PROGRAMMING MODEL FOR NETWORK PROCESSING ON AN FPGA

A Thesis Presented

by

ERIC ROBERT KELLER

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

February 2005

Electrical and Computer Engineering

PROGRAMMING MODEL FOR NETWORK PROCESSING ON AN FPGA

A Thesis Presented

by

ERIC ROBERT KELLER

Approved as to style and content by:

Russell G. Tessier, Chair

Tilman Wolf, Member

Lixin Gao, Member

Seshu B. Desu, Department Chair
Electrical and Computer Engineering

ACKNOWLEDGEMENTS

First, I would like to thank Dr. Russell Tessier for advising me throughout the entire process. His guidance and advice was invaluable and helped to make this thesis be of higher quality than I thought could be possible. I would also like to thank Dr. Tilman Wolf and Dr. Lixin Gao for providing their feedback and serving as thesis committee members.

I would like to thank Xilinx, Inc. for their generosity in funding my education. This allowed me to both work full time and further my education. I would like to thank my manager at Xilinx, Dr. Gordon Brebner, for many reasons. He provided me with the necessary flexibility to juggle the demands of both work and school. He was also instrumental in guiding the research direction for this thesis. I would also like to thank my colleague at Xilinx, Dr. Phil James-Roxby. He has taught me a lot, helped me technically on this thesis, and has been a good friend. I would also like to acknowledge the contributions of many others at Xilinx. In particular, Graham Schelle for his many achievements over the past two summers that really helped bolster the technical content of this thesis, Dr. Chidamber Kulkarni for his insights on Click, among other things, and Craig Cholvin for his help with the Xilinx back end tools.

I would also like to thank three people who have been integral in my career so far and have immensely contributed, indirectly, to this thesis. Dr. Mark Jones and Dr. Peter Athanas at Virginia Tech introduced me to the world of FPGAs. Dr. Steven Guccione introduced me to research and taught me many things about FPGAs and the world of commercial research. He also provided much needed encouragement throughout the entire process.

As a final and most important acknowledgement, I would like to thank my wife Kristen for being so supportive in every possible way.

ABSTRACT

PROGRAMMING MODEL FOR NETWORK PROCESSING ON AN FPGA

February 2005

ERIC ROBERT KELLER

B.S., VIRGINIA TECH

Directed by: Professor Russell G. Tessier

The increasing size, performance, and feature set of Field-Programmable Gate Arrays (FPGAs) have led to their adoption for many applications. The tremendous speedup over software and the flexibility advantages over ASICs enable FPGAs to provide a solution that offers a compromise. One particular area that is rapidly growing is network processing. The expansive growth of network enabled systems coupled with the variety of applications make FPGAs an ideal technology to use. Presented in this thesis is a programming model that provides a framework for developing networking applications that make use of FPGAs. The programming model abstracts the resources of the FPGA in terms of resources that are more suitable to the networking space. This abstraction then allows domain specific development tools to make use of FPGAs with less effort by bridging the high-level nature of the development environment and the low-level nature of the FPGA.

TABLE OF CONTENTS

ABSTRACT v

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTER Page

1 INTRODUCTION 1

2 BACKGROUND 5

2.1 Network Processing on FPGAs 5
2.2 Tools for FPGAs 5

2.2.1 Hardware Description Languages and Tools 5
2.2.2 Structural High-Level Language 6
2.2.3 Behavioral High-Level Language 7
2.2.4 Domain Specific Languages/Tools 7

2.3 FPGAs in Networked Systems 8

2.3.1 Routing and Switching 9
2.3.2 Protocol Boosters 10
2.3.3 Security 11
2.3.4 Web Server 12

2.4 Summary 12

3 PROGRAMMING ABSTRACTIONS 14

3.1 Motivation 14
3.2 Design flow 15
3.3 Description of Programming Interface 19

3.3.1 Threads 20
3.3.2 Included Intellectual Property 23
3.3.3 Interfaces 24
3.3.4 Memories 26
3.3.5 Inter-Thread Communication 33
3.3.6 Thread Synchronization 36

