pins of distributed memory blocks to optimize routability. Unfortunately, this optimization leads to inconsistent pin assignments, affecting the consistency of address bit assignments for each distributed memory block. To fix inconsistent pin assignments, the FPGA design tools were halted after placement. Placement results were then converted to XDL, corrected with a perl script, and converted back to the native format of the design tool. Afterwards, the FPGA design tools were allowed to route the design and produce a configuration bitstream.

4.3.3 ICAP Frame Reader

The ICAP Frame Reader can request the readback of one or more configuration memory frames, and store the result into a user memory. For this prototype, two consecutive configuration minor frames were read to retrieve the contents of a transport buffer. The FPGA configuration FSM can read consecutive minor frames by specifying the starting address of the first minor frame, and requesting $n + 1$ frames worth of data; the pad frame must be read before configuration frame data and must be accounted for in the number of DWORDs (32-bits) to readback. The XC2V1000 FPGA frame size is 3392 bits or 106 DWORDs, so the prototype must request $(2 + 1) \times 106 = 318$ DWORDs for readback.

The sequence of commands for readback through ICAP was adapted from [41][37], and is shown in Table 4.1. Since the ICAP has an 8-bit data interface, the DWORDs represented in Table 4.1 must be sent in one byte per clock cycle, with the most significant byte first.

Table 4.1 - ICAP readback sequence

d = number of words in a frame. *f* = number of readback frames.

CLKs	Command	DWORD (32-bit hex)	CE_N	WE_N	Description
1:4		0xFFFFFFFF	0	0	Dummy Word
5:8	SYNC	0xAA995566	0	0	Synchronize with Cfg. FSM
9:12	NOP	0x00000000	0	0	NOP
13:16	NOP	0x00000000	0	0	NOP
17:20	W. CMD	0x30008001	0	0	Write CMD (Command Reg.)
21:24	W. CMD	0x00000004	0	0	CMD: Read Cfg. Memory
25:28	W. FAR	0x30002010	0	0	Write FAR (Frame Addr.Reg.)
29:32	W. FAR	Frame Addr.	0	0	Frame address value
33:36	R. FDRO	0x28006000	0	0	Read Frame Data Reg. Output
37:40	R. FDRO	0x48xxxxxx	0	0	xxxxxx = Read num. of DWORDS ($d + f*d$)
41:44	NOP	0x00000000	0	0	NOP
45:48		N/A	1	1	Prepare for readback
49		N/A	0	1	Ignore garbage byte
50:T_1, $T_1 = 50+d$		0x00000000	0	1	Pad Frame
T_1+1:T_2, $T_2 = T_1+1+f*d$		CFG. DATA	0	1	Configuration Memory Frame Data
T_2+1:T_2+4	W. CMD	0x30008001	0	0	Write CMD (Command Reg.)
T_2+5:T_2+8	W. CMD	0x0000000D	0	0	CMD: Desynchronize from Cfg. FSM
T_2+9:T_2+12	NOP	0x00000000	0	0	NOP

4.3.4 ICAP Frame Writer

The ICAP Frame Writer is capable of writing one or more consecutive configuration memory frames, sourcing data from a user memory. The prototype required writing two consecutive minor frames of configuration memory. As in the case of the ICAP Frame Reader, a total of 318 DWORDs must be requested for writing to account for 3 frames of data, with the last frame being a pad frame. Configuration memory write was performed with CRC check disabled.

The command sequence for configuration memory writes through ICAP was adapted from [41][42][37], and is shown in Table 4.2. The ICAP Frame Writer could be modified to support multi-frame write (MFW) command of the configuration controller. With multi-frame writes, broadcast communication in the wire emulation prototype could be realized with minimal overhead, where one source transmits to multiple destinations.

Table 4.2 - ICAP frame write sequence

d = number of words in a frame. f = number of readback frames.

*The COR value was taken from the original configuration bitstream.

CLKs	Command	DWORD (32-bit hex)	CE_N	WE_N	Description
1:4		0xFFFFFFFF	0	0	Dummy Word
5:8	SYNC	0xAA995566	0	0	Synchronize with Cfg. FSM
9:12	W. CMD	0x30008001	0	0	Write CMD (Command Reg.)
13:16	W. CMD	0x00000007	0	0	CMD: Reset CRC
17:20	W. COR	0x30012001	0	0	Write COR (Configuration Option Reg.)
21:24	W. COR	(see note)*	0	0	Disable CRC check (set bit 29)
25:28	W. IDCODE	0x3001C001	0	0	Write IDCODE (FPGA ID Code Reg.)
29:32	W. IDCODE	0x01028093	0	0	ID Code of XC2V1000 Engr. Sample
33:36	W. FAR	0x30002001	0	0	Write FAR (Frame Addr. Reg.)
37:40	W. FAR	Frame Addr.	0	0	Frame address value
41:44	W. CMD	0x30008001	0	0	Write CMD (Command Reg.)
45:48	W. CMD	0x00000001	0	0	CMD: Write Configuration Data
49:52	W. FDRI	0x30004000	0	0	Write Frame Data Reg. Input
53:56	W. FDRI	0x50xxxxxx	0	0	xxxxxx = Write num. of DWORDS ($d + f*d$)
57:T_1, $T_1 = 57+f*d$		CFG. DATA	0	0	Configuration Memory Frame Data
T_1+1:T_2, $T_2 = T_1+1+d$		0x00000000	0	0	Pad Frame
T_2+1:T_2+4		0x0000DEFC	0	0	Value to bypass CRC check
T_2+5:T_2+8	W. CMD	0x30080001	0	0	Write CMD (Command Reg.)
T_2+9:T_2+12	W. CMD	0x0000000D	0	0	CMD: Desynchronize from Cfg. FSM
T_2+13:T_2+16	NOP	0x00000000	0	0	NOP

4.3.5 ICAP Frame Copier

The ICAP Frame Copier combines the ICAP Frame Reader and ICAP Frame Writer into a single component, and simplifies user memory interface requirements. The Frame Copier streamlines the configuration transfer process; it directs the ICAP Frame Reader to readback the source frames, and then directs the ICAP Frame Writer to write the readback result into the destination frames. The ICAP Frame Copier does not support broadcasting ability, where there is one source and multiple destinations, but could if the ICAP Frame Writer utilized the MFW command.

4.4 ICAP Multi-Frame Write

Previous sections mention the use of the mult-frame write (MFW) command to minimize broadcast overheads, where data from one transport is copied to multiple transports. Multi-frame

writes reduce overheads when writing the contents of one minor configuration frame to multiple minor configuration frames. While not incorporated into the prototype, the multi-frame write command was utilized in a side experiment, where two non-adjacent minor configuration frames were written with the same contents. The system used in this experiment is shown in Figure 4.7, and consists of a system FSM, an ICAP Frame Writer with MFW capabilities, two CLB columns of RAM128x1s, and debug components. Each CLB column of RAM128x1s is considered a buffer, although no decoding logic connects the RAM blocks together. Buffers are only instantiated to reserve the LUTs in CLB columns, so that configuration frame data writes to these LUTs do not corrupt design configuration. Recall that a column of RAM128x1s maps to two adjacent minor configuration frames. For simplicity, only the first minor configuration frame from each buffer is used. The debug components are similar to those found in the prototype shown in Figure 4.4. Verification is performed on a host PC using developed software. The system communicates to a host PC through a 57600 BAUD serial link. Details on the system FSM and ICAP Frame Writer with MFW are discussed below.

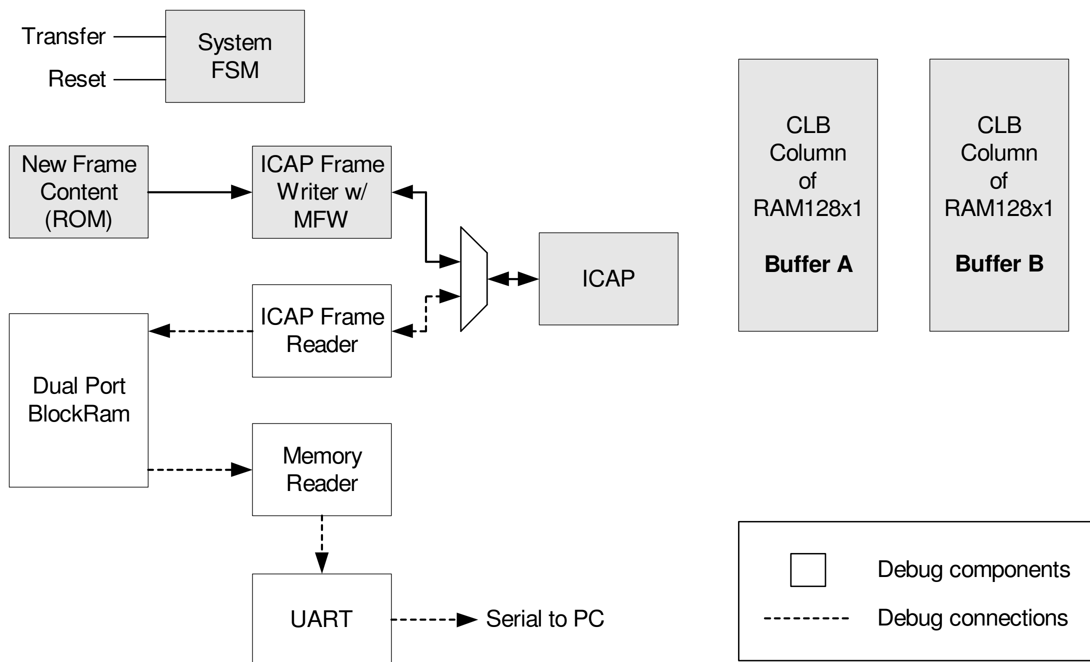

Figure 4.7 - Multi-frame write experimentation system

FSM connections not shown for clarity.

The system FSM, which executes on the assertion of the transfer signal, facilitates the verification of multi-frame writes. The FSM is as follows:

1. Readback first minor configuration frame from Buffer A. Send configuration data to host PC through UART.
2. Readback first minor configuration frame from Buffer B. Send configuration data to host PC through UART.
3. Perform a partial configuration write on the first minor configuration frames of Buffers A and B with the contents of the New Frame Content ROM.
4. Readback first minor configuration frame from Buffer A. Send configuration data to host PC through UART.
5. Readback first minor configuration frame from Buffer B. Send configuration data to host PC through UART.

Partial configuration write is performed using the ICAP Frame Writer with MFW capability, which is a modified version of the ICAP Frame Writer described in Section 4.3.4. Modifications include disabling adjacent minor frame writing, removing the pad frame write, and using the MFW command to write to random minor frame addresses. The MFW command allows the last written minor configuration frame to be written to random minor frame. Therefore, frames sharing the same contents should be written consecutively. As a result, the process of using the MFW command is simplified when the minor configuration frame access is done one at time, instead of the adjacent minor frame method described in Sections 4.3.3 and 4.3.4. Thus, adjacent minor frame writing is disabled.

Next, pad frame writes, as seen in Table 4.2, can be discarded with the use of the MFW command, as long as the pad frame write is replaced with a multi-frame write to the address of the last written configuration frame. This replacement multi-frame write may seem redundant, but without it, the pad frame is required to commit the last frame write. This use of the MFW command could also be applied to non-broadcast configuration writes in the wire emulation prototype to reduce writing overhead.

Lastly, the MFW command is used to write the contents of the last frame write to random frame addresses. In this experiment, the MFW command was used to write to a total of two minor configuration frames. The ICAP transactions used in this experiment are shown in Table 4.3. Note that the group of transactions highlighted in orange replaces the pad frame write, and reduces overhead by d – 28 clock cycles. Also note that the group of transactions highlighted in green can be repeated as many times necessary to write to other addresses. Therefore, the

increased cost of writing an additional frame of identical contents is an additional 28 clock cycles, a significant improvement over the traditional partial configuration writing methods described in Section 4.3.4.

Table 4.3 - ICAP frame writer with multi-frame write

Two non-adjacent minor configuration frames are written.

d = number of words in a frame.

*The COR value was taken from the original configuration bitstream.

CLKs	Command	DWORD (32-bit hex)	CE_N	WE_N	Description
1:4		0xFFFFFFFF	0	0	Dummy Word
5:8	SYNC	0xAA995566	0	0	Synchronize with Cfg. FSM
9:12	W. CMD	0x30008001	0	0	Write CMD (Command Reg.)
13:16	W. CMD	0x00000007	0	0	CMD: Reset CRC
17:20	W. COR	0x30012001	0	0	Write COR (Configuration Option Reg.)
21:24	W. COR	(see note)*	0	0	Disable CRC check (set bit 29)
25:28	W. IDCODE	0x3001C001	0	0	Write IDCODE (FPGA ID Code Reg.)
29:32	W. IDCODE	0x01028093	0	0	ID Code of XC2V1000 Engr. Sample
33:36	W. FAR	0x30002001	0	0	Write FAR (Frame Addr. Reg.)
37:40	W. FAR	1st Frame Addr.	0	0	Frame address value
41:44	W. CMD	0x30008001	0	0	Write CMD (Command Reg.)
45:48	W. CMD	0x00000001	0	0	CMD: Write Configuration Data
49:52	W. FDRI	0x30004000	0	0	Write Frame Data Reg. Input
53:56	W. FDRI	0x50xxxxxx	0	0	xxxxxx = Write num. of *d* DWORDS (one minor frame)
57:T_1, $T_1 = 57+d$		CFG. DATA	0	0	Configuration Memory Frame Data
T_1+1:T_1+4		0x0000DEFC	0	0	Value to bypass CRC check
T_1+5:T_1+8	W. FAR	0x30002001	0	0	Write FAR (Frame Addr. Reg.)
T_1+9:T_1+12	W. FAR	1st Frame Addr.	0	0	Frame address value
T_1+13:T_1+16	W. CMD	0x30008001	0	0	Write CMD (Command Reg.)
T_1+17:T_1+20	W. CMD	0x00000002	0	0	CMD: Multi-frame write (MFW)
T_1+21:T_1+24	W. MFW	0x30014002	0	0	Write MFW (MFW Reg.) with 2 words
T_1+25:T_1+28	W. MFW	0x00000000	0	0	Dummy word
T_1+29:T_1+32	W. MFW	0x00000000	0	0	Dummy word
T_1+33:T_1+36	W. FAR	0x30002001	0	0	Write FAR (Frame Addr. Reg.)
T_1+37:T_1+40	W. FAR	2nd Frame Addr.	0	0	Frame address value
T_1+41:T_1+44	W. CMD	0x30008001	0	0	Write CMD (Command Reg.)
T_1+45:T_1+48	W. CMD	0x00000002	0	0	CMD: Multi-frame write (MFW)
T_1+49:T_1+52	W. MFW	0x30014002	0	0	Write MFW (MFW Reg.) with 2 words
T_1+53:T_1+56	W. MFW	0x00000000	0	0	Dummy word
T_1+57:T_1+60	W. MFW	0x00000000	0	0	Dummy word
T_1+61:T_1+64	W. CMD	0x30080001	0	0	Write CMD (Command Reg.)
T_1+65:T_1+68	W. CMD	0x0000000D	0	0	CMD: Desynchronize from Cfg. FSM
T_1+69:T_1+72	NOP	0x00000000	0	0	NOP

Chapter 5

Results and Analysis

This chapter presents the results and analysis of the prototype described in Chapter 4. Here, issues relating to power, latency, throughput and efficiency will be presented. Also, results of the multi-frame write experiment described in Chapter 4 are presented. In addition, improvements to the existing Xilinx configuration interface that dramatically improve prototype performance are discussed. Finally, projected performance of a predicted wire emulation system for FPGAs is presented.

5.1 Prototype Wire Emulation System

It is expected that an alternate form of cross-chip communication can be realized using wire emulation, one that improves FPGA scalability and reduces routing congestion. The purpose of constructing a prototype was to demonstrate a wire emulation system that utilizes self-reconfiguration on existing FPGAs. This prototype demonstrated a one-way communication channel between a TX and RX transport, which was realized using self-reconfiguration instead of traditional programmable interconnects. The prototype was implemented and tested on an XC2V1000 engineering sample, with speed grade –4 and FG256 packaging, using a Memec Insight DS-V2LC rev2 development board [43]. The development board also includes an RS232 serial port (line drivers on-board), 32 MB DDR memory, 24 MHz and 100 MHz oscillators, voltage regulators, and various debugging components, such as push buttons, switches, a seven-segment display, and header pins for general purpose I/O.

A VHDL description of the prototype was synthesized using Synplicity Synplify 7.2 and implemented using the Xilinx ISE 5.1 tool flow. As described in Section 4.3.2.4, additional unconventional processing was performed within the Xilinx ISE tool flow. An FPGA configuration bitstream and its design information were produced at the end of the Xilinx ISE tool flow. The implemented FPGA, whose layout is shown in Figure 5.1, is estimated by the Xilinx ISE tools to operate at 75 MHz, even though the prototype communicates with the ICAP in a method theoretically limited to operating at 50 MHz. Using ICAP beyond the 50 MHz requires handshaking, providing minimal benefit because the ICAP hits a performance limit at 60 MHz.

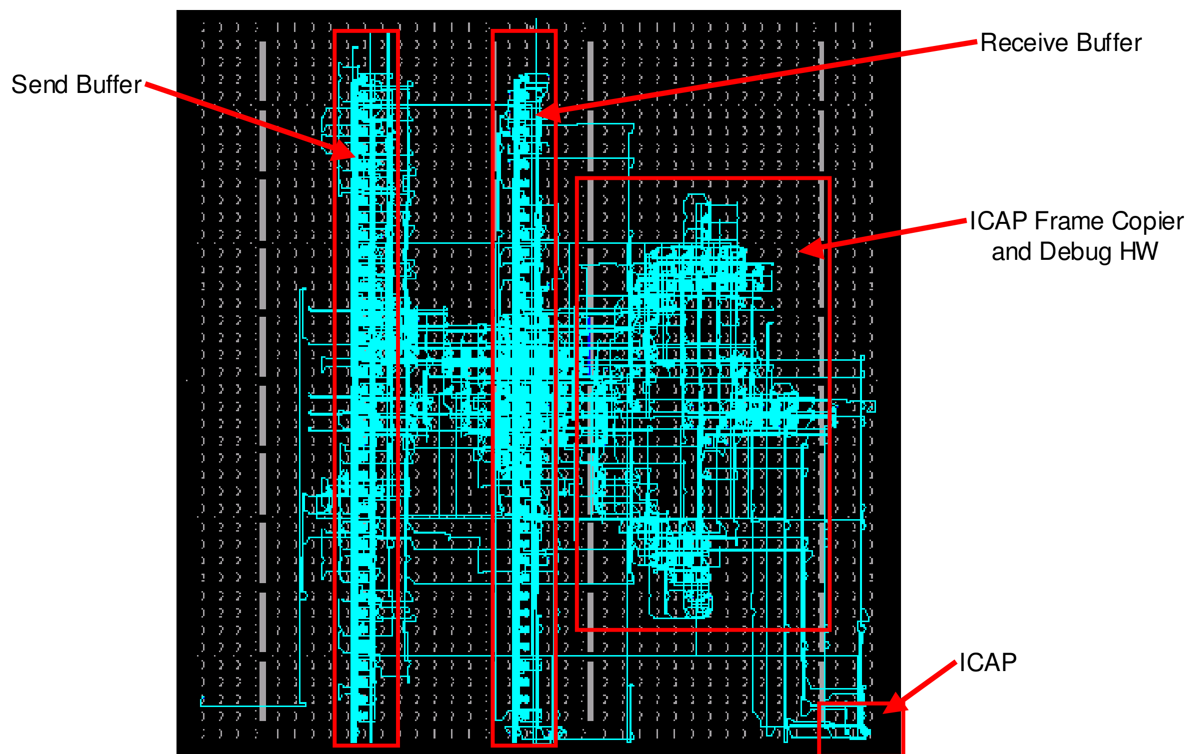

Figure 5.1 - Prototype implementation in XC2V1000

The prototype used 13% of the slices on an XC2V1000, with transport buffers contributing to 46% of the entire design. Furthermore, distributed memory contributed to 85% of the transport interface buffers, with the remaining 15% used for address decoding logic. Each additional transport buffer can be added with a 3% utilization increase. The self-reconfiguration driver, consisting of ICAP control and debug logic, is quite small, consuming 7% of total FPGA resources. The growth rate of the self-reconfiguration logic in reaction to an increased number of