4 COMPILATION 37

4.1 Hardware Generation 37
4.2 Top Level Design 38

4.2.1 Interface 39

4.2.2 Signals 41
4.2.3 Synchronization Logic 43
4.2.4 Instatiate Components 44

4.3 Mapping Threads to Hardware 48

4.3.1 Interface 49
4.3.2 Signals 52
4.3.3 Control 54
4.3.4 Combinatorial and Synchronous Processes 55

4.4 Bitstream Generation 67

5 HIGHER LEVEL TOOLS 68

5.1 Mapping Higher Level Tools to FPGAs 68
5.2 Click 68

5.2.1 Sub-graph Pattern Match and Replace 73
5.2.2 Move Elements 74
5.2.3 Split Paths 75
5.2.4 Merge Elements 75
5.2.5 Run Elements in Parallel 78

5.3 Teja 79

5.3.1 Teja Application Design Environment 79

5.3.1.1 Software Library 80
5.3.1.2 Software Architecture 81
5.3.1.3 Hardware Architecture 81
5.3.1.4 Hardware Mapping 82

5.3.2 Mapping Teja to FPGAs 82

5.3.2.1 Software Library 85
5.3.2.2 Software Architecture 86
5.3.2.3 Hardware Architecture 86
5.3.2.4 Hardware Mapping 87

5.4 Summary 88

6 EXPERIMENTAL APPROACH 89

6.1 Methodology 89
6.2 Gigabit Ethernet to Aurora Bridge 89

6.2.1 Gigabit Ethernet 90
6.2.2 Aurora 91
6.2.3 Architecture 92

6.3 Remote Procedure Call 94

6.3.1 Implemented Design 96
6.3.2 Architecture 97
6.3.3 Component Descriptions 99
6.3.4 Improvements for the RPC Implementation 101

6.4 IP Router 101

6.4.1 Implemented Design 101
6.4.2 Architecture 102

6.5 Network Address Translation 106

6.5.1 Implemented Design 107

7 RESULTS 109

7.1 Introduction 109
7.2 Gigabit Ethernet to Aurora Bridge 110
7.3 Remote Procedure Call 111
7. 4 Click Comparisons 112

8 CONCLUSIONS AND FUTURE WORK 116

8.1 Conclusions 116
8.2 Future Work 117

APPENDIX: LANGUAGE REFERENCE 119

A.1 Language 119
A.1 Details of Node Types 120
A.3 Implementation with XML 127

BIBLIOGRAPHY 129

LIST OF TABLES

Table Page

7.1 Comparison of XML implementation of the Aurora to gigabit Ethernet bridge using the programming interface from this thesis and the VHDL implementation from James-Roxby.....111

7.2 Comparison of XML implementation of the RPC protocol using the programming interface from this thesis and the software implementation as part of Linux. 112

7.3 Comparison of implementations of the Click designs IP router and NAT. 114

7.4 Comparison of XML implementation of the 16-port IP router using the programming interface from this thesis and the implementation by Shah.....115