transport buffers is small, since the FSM that schedules frame copying will only change. Overall, the area overheads of the prototype are small but significant in an XC2V1000. If the prototype were implemented in a larger FPGA, the self-reconfiguration driver will consume a smaller percentage of logic, while the transport buffers will grow in size due to a larger CLB column and an increased amount of address decoding logic. On the other hand, total FPGA utilization percentage may decrease, because the total number of CLB columns increases. While not accounting for address decoding logic, a simple method of estimating FPGA utilization of a single transport buffer is to take the inverse of the total number of CLB columns in the FPGA. Also recall that for the purposes of maximum throughput, a unified memory buffer was used to implement a transport. If implementing the register-like approach, as mentioned in Section 4.3.2.3, address decoding is unnecessary and the transport interface is more natural, therefore minimizing implementation overhead at the expense of throughput.

5.1.1 Verification

Verifying the correctness of the prototype was done on two fronts – inspection and operational verification. Design simulation was not used because self-reconfiguration cannot be simulated in conventional HDL simulators. First, Xilinx *FPGA Editor* was used to visually inspect that transport buffers had consistent placement and pin assignments. Inspecting all distributed memory blocks would be tedious, so only a few randomly selected distributed memory blocks were verified. Afterward, it was deemed worthwhile to perform operational verification.

Operational verification required that the prototype run on an FPGA and that written software parse prototype debug data. As mentioned in Section 4.3.1, this data consisted of configuration readback information of both transport buffers before and after the self-reconfiguration transfer. In addition, transport buffers were read back using transport user interfaces. One can determine if problems are associated with a configuration memory transfer or a transport buffer placement/pin assignment by inspecting both configuration and user interface readback. Control of self-reconfiguration transfers was conducted using a push button found on the development board, where transfers occurred while the push button was depressed. Verification was performed on the host PC, where it was shown that the two transport buffers matched after self-reconfiguration transfer.

5.1.2 Performance

The prototype was verified to operate at up to 46.7 MHz. Testing at various clock speeds was tedious, because a VHDL file had to be modified to account for clock frequency, since the clock divider/PLL and UART settings were clock speed dependent. Therefore, the clock speed changes required rerunning the synthesis and implementation tools. While 46.7 MHz is shy of the maximum theoretical free-running mode speed of ICAP (50 MHz), marginal benefit would be seen at 50 MHz, due to the large overheads introduced by the 8-bit data port of the ICAP. After further debugging, it was determined that the user programmable logic was not a source of failure. Therefore, failures above 46.7 MHz were most likely due to engineering sample unreliability.

Since the wire emulation system is deterministic and system FSM clock cycles are predictable, performance numbers such as latency, throughput, and efficiency can be calculated. Latency is described as the time needed for the contents in the transmitter to be reflected in the receiver, or more simply, the time needed to perform one self-reconfiguration transfer. Throughput is described as the effective user data transfer rate when using self-reconfiguration. Lastly, efficiency describes the percentage of user bits within a minor configuration frame. Recall that two consecutive frames must be transferred to copy the LUT contents of an entire CLB column. The following equations were used to calculate system performance:

$$T_c(n) = T_r(n) + T_w(n) \quad (1)$$

$$T_r(n) = t_{rhdr} + (n + 1)\, t_{frame} + t_{rtail} \quad (2)$$

$$T_w(n) = t_{whdr} + (n + 1)\, t_{frame} + t_{wtail} \quad (3)$$

where

n is the number of consecutive minor configuration memory frames

$T_c(n)$ is the time required to copy n consecutive minor frames

$T_r(n)$ is the time required to read n consecutive minor frames

$T_w(n)$ is the time required to write n consecutive minor frames

t_{rhdr} and t_{rtail} are the times required to send read request header and tail, respectively

t_{whdr} and t_{wtail} are the times required to send write request header and tail, respectively

t_{frame} is the time required to shift in/out a minor frame through ICAP

These equations assume that two consecutive minor frames are read and then written. Time in these equations can be represented in either clock cycles or physical time. Notice that (2) and (3) account for a pad frame with the (n + 1) term. Using these equations and the FPGA CLB data from [8], performance for all Virtex-II family members was calculated and is shown in Table 5.1. This table assumes operation at 46.7 MHz. Maximum throughput calculations assume that all distributed memory (user storage) bits in a CLB column are used.

Table 5.1 - Calculated performance of Xilinx Virtex-II family operating at 46.7 MHz

FPGA	Storage Capacity per CLB Col. (bits)	CLB Copy Time (clks)	Latency (us)	Max Throughput (Mb/s)	Max Throughput (MB/s)
XC2V40	1024	757	16.21	63.17	7.90
XC2V80	2048	1237	26.49	77.32	9.66
XC2V250	3072	1717	36.77	83.55	10.44
XC2V500	4096	2197	47.04	87.07	10.88
XC2V1000	5120	2677	57.32	89.32	11.16
XC2V1500	6144	3157	67.60	90.89	11.36
XC2V2000	7168	3637	77.88	92.04	11.50
XC2V3000	8192	4117	88.16	92.92	11.62
XC2V4000	10240	5077	108.72	94.19	11.77
XC2V6000	12288	6037	129.27	95.06	11.88
XC2V8000	14336	6997	149.83	95.68	11.96

Another aspect of this system that can be calculated is efficiency. With configuration memory frame sizes always greater than the number of LUT bits in a slice column, efficiency will never reach 100%. This is due to the IOB configuration bits at the beginning and end of each configuration memory frame, and the 8-bit gaps between LUTs in the same slice. Efficiency, shown in Table 5.2, was calculated by dividing the number of user storage bits in a slice column by the number of bits in minor configuration frame. Storage capacity was calculated by multiplying the number of CLB rows (CLBs per column) by the number of slices per CLB (4) and by the number of LUT bits per slice (2 x 16). The number of CLB columns is also listed to show the number of possible instantiated transports, where each transport consumes one CLB column.

Table 5.2 - FPGA user storage and frame utilization

FPGA	CLB Rows	CLB Cols.	Slices per CLB Col.	Storage Capacity per CLB Col. (bits)	Storage Capacity per Slice Col. (bits)	Minor Cfg. Mem. Frame Size (bits)	Efficiency
XC2V40	8	8	32	1024	512	832	61.5%
XC2V80	16	8	64	2048	1024	1472	69.6%
XC2V250	24	16	96	3072	1536	2112	72.7%
XC2V500	32	24	128	4096	2048	2752	74.4%
XC2V1000	40	32	160	5120	2560	3392	75.5%
XC2V1500	48	40	192	6144	3072	4032	76.2%
XC2V2000	56	48	224	7168	3584	4672	76.7%
XC2V3000	64	56	256	8192	4096	5312	77.1%
XC2V4000	80	72	320	10240	5120	6592	77.7%
XC2V6000	96	88	384	12288	6144	7872	78.0%
XC2V8000	112	104	448	14336	7168	9152	78.3%

Table 5.1 shows that maximum throughput slightly improves while latency worsens with increased FPGA size. Maximum throughput grows slowly when comparing larger devices. This is due to the reduction in configuration memory access header and tail overhead relative to configuration frame size, since header and tail time stays constant regardless of device, while configuration frame size increases with FPGA growth. Therefore, the configuration FSM command overhead reduces as FPGA size increases, but this reduction is small and insignificant in larger devices. Another reason for slow increase in maximum throughput is that the number of clock cycles required for a CLB copy increases with user storage capacity in a CLB column. The last reason for slow maximum throughput increases can be attributed to slight increases in efficiency, as shown in Table 5.2.

A more obvious trend is that latency worsens with FPGA growth, because it takes longer to copy more data; therefore, this system scales poorly with size, not to mention that latency times for any device are unacceptable, since an ideal latency should be that of an IC long wire.

Using the register-like approach to transport interfaces, throughput and efficiency would have been 1/16th the numbers shown in Table 5.1 and Table 5.2 respectively, since only one address and therefore one storage bit is used from each LUT. Latency numbers are the same as those shown in the Table 5.1, since the number of involved configuration frames does not change, assuming that an entire CLB column is either all transmitters or all receivers. If the CLB is split

into half, such that one slice column is for transmitting and the other is for receiving, latency will increase and throughput numbers will decrease. Because one minor frame will be read and the other will be written, minor configuration frames must copied one at a time instead of two at time. Reading and writing one frame at a time is more time consuming than two frames at a time, since command headers and tails, and more significantly read or writing a pad frame, must be accounted for in each access.