A.1 Details the attribute values for the operation element type.....126

A.2 Details the attribute values for the condition element type.....127

LIST OF FIGURES

Figure Page

1.1 Design flow from XML description of the application to a configuration bitstream. 3

3.1 Abstraction layers for the programming model. 16

3.2 Design Flow. 19

3.3 Graphical representation of the available constructs. 20

3.4 Example thread definition for interfacing to the gigabit Ethernet's receive (RX) port. 23

3.5 Example definition of an instantiation of included Intellectual Property. 24

3.6 Example syntax used to include and interface and associate with a thread. 26

3.7 Example syntax to define a memory as well as attaching it to thread. 27

3.8 Diagram of implementation of locking in the Shared Memory element. 29

3.9 Structure of DPMem memory element's data access. 30

3.10 Structure of DPMem memory element's buffer allocation. 32

3.11 Structure of DPMem memory element's buffer deallocation. 33

3.12 Example of explicit inter-thread communication. 34

3.13 Example syntax to define a channel as well as connect the thread to the channel. 35

4.1 Design Flow. 37

4.2 Structure of top-level generated VHDL file. 39

4.3 XML code for interface. 40

4.4 VHDL code for interface. 40

4.5 XML code for signals. 42

4.6 VHDL code for signals. 43

4.7 VHDL code for synchronization. 44

4.8 VHDL code for interface component instantiation. 45

4.9 XML code for instantiation of thread. 46

4.10 VHDL code for instantiation of thread. 47

4.11 Example of clock domain determination. 48

4.12 Structure of generated VHDL file for threads. 49

4.13 VHDL code for interface of thread. 51

4.14 VHDL code for interface of thread. 51

4.15 XML code for signals of thread.....53
4.16 VHDL code for signals of thread. 53
4.17 VHDL code for control. 55
4.18 VHDL code for structure of combinatorial process.....56
4.19 VHDL code for structure of synchronous process. 57
4.20 XML code and VHDL code for general operations. 58
4.21 XML code and VHDL code for synchronization operations.....59
4.22 XML code and VHDL code for memory operations.....60
4.23 XML code and VHDL code for channel put operations.....61
4.24 XML code and VHDL code for channel get operations.....62
4.25 XML code and VHDL code for conditionals. 63
4.26 XML code and VHDL code for transitions. 64
4.27 Circuitry used to support Alias signals.....65
4.28 Timing Diagram for reading from memory.....66
4.29 FPGA back end tools design flow.....67
5.1 Click graph of a 2 port IP router.....69
5.2 Summary of process to map Click to an FPGA using the XML based language presented in this thesis. 70
5.3 XML based implementation of a two port IP router.....72
5.4 Transformed Click graph for the two port IP router using sub-graph pattern match and replace.....74
5.5 Transformed Click graph for the two port IP router using technique to move elements.....75
5.6 Transformed Click graph for the two port IP router using technique to split paths.....75
5.7 Transformed Click graph for the two port IP router after merging elements. 77
5.8 Transformed Click graph with implementation details for elements executing in parallel for the two port IP router.....79
5.9 Summary of process to map Teja to an FPGA using the XML based language presented in this thesis. 84
6.1 Gigabit Ethernet Interface. 91
6.2 Aurora Interface 92
6.3 Gigabit Ethernet to Aurora Bridge Architecture. 94
6.4 A step-by-step explanation of an RPC call.....95
6.5 Architecture of RPC implemented on FPGA. 98
6.6 Architecture of a 2-Port IP Router implemented on an FPGA. 103

6.7 Architecture of 16-port IP Router implemented on an FPGA. 105
6.8 Diagram of click graph of the NAT application 106
6.9 Architecture of NAT implemented on an FPGA 108
A.1 Graphical view of programming language. 120
A.2 Example XML Syntax. 128
A.3 Sample XML code making use of specific tags. 128

CHAPTER 1

INTRODUCTION

With the explosive growth of the Internet, it is becoming more and more common for a system to be network enabled. Due to the usefulness of systems being able to communicate with one another, many new types of end systems have begun to appear beyond the traditional workstations and personal computers. Appliances such as phones, picture frames, and game consoles are all examples of devices that have network capabilities. While networking is mainly routing and switching between hosts and routers, the requirements of the network infrastructure and networking capabilities varies from application to application. One application, such as streaming video, has one set of requirements while another application, such as sensor networks, has a completely different set of requirements. This variety of requirements demonstrates the need for a programming model that can facilitate ease of design of these systems while achieving the necessary requirements.

Field Programmable Gate Arrays (FPGAs) can provide an ideal environment for implementing a diversity of protocols and requirements. The parallelism provided by implementing protocols in hardware as well as the customizability of the FPGA provides great capabilities. This technology provides configurable hardware, thus making it flexible enough to implement many protocols as well as providing performance that can be orders of magnitude better than software based counterparts.

One problem, however, is that designing for an FPGA typically requires hardware design. This is commonly considered a difficult task. To handle this difficulty requires design tools. One particular approach to design tools revolves around domain specific programming languages or tools. These are languages or tools that appeal to a specific application domain by using constructs and design flows that are familiar to the designer.

While domain specific languages can provide benefits, mapping to FPGAs is still a complex task. Therefore, this makes the compilation tools very complex. An alternate approach is to use an intermediate platform that the tools can target. This platform is an abstraction above the FPGA that will enable simpler mappings. A separate tool will then map the description, which is in terms of the platform resources, to the FPGA.

Typically the platform is domain specific. The platform can be presented in terms of resources that will make the compiler's task simpler. This means that the resources are closely related to the user's view. In the networking space, past research has demonstrated one such platform.