5.1.3 Power

Since the configuration controller of the Virtex-II usually idles after initial configuration, there was interest in measuring the prototype's current consumption, where the configuration controller continuously runs during design operation. To measure maximum current consumption, the prototype was modified such that debug circuitry was not executed in the system FSM. Also, any debugging output pins of the FPGA were disabled, so that only necessary IOBs remained in the design. Lastly, the prototype was operated at the maximum verified clock speed.

The measurement setup required a 1-ohm power resistor between the grounds of the board and the 5V power supply. Current consumption was measured by measuring the voltage across the resistor. In addition, an oscilloscope was used to detect any spikes in the voltage supply, with no spikes to be found. While this method measures the current consumption of the entire board, devices other than the FPGA were disabled or set in a current minimizing state. It would have been ideal to measure current at the pins of the FPGA, but the BGA packaging and the sheer number of power pins prevented us from doing so.

The current was measured with the FPGA in several situations: unconfigured, configured but in reset, configured with frame copier disabled, and configured with frame copier enabled. The results of this experiment are shown in Table 5.3. Measuring the unconfigured FPGA allows us to estimate the static consumption of the FPGA, along with any other board power overheads, i.e. oscillators and voltage regulators. Measuring the configured FPGA with logic in reset allows us to examine the increase in consumption by a configured, non-switching FPGA. Then, the current is measured with the frame copier disabled, which allows all logic to run except those that drive the Virtex-II configuration controller. Lastly, the current is measured with the frame copier enabled, which allows all logic and self-reconfiguration to operate.

Table 5.3 - Prototype (no debug HW) current consumption

Test Case (operating at 46.7 MHz)	Current (mA)
FPGA unconfigured	38.0
FPGA configured, in reset	40.9
FPGA configured, frame copier disabled	53.9
FPGA configured, frame copier enabled	61.6

The results in Table 5.3 show that self-reconfiguration required an additional 7.7mA or 14% of current; therefore, it is shown that self-reconfiguration can be performed with a reasonable increase in current consumption.

5.1.4 Multi-Frame Write Experiment

While multi-frame write was not incorporated into the wire emulation prototype, the separate experiment described in Section 4.4 was verified on the same FPGA and development board used by the prototype. The host PC software used to verify the prototype required little modification to process the debug data of the experiment. Therefore, the multi-frame write transactions described in Table 4.3 could be incorporated into the wire emulation prototype to implement broadcast (one-to-many) communication with reduced overheads.

The use of multi-frame writes for broadcast communication follows a set performance calculations that differ from (1) and (3), since multi-frame writes are more efficiently implemented when adjacent minor configuration frames are not written together. Therefore, these equations would better reflect broadcast communication if MFW commands were incorporated into the wire emulation prototype:

$$T_c(n, b) = T_r(n) + T_w(n, b) \quad (4)$$

$$T_w(n, b) = (n)(t_{whdr} + t_{frame} + (b)\, t_{mfw} + t_{wtail}) \quad (5)$$

where

n is the number of consecutive minor configuration memory frames
b is the number of broadcast destinations

$T_c(n, b)$ is the time required to copy *n* consecutive minor frames to *b* destinations

$T_r(n)$ is the time required to read *n* consecutive minor frames; see (2)

$T_w(n, b)$ is the time required to write *n* consecutive minor frames to *b* destinations

t_{whdr} and t_{wtail} are the times required to send write request header and tail, respectively

t_{frame} is the time required to shift in/out a minor frame through ICAP

t_{mfw} is the time required to send a multi-frame write request

Consecutive minor frame readback is still assumed, since the transport buffers used in the wire emulation prototype are adjacent in configuration memory. Therefore, (2) holds true in broadcast communication. Using (2), (4) and (5), a version of the wire emulation prototype supporting the MFW command implemented on a Xilinx XC2V1000 operating at 46.7 MHz would have the calculated broadcast throughput and latency shown in Table 5.4.

Table 5.4 - Estimated broadcast performance on XC2V1000 operating at 46.7 MHz

No. of Destination Transport Buffers	CLB Read Time (clks)	CLB Write Time (clks)	CLB Copy Time (clks)	Total Storage Copied (bits)	Latency (us)	Throughput (Mbps)	Throughput (MB/s)
1	1333	1040	2373	5120	50.81	100.8	12.6
2	1333	1096	2429	10240	52.01	196.9	24.6
3	1333	1152	2485	15360	53.21	288.7	36.1
4	1333	1208	2541	20480	54.41	376.4	47.0
5	1333	1264	2597	25600	55.61	460.3	57.5
6	1333	1320	2653	30720	56.81	540.8	67.6
7	1333	1376	2709	35840	58.01	617.8	77.2
8	1333	1432	2765	40960	59.21	691.8	86.5
9	1333	1488	2821	46080	60.41	762.8	95.4
10	1333	1544	2877	51200	61.61	831.1	103.9

Table 5.4 demonstrates that latency remains about the same as the regular configuration frame copy procedure, but throughput increases as the number of destination transport buffers increases. Improved throughput is a result of latency increasing slowly in relation to the amount of copied data.

5.1.5 Improvements to Performance

As discussed in Section 5.1.2, the 8-bit data port of the ICAP and the overheads of configuration read and write commands limit prototype performance. These bottlenecks are not necessary,

knowing that minor configuration memory frames can be read or written to in one configuration clock cycle. In addition, the behavior of the MFW command, as investigated in Section 4.4, leads one to believe that there is an internal register the size of minor configuration memory frame inside the Virtex-II configuration controller, since the contents of the last written frame can be written to other addresses in the time it takes to shift seven 32-bit words, as seen in Table 4.3. If configuration memory could be readback into this internal register, without the need to shift out the result through the ICAP, then readback could potentially be implemented in one clock cycle. Then, writing the contents of the internal register to destination frame with one additional clock cycle performs a configuration copy; therefore, a configuration copy could take as little as two clock cycles, although it would be reasonable to assume that several additional clocks may be necessary to decode the source and destination addresses. This modification to the Virtex-II would dramatically increase the performance of the prototype system, where minimum latency would be on the order tens of nanoseconds and throughput would be on the order of 10s to 100s of Gb/s. In addition, throughput would increase while latency would remain constant with FPGA size, since the LUT storage size per CLB column increases with FPGA size and configuration memory access time would be independent of CLB column size. Finally, this scheme of configuration memory copy enables position independent performance, since configuration memory access is consistent.

5.2 Wire Emulation Architecture Analysis

While the wire emulation prototype implemented in this thesis has promise, given modifications to enabling faster configuration memory access, it still remains unsuitable for latency-driven performance. Consider that the propagation delay of the longest programmable wire in the FPGA is on the order of several nanoseconds, as shown in Table 2.1, while accesses to configuration SRAM could be as fast as several nanoseconds. Copying SRAM contents could take twice long as the propagation delay of a long wire. Unless the access time to configuration memory contents improves dramatically through the use of future memory technology, this serialized method of self-reconfiguration becomes a non-ideal choice for wire-emulation implementation, especially when multiple transport pairs must be serviced in a round-robin fashion. In addition, this implementation still relies on the FPGA architecture, where configuration memories introduce performance and area overheads, with unknown scalability of configuration memory performance.

An ideal FPGA wire emulation architecture should not be implemented with a system that relies on FPGA configuration memory transfers to realize communication. In addition, scaling issues are better addressed in a parallel system, since scalability is an issue in systems whose transfers must all occur through a single, common device; therefore, it is recommended that a wire emulation system be modeled after a network-on-chip, where parallelism and resource reuse are evident. In addition, this NoC-like implementation may need a global asynchronous, local synchronous (GALS) time domain, because global synchronization of NoC components may be difficult to implement. Global clock distribution with a frequency equal to the link speed could be costly to design. If a distributed resource NoC were developed, then routers and network interfaces would have to be asynchronous to one another at high operational speed, or synchronous at low operational speed.

If such a wire emulation system were built to enhance FPGA architectures, cross-chip communication performance and congestion could improve. A network built with consideration for power is important, since wires consume a significant portion of power. Differential wiring may meet the performance and power requirements of network links. Differential wires help minimize line current requirements, improve propagation delay, reduce power, and minimize noise. Differential wire performance potential within an IC has been difficult to quantify in current and future manufacturing technologies, due to the confidentiality of this information. It was demonstrated in [6] that a differential 10mm long wire pair in 0.18um technology could achieve 1GHz operation, with an efficiency of 1 pJ/bit. The wire resistance was targeted to be comparable to a 4.2mm differential wire pair implemented in 100nm technology for use in a smart memory. More recent data has been difficult to find, but other developments in off-chip high-speed serial communications may indicate future direction. Take for example the RocketIO [44] resources of the Virtex-II Pro X, which can serially transmit data through differential wire at speeds ranging from 3 Gb/s to 10 Gb/s. RocketIO utilizes current mode logic and coding schemes such as 8B/10B for clock recovery and DC balance. Also, RocketIO also uses pre-emphasis to boost higher frequencies and compensate for lossy communications in PCB traces.

Another possibility for NoC links is wave pipelining [45]. In [45], link operation clock speeds of 3.45 GHz were simulated on a 10mm long wire with a 3.2um pitch and 1.2um width on 0.25um technology. A system supporting 16 wires was simulated to consume 1039 mW, with an

efficiency of 18.8 pJ/bit [45]. While the potential bandwidth is excellent, the power efficiency is less than that of differential wires.

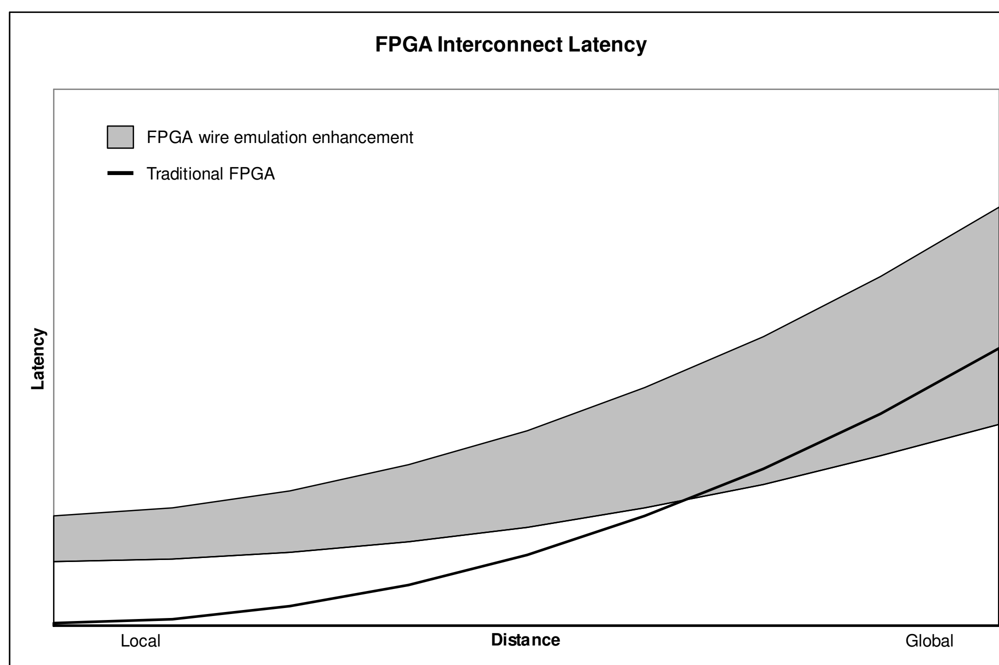

Figure 5.2 - Potential FPGA interconnect latency

The latency of chip-length (global) programmable interconnects grows with the FPGA size and is on the order of several nanoseconds, as discussed in Chapter 2. The wire emulation system for FPGA enhancement will not improve latency in local and semi-global programmable interconnects since the latency of NoC hop routing in combination with NoC link latency will likely be greater than the latency of the aforementioned interconnects. On the other hand, the latency of NoC links and hop routing latency may be smaller or greater than the latency of a global programmable interconnect. The actual latency of the NoC-like wire emulation system could be smaller than that of a global programmable interconnect if the low-swing voltage transitions of differential wires enable signal propagation delays lower than that of traditional wires. As mentioned before, [6] implemented a 10mm differential wire pair in 0.18um technology with a propagation delay of 1ns. Assuming that differential wire pair performance has increased since then, and that we can implement wires of the same width in future technologies, or more precisely, wires of the same resistance, then differential wires will have a minimum propagation delay of 1ns at 10mm length. Also assume that future FPGAs will not greatly surpass the die size of the largest FPGA in the Virtex-II family, the XC2V8000, whose die

was identified in Table 2.1 to have a maximum edge dimension of 26mm. This assumption is reasonable since manufacturing yields may decrease with increases in die size. Therefore, an NoC-like approach could be implemented on the largest FPGA with a maximum of three differential links, each approximately 9mm in length; therefore, NoC links could contribute a maximum latency of 3ns with differential wires, with minimum latency around 100s of ps with wave pipelining [45].

The other component to NoC latency is router hop latency. Routers may operate at speeds slower than the link latency to operate synchronously, where a global clock distribution is difficult at higher speeds. Worst-case router clock rates would be around 500MHz, since current FPGAs can distribute a programmable global clock at this frequency with controlled skew. Best-case router clock rates could be achieved if NoC components were globally asynchronous, where components could be clocked faster than 500MHz. Assume that data may experience forwarding latency of several router clock cycles at each intermediate hop. This would be possible in a routing scheme that utilizes schedules to minimize data buffering at router hops, where data is immediately forwarded to the next hop, as demonstrated in NoCs implementing virtual connections. Therefore, upper-bound routing hop latency could be between 5-15ns, while lower-bound latency could be as high as 1ns (slowest link latency) and as low as 100s of ps (potential link latency). Since the minimum number of maximum length links is three, the hop count could be as small as two.

The enhanced FPGA could have latency as shown in Figure 5.2, where cross-chip latency has a lower bound of about 2-3ns, assuming asynchronous NoC with wave pipelining link latency and an upper bound of about 48ns, assuming synchronous NoC with differential wire link latency. For comparison, the worst latency of an existing FPGA programmable cross-chip interconnect is approximately 6ns (Table 2.1). Since the length of a cross-chip wire does not change with scaling, then it is a fair assumption that the latency of these wires does not improve with scaling. In summary, the wire emulation system could improve cross-chip latency, even in future technologies, because the wire emulation system is built with shorter wires, and therefore can scale with manufacturing technology.

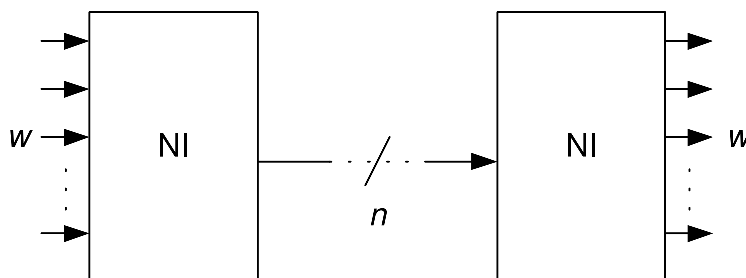

Figure 5.3 - Simplified wire emulation system

Another aspect to performance is effective global wire throughput, as shown in Figure 5.4. First, the throughput of a traditional FPGA wire is one bit per clock cycle, where a clock cycle is the propagation of the wire. Using the cross-chip delay for the largest FPGA of 6ns, a traditional FPGA wire has the throughput of 166 Mb/s. The wire emulation can be simplified, as shown in Figure 5.3, to approximate the throughput of an emulated wire. For this approximation, let n be the width of link, let b be the bandwidth of each link wire, let k be router efficiency, where $0 < k \leq 1$, let h be the number of hops, let i be the injection rate into the network, where $0 < i \leq 1$, and let w be the number of emulated wires for a given pair of network interfaces (NIs). Therefore, the throughput of an emulated wire (z) can be expressed as

$$z = i\, k^h\, n\, b \,/\, w \qquad (5)$$

Since routers are inserted between link segments, k accounts for the ability of a router to forward a data at link speed. An ideal router will have an efficiency (k) of 1, and route all data at link speed. Note that a router can achieve an efficiency of 1 without zero forwarding delay if a router can pipeline data internally at link speed. If the router cannot forward data at link speed, which may be the case of routers operating at frequency slower than link frequency, then $k < 1$. This model also assumes that k is equal for all encountered routers, which may not always be a realistic case, since congestion can vary k at each hop.

To approximate upper-bound throughput of an emulated wire, for the moment assume that globally asynchronous routers can route data at link speed, therefore having an efficiency $k = 1$. Since ideal router efficiency is 1, then h, the number of hops, can be omitted in (5). In addition, assume that each NI can emulate four wires in one direction, therefore $w = 4$. If more wires were emulated simultaneously, then emulated wire throughput would decrease. If fewer wires were emulated simultaneously, then NIs would provide limited usability and link bandwidth might be

underutilized. Also, assume that NIs can inject data at all times, implying $i = 1$. Lastly, assume that links can be implemented with $n = 1$ to 4 wires, with an individual bandwidth $b = 1$ Gb/s to 3.45 Gb/s, where link are implemented with differential wire pairs or wave pipelining, although future differential wire pairs may improve on these numbers. Using (5), the upper bound can be at least 3.45 Gb/s, where the $n = w$.

Approximation of lower bound throughput of an emulated wire is more difficult, since congestion and router efficiency are implementation dependent. Perhaps a worst-case scenario would be a NoC implementation where links used traditional wires and NoC routers and NIs were globally synchronized at minimum of 500 MHz, the maximum speed of a globally distributed clock in Virtex-II. With global NoC synchronization, virtual channels can be implemented, providing a means for guaranteed latency and throughput. Therefore, assume lower bound throughput will use one link wire operating at 500 MHz, and each NI emulates four wires. Assume the use of virtual channel router scheduling allows for new data to be injected into the network once every four NoC clock cycles, so let $i = 0.25$. Virtual channel routing allows data to be forwarded at NoC speed with latency only associated with pipelining internal to the router, so let $k = 1$; therefore, the hop count h is negligible in (5). Using (5), the lower bound of an emulated wire is 31.3 Mb/s.