This thesis addresses a programming model that will incorporate the flexibility and performance of FPGAs, the needs of domain specific languages, and the research that has shown an abstraction model particularly suitable to network applications. The contribution, as presented in this thesis, is an application programming interface (API) that will allow tools to map to an FPGA by making use of the abstractions provided. The API has both a programming language aspect to it as well as a compiler aspect to it. The API is used by the user to specify the functionality and architecture of the application in a form resembling a programming language. The API also includes the ability to compile the description and generate a hardware description language representation of the design that can then be compiled to hardware by existing back-end tools. The abstractions provided to the designer using the API include threads for control, communication between elements in the system, and memory for storage of the packets. The API allows the tools to not only define the functionality of the system, but also the architecture. Defining the architecture involves the structure of each of the elements as this is not fixed. For example, there is not a standard bus that is already implemented that must be used. Instead a user can choose to have elements within the FPGA communicate over a bus or using point-to-point connections. The API is also presented as an intermediate textual format using XML. This provides a more flexible design entry mechanism as it is programming language independent and

can be hand coded. The use of XML forms the main focus of the grammar of the programming language. The flow focused on in this thesis is shown in Figure 1. The user specifies a design in XML using the grammar created for this thesis. The XML is then compiled to a hardware description by a tool created for this thesis. Finally, the back-end tools provided by the FPGA vendor are used to generate the configuration bitstream for the FPGA.

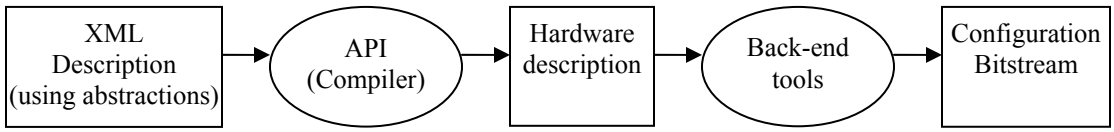

Figure 1.1 Design flow from XML description of the application to a configuration bitstream.

The programming interface presents the user with a model of a number of micro-engines that can perform operations on data. The data in this case is typically parts of a packet. However, taking advantage of the flexibility of FPGAs, the micro-engines are not typical sequential microprocessors. Instead, the compiler automatically generates custom hardware to implement the functionality described by the designer using the language of the API. The communication between the threads as well as the memory architecture are both programmable. While there is no analogy to being able to define the architecture, an analogy can be made for the functionality. The programming interface presented in this thesis can be thought of as providing an assembly language for the underlying technology. The domain specific languages are analogous to programming in high-level languages. This analogy makes sense because the assembly language hides the implementation of the underlying technology. In this case, the underlying technology is the FPGA.

To demonstrate the capabilities of the programming interface and to evaluate the efficiency of the compilation process, four applications were implemented. The first is a bridge between Aurora, a Xilinx proprietary point-to-point link-layer protocol, and gigabit Ethernet, a link-layer

protocol standardized by the IEEE. The second is a remote procedure call server. The third is an IP router. The design is simplified to match a previous implementation that was created using a higher-level tool to allow for a comparison. The fourth was a network address translation (NAT) device. Like the router, the specification of the design was chosen to match a previous implementation. All four were implemented directly using the intermediate textual format, XML. The programming interface presented in this thesis was then used to automatically generate the necessary hardware. They were tested and verified in actual hardware. Analysis of the timing reports show that the required performance of gigabit line rates was achieved. When compared to implementations using a hardware description language, the performance and area were roughly the same for the bridge. For the RPC server, the FPGA implementation achieved a speedup 8.7 over a Linux workstation with a 2 GHz Pentium processor. The final two designs, the IP router and NAT, achieved similar throughput to previous implementations. However, the area and latency were greatly reduced.

The thesis is organized into 8 chapters. Chapter 1 gives an introduction into the problem being addressed and the approach to solving the problem. Chapter 2 gives background information on tools for creating FPGA designs as well as an overview of networking applications that have been implemented on FPGAs. Chapter 3 introduces the user interface of the programming model along with the design abstractions. Chapter 4 then details how the design abstractions presented in chapter 3 are mapped to an FPGA. Chapter 5 presents an overview of how two high level tools could make use of the programming model presented in this thesis for mapping to FPGAs. Chapter 6 describes several applications that were implemented to demonstrate the capabilities of the programming interface. Chapter 7 then presents the results of the applications in terms of performance and area and compares them to existing implementations on various technologies. Chapter 8 presents conclusions and future work.