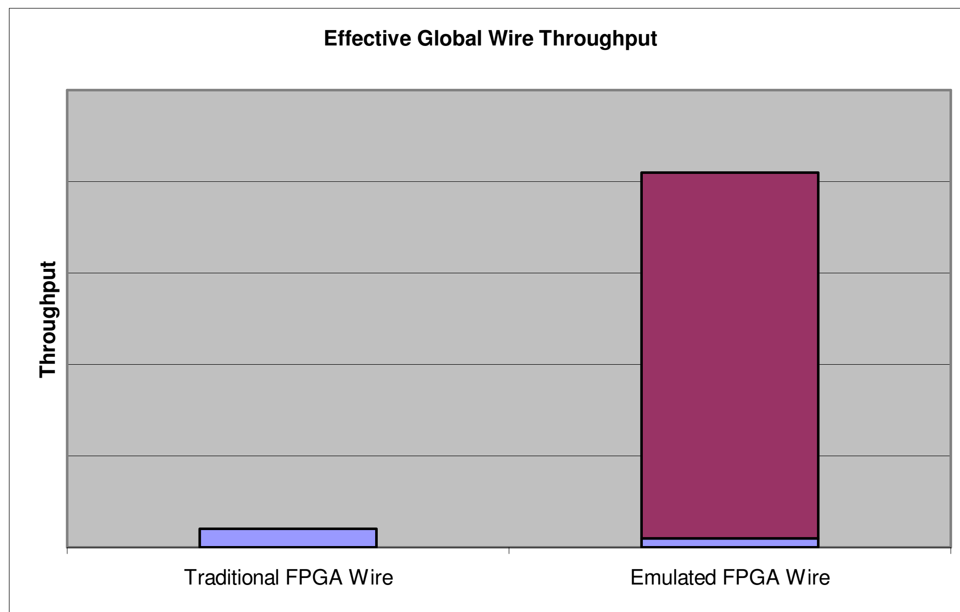

Figure 5.4 - Potential effective global wire throughput

In addition to performance, global routing congestion can be reduced on two fronts: FPGA architecture routing and user routing. By applying Rent's Rule [4], it is predicted that the number of global routing resources must increase as FPGA density increases; therefore, a wire emulation system that incorporates serial links will decrease global routing resource congestion through reuse and wire consolidation. User design global routing congestion will decrease as well, although reduction is design specific. Designs requiring complex global routing will benefit most, such as an SoC design with a high fan-out interconnection bus. The congestion of a design with complex global routing on a traditional FPGA is shown in Figure 5.5, where congestion is greatest at the center of the FPGA in attempts to minimize fan-out delay. If this design were to be implemented in the enhanced FPGA architecture, congestion would resemble Figure 5.6. In the enhanced architecture, minimal congestion is evident, with most congestion appearing in areas surrounding network interfaces, where network interfaces are evenly distributed in a 4x4 mesh. Congestion is only seen in these regions because the nets requiring global routing connect to these network interfaces. The wire emulation architecture handles the data movement that effectively realizes the global routing requirements, and relieves traditional FPGA routing from implementing these connections.

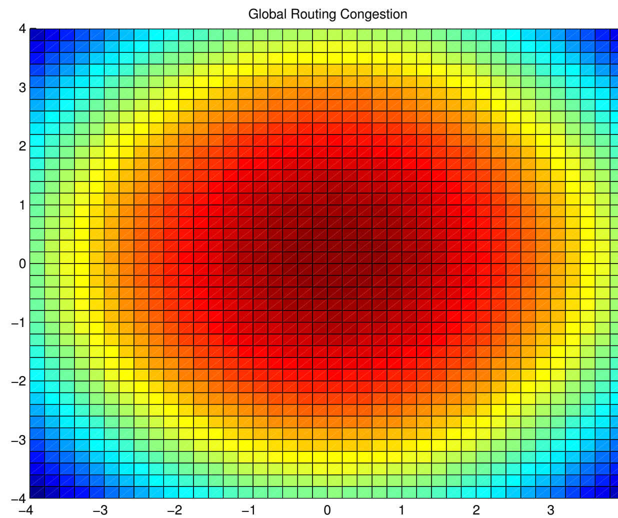

Figure 5.5 - Traditional FPGA user global routing congestion

(warmer color = greater congestion)

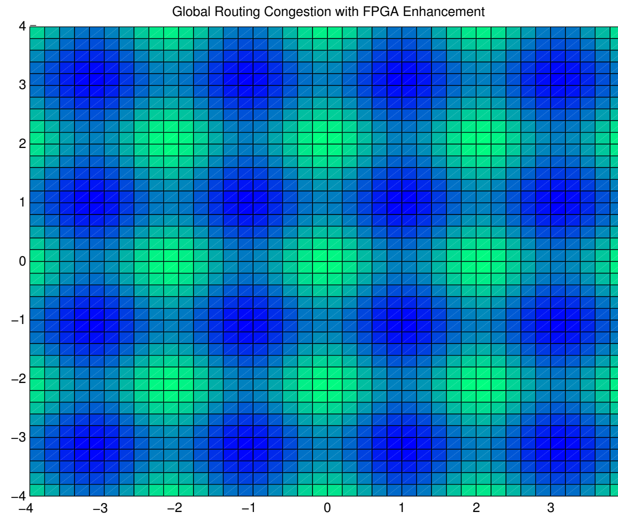

Figure 5.6 - Enhanced FPGA user global routing congestion

(warmer color = greater congestion)

Enhanced FPGA uses a wire emulation system consisting of 16 NIs evenly arranged in a 4x4 mesh.

Reducing congestion and eliminating red “hot-spots”, as shown in Figure 5.6, allows global communication to have optimal routing delays. As a result, a design implemented on the enhanced FPGA will have improved performance over a traditional FPGA, because heavy congestion leads to sub-optimal route delays.

5.2.1 Effects on FPGA Layout

Wire emulation components might disturb the layout regularity of the FPGA. This section discusses the implications of network interface (NI) and routing resource placement.

Since wire emulation NIs must interact with programmable logic, NIs should be placed to provide easy access to existing programmable interconnects and minimize CLB and programmable interconnect layout modification. By providing easy accessibility and minimizing modifications to existing layout regularity, FPGA design tool modification can be minimized. A similar situation is encountered in the Virtex-II architecture, where BlockRAM and multipliers are placed within a sea of CLBs. BlockRAMs and multipliers are organized as columns that span the full

length of the FPGA to minimize routing irregularity. This column approach could be used in placing NIs. Columns of NIs could be placed near the edges and the center of the FPGA programmable fabric. NIs near FPGA edges support cross-chip communication, and provide access to both programmable logic and IOBs; NIs near the center support communication that branches outward, towards the edges of the FPGA. To support further flexibility, a row of NIs could be placed near the top and bottom edges of the FPGA, improving NI accessibility to IOBs on these edges of the FPGA. Another approach to NI placement is to evenly distribute NIs in both axes of the FPGA programmable fabric. To maintain programmable routing regularity, NIs may be required to replace CLBs. With either NI placement approach, programmable interconnection switches may require modification to support the NI user interface.

Wire emulation routing resource placement is flexible, since these resources do not interact with FPGA programmable resources. None the less, efforts to minimize intervention of FPGA layout regularity should be made to simplify CAD tool design. If routing resources are large enough that they must replace CLBs, it is expected that the FPGA design tools can account for interruption of regularity within a sea of CLBs. An example of this situation is found in the Xilinx Virtex-II Pro, where at least one PowerPC microprocessor core is placed in the middle of a sea of CLBs. In a NoC-like wire emulation system, routing resources could be placed to evenly distribute propagation delays of links between routing resources and NIs.

Wire emulation system effects on FPGA wiring congestion is implementation dependent. If the wire emulation replaces a portion of the cross-chip programmable interconnects, then it is possible to reduce cross-chip interconnection congestion if the wire emulation system utilizes fewer cross-chip interconnects. Utilization efficiency of FPGA cross-chip wires is user design dependent. In a user design that requires complex global communication, wires associated with the wire emulation system may have a higher probability of utilization than those associated with traditional cross-chip programmable interconnects. Traditional cross-chip programmable interconnects are selectable on an individual basis and provide a single connection, whereas wire emulation systems may reuse resources and wires to actively serve multiple connections.

With a wire emulation system implemented with a NoC approach, wiring congestion is dependent on network topology and link width. Congestion caused by NoC links is dependent on NoC router placement. If routers are placed close together, links can be implemented on lower metal layers since shorter wires can be used, and therefore only affect lower metal layer congestion. If

routers are placed far apart, links can be implemented with upper metal layer wires, which allow for longer, wider wires. In this case, wiring congestion can be affected on several metal layers, because vias can occupy space in middle and lower metal layers. Also, an NoC approach may require additional global clock signals to synchronize NoC routers and NIs, increasing congestion in the upper metal layers.

5.2.2 Effects on FPGA Design Tools

As an alternate means to cross-chip communication, the wire emulation system will affect many aspects of FPGA user design tools. An ideal tool set should abstract away the use of the wire emulation system. The three traditional components in a FPGA design flow are simulation, synthesis and vendor-specific implementation tools. Simulation tools typically use user HDL files or netlists to simulate a design. Synthesis tools translate user HDL files into a netlist of primitive logic (gates and FPGA-specific components) and nets. The vendor-specific implementation tools take netlists and user constraint files in generating a place and routed design, which then can be loaded onto an FPGA using a configuration bitstream. In addition, vendor-specific implementation tools can produce timing analysis results of a design. The wire emulation system might not affect synthesis tools, since the user will write HDL code without knowing the existence of this system, and assume that all nets are implemented using traditional, programmable interconnects. Optionally, a synthesis tool may incorporate knowledge of the wire emulation system if synthesis tool authors feel that this knowledge is necessary to produce optimized designs. Therefore, netlists would need the ability to instantiate wire emulation NIs and specify NI properties, such as latency and bandwidth requirements. Netlist support for NI instantiation would also be necessary if users wanted to explicitly instantiate a NIs.

In terms of vendor-specific implementation tools, the wire emulation system will affect place and route tools, since cross-chip communication costs change. Costs can be evaluated by delay, net importance, and congestion. Placement tools will continue to maintain locality within cores, but cores may have more placement options if cross-chip communication costs reduce. Routing tools will have to account for another set of interconnects (wire emulation system), which may have a lower cost than regular cross-chip interconnects. Critical paths will continue to route along shorter, faster wires. Wires with high, global fan-out, such as busses, will be good candidates for

the wire emulation system, assuming that the implementation of this FPGA enhancement efficiently emulates fan-out.

In addition, the routing tools will be responsible for generating schedules for wire emulation solutions that require them. This schedule will consider wire emulation operating frequency and therefore address power/performance ratio requirements. The scheduler will also compute worst-case delays for emulated wires. The resulting wire emulation system configuration could be loaded into the FPGA using a configuration bitstream and stored within the FPGA in configuration memory. Thus, the configuration bitstream generation tools would also need modification to account for the wire emulation system.

Effects on simulation tools will depend on the user abstraction level of the wire emulation system. If wire emulation use is abstracted from the user, then wire emulation timing information is only determined after place and route. Simulation is usually performed at the HDL level, so the design would lack knowledge of the wire emulation system. As a result, accurate simulation would require back annotating the design with timing information provided by the router.

If the user is able to instantiate the wire emulation system, then simulation tools that emphasize system-on-chip design could be appropriate, where system components can be simulated modularly and at various abstraction levels. In addition, supporting simulation at higher levels of abstraction allows for manageable simulator performance. One such simulation environment may involve SystemC modeling of system components and the wire emulation system interconnects these components. A wire emulation system model could be described at the transaction level, where transactions are based on deterministic characteristics of the wire emulation system. Clock-cycle accurate models would not be necessary since the transaction model reflects guaranteed performance metrics of the wire emulation system. The wire emulation system could be incorporated into a model library as a system bus, along with sub-component wrappers that bridge existing standard communication interfaces to the wire emulation system. In addition, standard communication interface wrappers would incorporate information regarding required throughput and latency, which the wire emulation system would be required to provide.

Given these wire emulation system and communication interface wrapper models, a user could create a system simulation by instantiating system component models and connecting them to the wire emulation system with communication interface wrappers. Timing requirements could then

be extracted from the communication interfaces to determine the wire emulation system configuration. The system floorplan could also be a factor in determining wire system configuration. If the wire emulation system can meet all system requirements, then simulation is allowed. Otherwise, the user is notified that communication requirements cannot be fulfilled.

Optionally, a wire emulation configuration generation tool could be offered, providing full user access to the wire emulation system. This tool could allow users to query and modify low-level wire emulation system configuration for the purpose of researching and developing guaranteed performance algorithms on an existing platform.

Chapter 6

Conclusions

6.1 Summary

This thesis described the concerns of FPGA architecture scaling, and showed that chip-length interconnections will not scale with manufacturing technology. Wire emulation was proposed to combat FPGA chip-length interconnection scalability and improve routing congestion. A prototype wire emulation system was demonstrated using an existing FPGA, investigating the potentials of a wire emulation system based on self-reconfiguration. Lastly, the potential benefits and effects of wire emulation systems on FPGAs were presented. The following summarizes the contents of each chapter.

Chapter 1 presented the motivation for an FPGA architectural enhancement, one that improves the performance and scaling of chip-length interconnects. Chapter 2 provided background on FPGA architecture and scaling theory. Also, Chapter 2 expanded on the motivation presented in Chapter 1, where scaling theory was applied to FPGA architecture, demonstrating existing cross-chip programmable interconnect delay and showing how it would worsen with smaller feature manufacturing technologies. In addition, background information was presented on wire emulation techniques that utilize standard manufacturing processes, including previous work on FPGA self-reconfiguration and SoC network-on-chip.

Then, Chapter 3 presented a conceptual wire emulation system for FPGAs, demonstrating that the wire emulation system adds another dimension to FPGA intra-communication. Since wire emulation systems can be implemented in a variety of ways, a methodology for wire emulation

system design was presented. Next, Chapter 4 presented a prototype wire emulation system that harnessed features of an existing self-reconfigurable FPGA. The prototype demonstrated the one-way communication between two cores using transport cores. These transport cores, implemented using FPGA programmable fabric, were not connected to one another using traditional FPGA programmable interconnects. Instead, communication between these transport cores was realized through the transfer of FPGA configuration memory.

Chapter 5 presented the results of the implemented prototype, which was implemented and verified on a Xilinx Virtex-II XC2V1000 part at a maximum operational speed of 46.7 MHz. Included in these results were performance metrics, such as communication latency and throughput. These performance metrics were also calculated for all members of the Xilinx Virtex-II family. Also, prototype power consumption was also measured, demonstrating that continuous self-reconfiguration increased current consumption by an acceptable 14%. Further more, results from the multi-frame write experiment were discussed, and performance calculations were derived for a broadcast version of the wire emulation prototype.

The prototype results were also analyzed in Chapter 5. It was demonstrated that throughput slightly increased while latency increased with FPGA size; both metrics were unacceptable for high performance designs. The narrow ICAP interface, in combination to FPGA configuration manager data padding requirements, proved to be major performance bottlenecks. FPGA configuration manager improvements that would greatly improve the prototype performance were also suggested. It was determined that even with FPGA configuration manager improvements, the reliance on configuration memory transfers would become a scalability issue because serialized access to configuration memory would poorly affect communication latency when supporting multiple transport modules.

Therefore, Chapter 5 proposed the investigation of an NoC-like wire emulation system that embodied parallelism, such that communication between multiple transport modules or NIs could be supported simultaneously. Using assumptions of current and future wiring technologies, NoC-like wire emulation system performance – latency and emulated wire throughput – were approximated. In addition, effects to programmable routing congestion were also examined. Finally, wire emulation system effects on FPGA layout and FPGA CAD tools were investigated.

6.2 Future Work

Network-on-chip (NoC) research has flourished in recent years, with NoCs touted as a viable global interconnection solution for SoCs. Much of their research can be applied in designing an NoC-like wire emulation system for FPGAs, where much future work is needed. NoC network characteristics such as topology, routing policies, and congestion avoidance will have to support a large number of NoC network interfaces. A large number of NoC NIs will be most beneficial in an FPGA, since NI accessibility will affect programmable routing congestion. User interfaces for NIs will also need investigation, determining the optimal number of emulated wires supported by each NI. Additionally, NoC link implementation will also need investigation. Chapter 5 suggested various link implementations, such as differential wires and wave pipelining. These wiring schemes and others will require further exploration, simulation, implementation, and experimentation. Ultimately, a link solution that balances propagation delay and power consumption must be determined.

With a wire emulation system supplementing FPGA architecture, FPGA CAD tools will have to be modified to reap the benefits of such a system. Chapter 5 previewed the effects on FPGA CAD tools, and discussed that the wire emulation system should be easy to use, from both FPGA user and FPGA CAD tool designer viewpoints. FPGA CAD tool modification can be made simpler by minimizing impact to existing FPGA layout regularity.

Another area of future work is FPGA architectures, and how revolutionary FPGA architectures will need to be developed to reduce power and increase performance to match the benefits provided by ASICs. This thesis discussed how local interconnects and CLB performance would scale with technology processes, but newer approaches will be needed to reduce the 20 to 1 logical overhead that FPGAs experience in comparison to ASICs. In addition, investigation of newer, non-volatile memories will be needed to reduce memory overheads and improve design security.

Lastly, Chapter 5 also mentioned FPGA configuration management improvements that would greatly benefit the prototype's performance. Reducing overheads in configuration memory access interfaces would aid future research in the areas of self-reconfiguration.

In summary, this thesis provided several contributions. First, a wire emulation concept for FPGA architecture enhancement was defined and developed. In addition, the benefits of wire emulation were analyzed quantitatively and qualitatively. Finally, this thesis demonstrated a working application of FPGA self-reconfiguration.

6.3 Conclusions

FPGA architecture will continue evolving to meet the performance and power requirements of a wide audience. The wire emulation system concept and methodology presented in this thesis will prove useful in the design of future FPGAs, and will help reduce chip-length interconnection overheads and congestion, and will improve scalability.

This thesis was successful in demonstrating a prototype wire emulation system on existing FPGA architecture. In fact, this prototype found use for FPGA resources that would have otherwise sat idle after FPGA initialization. While the performance results of the prototype were not sufficient for high performance communication, it revealed simple improvements to an existing FPGA configuration memory interface that would prove beneficial for this prototype, and for the self-reconfiguration research community.

Bibliography

[1] N. Tredennick and B. Shimamoto. "Go Reconfigure," *IEEE Spectrum*, vol. 40, issue 12, Dec. 2003, pp. 36-40.

[2] T. Taghavi, S. Ghiasi, A. Ranjan, S. Raje, and M. Sarrafzadeh. "Innovate or Perish: FPGA Physical Design," in *Proc. of the International Symposium on Physical Design (ISPD'04)*, Apr. 2004, pp. 148-155.

[3] R. Ho, K. Mai, and M. Horowitz. "The Future of Wires." *Proceedings of the IEEE*, vol. 89, no. 4, April 2001.

[4] B. S. Landman and R. L. Russo, "On Pin Versus Block Relationship for Partitions of Logic Circuits," *IEEE Transactions on Computers*, c-20: 1469-1479, 1971.

[5] Semiconductor Industry Association. *The International Technology Roadmap for Semiconductors, 2003 edition*. International SEMATECH: Austin, TX, 2003.

[6] R. Ho, K. Mai, M. Horowitz. "Efficient On-Chip Global Interconnects," in *Digest of Technical Papers, Symposium on VLSI Circuits*, June 2003, pp. 271-274.

[7] L. Shang, A. S. Kaviani, and K. Bathala. "Dynamic Power Consumption in Virtex-II FPGA Family," in *Proc. of 10th ACM International Symposium on Field-Programmable Gate Arrays (FPGA'02),* Feb. 2002, pp.157-164.

[8] Xilinx, San Jose, California. *Virtex-II Platform FPGAs: Complete Data Sheet*, http://direct.xilinx.com/bvdocs/publications/ds031.pdf, DS031 (v3.3) June 24, 2004.

[9] Xilinx, San Jose, California. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, http://direct.xilinx.com/bvdocs/publications/ds083.pdf, DS083 (v4.0) June 30, 2004.

[10] Xilinx, San Jose, California. *Virtex™ 2.5 V Field Programmable Gate Arrays*, http://direct.xilinx.com/bvdocs/publications/ds003.pdf, DS003-1 (v2.5) April 2, 2001.