CHAPTER 2

BACKGROUND

2.1 Network Processing on FPGAs

As this thesis is focused on enabling network processing on FPGAs, this chapter discusses previous research in the area. This includes an overview of design tools that can be used to design for FPGAs as well as networking applications that have been implemented on FPGAs.

2.2 Tools for FPGAs

While the advantages of configurable logic are numerous, a barrier to using them still remains. The design process is considered difficult as it is geared towards hardware design. This is often considered difficult due to the unfamiliarity of many engineers used to designing with software methodologies. Due to the difficulty there have been many attempts to simplify the design process. Discussed in this section are some of the example classes of design techniques.

2.2.1 Hardware Description Languages and Tools

Hardware description languages (HDLs) are the most common method for designing for FPGAs today. Languages such as VHDL [5] and Verilog [6] include constructs that are common to hardware design. High-level languages such as C are inherently sequential. They define the behavior of the application as a series of instructions. Hardware, on the other hand, is inherently parallel. It is commonly defined in terms of logic gates connected together by wires. These correspond to language features found in HDLs, namely signals and modules. Signals are the equivalent of variables in high-level languages. Instead of representing a memory location, as in high-level languages, in HDLs they represent wires. Modules are blocks of logic with a given

interface. The interface is the wires that are inputs to the logic block and the wires that are the outputs of the logic block.

A special type of wire is a clock. Clocks are a central design element in hardware systems as it is the most common way to design hardware. As such, HDLs include language features allowing the capability of expressing functionality synchronized to clock edges.

An additional feature of HDLs is that they can include information about timing in the design to allow for a more accurate simulation. Delay on wires and through logic is a finite non zero value. Adding real values to the delays provides accurate simulations of hardware behavior. While HDLs were initially designed as simulation languages, the improved quality in logic synthesis has enabled them to be used as design languages as well.

2.2.2 Structural High-Level Language

A new class of design languages has arisen to take advantage of features in high-level languages. These languages provide the same capabilities as HDLs. However, they do so using a high-level language such as Java. Like HDLs, they specify the application in terms of wires and modules. The advantage of doing this is twofold. First, it allows designers to use a familiar language rather than learning a new one. While the design style may be unfamiliar, the syntax of the language remains the same. The second reason is that it allows for additional features beyond simply hardware design. For example, taking advantage of looping may allow for easier parameterization.

One example language is JHDL. With JHDL, Java is used to design for FPGAs [7]. In addition to the capabilities for designing hardware, JHDL adds a debugging environment that has added benefits that could not be achieved with a hardware description language. Likewise, JBits [8], through RTPCores [9], is another example of being able to design hardware with a language like Java. JBits is similar to JHDL in that it provides an API for defining hardware modules and connecting them together with wires. However, the additional capabilities that JBits provides

include the ability to design a run-time reconfigurable application. That is, an application where the design can be altered during execution.

2.2.3 Behavioral High-Level Language

An alternate approach to using high-level languages commonly used for software is to provide a software design environment. Here, the behavior of the design is defined as a software program. A compiler then compiles the description of the design to hardware. This allows software designers the ability to design hardware. Celoxica's DK1 toolkit allows a user to describe an application in a language called Handel-C [10]. Handel-C has C like syntax but is modified to include constructs more applicable to hardware. Parallelism is explicitly specified through special tags. Xilinx's Forge took a different approach while still compiling a high-level language to hardware [12]. Instead of introducing hardware constructs into a commonly used language like Handel-C has, Forge uses pure Java code as the input. The compiler will attempt different levels of parallelism by mapping different possibilities to the hardware.

2.2.4 Domain Specific Languages/Tools

Instead of appealing to software engineers in general, a class of languages aims to appeal to domain experts. These domain specific languages, including tools, provide a design environment specific to the domain. One example in the networking space is CLIFF [3]. CLIFF is a tool that takes a description of an application written in Click [13] and maps the design to an FPGA. While Click was originally designed as a tool for designing modular routers implemented on a Linux workstation, its use is more general. It includes a set of elements that are common to networking applications. One example element is the IPClassifier element which will inspect the header of an IP packet and classify the type of segment that is contained in the payload, for example a TCP segment. Another example element is the DecTTL element which will decrement the time to live field in an IP packet. This is an operation that must be done at each hop in a

route. The designer then specifies the use and connectivity of these elements using the Click scripting language. In the original workstation implementation the elements were written in C++. The functionality and interface were described in these classes. In the FPGA implementation, CLIFF, the elements are written in Verilog with a standard interface. A script then takes the Click description of the design and stitches the elements together by creating a top level Verilog file.

Another example of a domain specific language that has been mapped to FPGAs is Snort. Snort is a language and database that is used for creating rules used in network intrusion detection systems [45]. The Snort language is organized as a series of rules that specify a regular expression that is used to match against a specified field of a specified protocol. Also included in the description using the Snort language is an action that is to be taken when a received packet matches one of the rules, such as drop the packet. Several systems have been created for compiling the regular expressions in Snort to FPGA based systems [42][43][44][61].

A final example domain specific language is Ponder. Ponder is a language for specifying security policies that is independent of implementation [35]. Like Snort, Ponder includes a mechanism for specifying rules and corresponding actions to be taken for that rule. A subset of Ponder targeting firewall applications has been mapped to an FPGA [36]. A compiler creates an optimal hardware implementation for performing the rule matching. It does this by first performing a rule reduction technique that will eliminate redundancy and share resources based on similarity of the rules. The reduced rules are then used to generate a VHDL implementation of the hardware.

2.3 FPGAs in Networked Systems

While earlier FPGAs were limited to use as glue logic, recent advances in speed, size, and features have given FPGAs the capability to perform networking related functionality. There exists much literature on different networking applications that have been mapped to FPGAs.

Some applications simply use the FPGA as an ASIC replacement while others make use of the flexibility and reprogrammable nature of FPGAs. The variety of applications is great and the applications are not limited to a certain functional class or layer in the layered network architecture model. A few examples are discussed here to demonstrate the use of FPGAs in networked systems.

2.3.1 Routing and Switching

One of the main classes of networking functions includes routing. Routing is used to direct messages through a network to the correct destination. The implementation of a typical router includes lookup and switching. Lookup algorithms and implementations have many forms. One example is a content addressable memory (CAM), as demonstrated on an FPGA by Ditmar [29]. Here the CAM is presented with an IP address and the CAM will output a tag indicating the result of the lookup. Another method is to use tree based lookup implemented in hardware, as shown by Lockwood [15]. The functionality is similar in input and output, but the internal structure is more efficient.

Switching has also been implemented on an FPGA. Using reconfiguration, Young was able to create a parameterizable crossbar switch that made use of run-time reconfiguration [11] to enable it to fit on a Xilinx XC2V6000 device. The size ranged from a 1024 by 1024 switch operating at 155.52 Mbps to a 16 by 16 switch operating at 9.952 Gbps. Latency was fixed in each switch and was at most 22 cycles for the largest number of inputs.

Brebner implemented an IP router targeted at a platform FPGA that was capable of handling IP version 4 (IPv4) traffic as well as IP version 6 (IPv6) traffic [1]. This router, named MIR, was a four port router that handled straightforward through routing, encapsulation of packets within packets, and conversion of packets between IP versions. The function performed depends on the packet type as well as the destination. Implemented using a Xilinx Virtex-II Pro FPGA with multi-gigabit transceivers, each of the ports operated at gigabit Ethernet line rates.

The routing function was able to be achieved with a latency of zero since the packet was processed and ready to transmit as soon as the final byte arrived. Here, the definition for store-and-forward devices was used to define latency as the time from when the last bit arrives to when the first bit is transmitted [37].

2.3.2 Protocol Boosters

Another interesting application implemented on FPGAs was Hadzic's protocol boosters [16]. These were additional functions added to a protocol that can improve the performance of the protocol. They are used in between two end systems at any point in the network. Examples of protocol boosters include encryption/decryption, compression, and error correction coding.