[11] Altera, San Jose, California. *Stratix-II Device Handbook, Volume 1*, http://www.altera.com/literature/hb/stx2/stx2_sii5v1.pdf, v1.0, Feb. 2004.

[12] Altera, San Jose, California. *Stratix Device Handbook, Volume 1,* http://www.altera.com/literature/hb/stx/stratix_vol_1.pdf, v3.0, April 2004.

[13] C. Yiu, G. Swift and C. Carmichael. "Single Event Upset Susceptibility Testing of the Xilinx Virtex II FPGA," presented at 5th annual MAPLD International Conference,

Laurel, MD, September 10-12, 2002. [Online document], [cited August 3, 2004], Available at HTTP: http://parts.jpl.nasa.gov/docs/P29_yui.pdf.

[14] N. Tredennick and B. Shimamoto. "The Inevitability of Reconfigurable Systems," *ACM Queue*, vol. 1, no. 7, October 2003, pp. 34-43.

[15] The Internet Engineering Task Force, "Pseudo Wire Emulation Edge to Edge (pwe3) Charter," [Online document] Sept. 7, 2004, [cited Sept. 13, 2004], Available at HTTP: http://www.ietf.org/html.charters/pwe3-charter.html.

[16] R. Tessier, J. Babb, M. Dahl, S. Hanano, and A. Agarwal. "The Virtual Wires Emulation System: A Gate-Efficient ASIC Prototyping Environment," in the *Proc. of the 2nd International ACM/SIGDA Workshop on Field Programmable Gate Arrays*, Feb. 1994.

[17] M. F. Chang, V. P. Roychowdhury, L. Zhang, H. Shin, and Y. Qian, "RF/Wireless Interconnect for Inter- and Intra-Chip Communications," in *Proc. of the IEEE*, vol. 89, issue 4, Apr. 2001, pp. 456-466.

[18] I. O'Connor, "Optical Solutions For System-Level Interconnect," in *Proc. of the International Workshop on System-Level Interconnect Prediction (SLIP'04)*, Feb 2004, pp. 79-88.

[19] G. Brebner and A. Donlin. "Runtime Reconfigurable Routing," in *Proc. of Reconfigurable Architectures Workshop (RAW'98)*, 1998.

[20] A. Donlin. "Self Modifying Circuitry – A Platform for Tractable Virtual Circuitry," in *Proc. of 8th International Workshop on Field Programmable Logic and Applications (FPL'98)*, 1998.

[21] N. Steiner, "A Standalone Wire Database fore Routing and Tracing in Xilinx Virtex, Virtex-E, and Virtex-II FPGAs," Master's Thesis, Virginia Tech, Aug. 2002.

[22] W. J. Dally and B. Towles. "Route Packets, Not Wires: On-Chip Interconnection Networks," in *Proc. of 38th Conference on Design Automation (DAC 2001)*, June 2001, pp. 684-689.

[23] L. Benini and G. De Micheli. "Networks on Chips: A New SoC Paradigm," *IEEE Computer*, vol. 35, issue 1, January 2002, pp. 70-78.

[24] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, A. Sangiovanni-Vincentelli. "Addressing the System-on-a-Chip Interconnect Woes Through Communication-Based Design," in *Proc. of the 38th Conference on Design Automation (DAC 2001)*, June 2001, pp. 667-672.

[25] D. Wingard. "MicroNetwork-Based Integration for SoCs," in *Proc. of the 38th Conference on Design Automation (DAC 2001)*, June 2001, pp. 673-677.

[26] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, C. A. Zeferino. "SPIN: a Scalable, Packet Switched, On-chip Micro-network," in *Proc. of the Design, Automation and Test in Europe Conference (DATE) Designers' Forum*, March 2003, pp. 70-73.

[27] P. Guerrier and A. Greiner. "A Generic Architecture for On-Chip Packet-Switched Interconnections," in *Proc. of the Design, Automation and Test in Europe Conference (DATE)*, 2000, pp. 250-256.

[28] J. Dielissen, A. Rădulescu, K. Goossens, and E. Rijpkema. "Concepts and Implementation of the Phillips Network-on-Chip," International Workshop on IP-Based System-on-Chip Design, November 2003.

[29] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Öberg, K. Tiensyrja, and A. Hemani. "A Network on Chip Architecture and Design Methodology," in *Proc. of the IEEE Computer Society Annual Symposium on VLSI*, April 2002, pp.117-124.

[30] J. Liu, M. Shen, L.-R. Zheng, H. Tenhunen. "System Level Interconnect Design For Network-on-Chip Using Interconnect IPs," in *Proceedings of the 2003 International Workshop on System-level Interconnect Prediction (SLIP'03)*, 2003, pp. 117-124.

[31] D. Pamunuwa, J. Öberg, L. R. Zheng, M. Millberg, A. Jantsch, and H. Tenhunen. "Layout, Performance and Power Trade-Offs in Mesh-Based Network-on-Chip Architectures," in *Proc. of the 12th IFIP International Conference on Very Large Scale Integration (VLSI-SoC 2003)*, Dec. 2003, pp. 362-367.

[32] M. Millberg, E. Nilsson, R. Thid, A. Jantsch. "Guaranteed Bandwidth using Looped Containers in Temporally Disjoint Networks within the Nostrum Network on Chip," in *Proc. of the Design, Automation and Test in Europe Conference and Exhibition Volume II (DATE'04)*, Feb. 2004, pp. 20890-20895.

[33] E. Rijpkema, K. G. W. Goossens, A. Rădulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. "Trade Offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip," in *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, March 2003, pp. 10350-10355.

[34] S. G. Pestana, E. Rijpkema, A. Rădulescu, K. Goossens, and O. P. Gangwal. "Cost-Performance Trade-offs in Networks on Chip: A Simulation-Based Approach," in *Proc. of the Design, Automation and Test in Europe Conference and Exhibition Volume II (DATE'04)*, Feb. 2004, pp. 20764-20769.

[35] R. Thid, M. Millberg, and A. Jantsch. "Evaluating NoC Communication Backbones with Simulation," in *Proceedings of the IEEE NorChip Conference*, November 2003.

[36] R. J. Fong, S. J. Harper, and P. M. Athanas. "A Versatile Framework for FPGA Field Updates: an Application of Partial Self-Reconfiguration," in *Proc. of 14th IEEE International Workshop on Rapid Systems Prototyping*, June 2003, pp. 117-123.

[37] Xilinx, San Jose, California. *Dynamic Reconfiguration of RocketIO MGT Attributes*, http://www.xilinx.com/bvdocs/appnotes/xapp660.pdf, XAPP660 (v2.2) February 4, 2004.

[38] Xilinx, San Jose, California. *In-Circuit Partial Reconfiguration of RocketIO Attributes*, http://www.xilinx.com/bvdocs/appnotes/xapp662.pdf, XAPP662 (v2.4) May 26, 2004.

[39] B. Blodget, S. McMillan, and P. Lysaght. "A lightweight approach for embedded reconfiguration of FPGAs," in *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, March 2003, pp. 399-400.

[40] D. P. Schultz, L. C. Hung, and F. E. Goetting. "Method and Structure for Configuring FPGAs," U.S. Patent no. 6,204,687, March 2001.

[41] Xilinx, San Jose, California. *Virtex FPGA Series Configuration and Readback*, http://www.xilinx.com/bvdocs/appnotes/xapp138.pdf, XAPP138 (v2.7), July 11, 2002.

[42] Xilinx, San Jose, California. *Virtex Series Configuration Architecture User Guide*, http://www.xilinx.com/bvdocs/appnotes/xapp151.pdf, XAPP151 (v1.6), March 24, 2003.

[43] Memec, LLC, "Virtex-II™ LC1000 Development Kit," [Online document] MDS008-03 Feb. 10, 2003, Available at HTTP: http://www.insight.na.memec.com/Memec/iplanet/link1/Virtex-IILC1000_1.pdf.

[44] Xilinx, San Jose, California. *RocketIO™ Transceiver User Guide*, http://direct.xilinx.com/bvdocs/userguides/ug035.pdf, UG035 (v1.4) June 29, 2004.

[45] J. Xu and W. Wolf, "A Wave-Pipelined On-chip Interconnect Structure for Networks-on-Chips," *Hot Interconnects: 11th Symposium on High Performance Interconnects (Hot-I 2003)*, August 2003, pp. 10-14.

Vita

Ryan Fong was born on a snowy day in October 1979 in Towson, MD. After graduating from Dulaney High School in Timonium, MD in 1997, he enrolled at Virginia Tech to pursue an undergraduate degree in engineering. Although his first inclination was to transfer into electrical engineering track after his first semester, it was his interest in hardware-software interaction that led him to choose computer engineering. As a result of his hard work and dedication, Ryan received a Bradley Scholarship from the Bradley Department of Electrical and Computer Engineering at Virginia Tech. As an undergraduate, he participated in many student engineering projects, societies, and student government. In the spring of 2001, Ryan graduated *summa cum laude*, completing his B.S. degree in Computer Engineering with a minor in Economics. Prior to completing his B.S. degree, Ryan enrolled into the combined M.S./B.S. Computer Engineering degree offered by Virginia Tech and joined the Configurable Computing Laboratory (CCM Lab) as a research assistant. As a graduate student, he continued work at the CCM Lab and received a Bradley Fellowship from the Bradley Department of Electrical and Computer Engineering at Virginia Tech. After two years of pursuing his M.S. degree in Computer Engineering, he left academia and joined EVI Technology in Columbia, MD as an embedded systems engineer. After an enjoyable one-year stay at EVI, Ryan decided to go back to Virginia Tech to complete his M.S. degree. He will resume his job at EVI after completing his M.S. degree.