Encryption can be used to securely transmit data across an insecure network. This can be used, for example, in virtual private networks where employees of a company can access sensitive company information from a home computer over the Internet. Due to the high level of parallelism, encryption has been shown to be an application well suited to FPGAs. Patterson demonstrated an implementation of the data encryption standard (DES) that achieved a 10.7 Gbps encryption rate on a Xilinx XC2V400 FPGA [4]. This was faster than all other publicly announced implementation at the time of publication, including an ASIC that could achieve a 9.28 Gbps encryption rate. Bellows, et al, presented a PCI card with four Xilinx XC2V1000 FPGAs on it that was used as an offload engine for various aspects of the IP security protocol [34]. Example accelerators include the secure hashing algorithm, SHA-1, and the advanced encryption standard, AES. The performance of the SHA-1 accelerator was 1.06 Gbps and the performance of the AES accelerator was 1.14 Gbps.

Compression can be used to boost the performance of networked communication. By using extra processing at the client and server, compressed files can be transferred to compensate for bandwidth limitations. This is currently used by dial-up internet service providers such as America Online [39] and United Online [38]. Huang, et al, implemented a Lempel-Zev

compression algorithm on four Xilinx XC4036XLA FPGAs [33]. The design achieved 128 Mbps compression rate with a clock speed of 16 MHz. They compared this to a software implementation using a 450 MHz Pentium II processor that was able to achieve 4.8 Mbps. Since the FPGA implementation required four chips, the Pentium system should be scaled to 19.2 Mbps to provide a more fair comparison. That still represents a 6.7x speedup for the FPGA implementation over the processor system that was obtainable due to the high parallelism achievable even with a clock speed that was 28 times slower.

Error correction is a technique used for sending data over noisy channels. The error correction algorithm allows for some of the data corrupted by the channel noise to be recovered. This provides for more reliable communication. Intellectual property cores are available from FPGA vendors for popular error correction algorithms. For example, Xilinx offers a turbo decoder for a Virtex-II FPGA that can operate at 2 Mbps [41]. Liang, et al, created a power efficient implementation of the algorithm using run-time reconfiguration [40]. The power efficient turbo decoder required only 248 mW compared to the 970 mW from the Xilinx core while only reducing the throughput to 1.4 Mbps,

2.3.3 Security

With the increasing number of hackers and viruses, security is becoming a bigger issue. Protection on local networks is necessary. Firewalls provide one line of defense by only allowing accepted traffic. McHenry demonstrated a firewall implemented in an FPGA that could operate at line rates [17]. Virus detection and elimination provides another line of defense. This is typically implemented as a software program that scans all downloads from the Internet [18][19]. As Lockwood demonstrates, the use of FPGAs is ideal for this task as each of the virus definitions can be checked in parallel [14]. This enables virus detection at hardware speed and can be used at any gateways between the internal local area network and the external Internet. Hutchings, et al, also implemented regular expression matching, central to network intrusion

detection systems, on an FPGA [42]. Their regular expression compiler used the Snort rule database as an input and an EDIF netlist for an FPGA implementation was generated. The compiler was implemented in Java and therefore they were able to generate the hardware using the JHDL API. This allowed complex circuitry to be generated in response to the database without having to also create the underlying netlist generation software. For a 10 MB data set and a 4971 character regular expression, the implementation by Hutchings, et al, was able to achieve a throughput 455.8 times greater than a software implementation. Baker and Prasanna similarly used the Snort rule base to generate hardware for implementation on an FPGA [43]. Instead of focusing on the efficiency of a single match unit, they instead focused on the efficiency of the entire system. They did this by performing graph optimizations on the entire rule set to allow for sharing of redundant match units. By doing this, they were able to achieve a throughput/area performance improvement of 2X over other implementations.

2.3.4 Web Server

Another application that is almost exclusively implemented in software is a Web server. Visiting a web site involves a web browser opening a TCP/IP connection with the server. Then, using the HTTP protocol, the browser requests a certain web page or file. The server then fetches the page and returns it to the browser. Fallside was one of the first to suggest that TCP/IP can be done on FPGAs and thus enabling connections to the Internet [20]. The demonstrated use was through an FTP server. Wincom Systems makes a commercially available Web server based solely on FPGAs [21]. It has been used to demonstrate a speedup that is 50 to 300 times the performance of Intel or Sun based servers.

2.4 Summary

The flexibility and potential for parallelism on a FPGA have proven it to be a useful platform for implementing the functionality associated with network processing. Different

languages exist that target different classes of users. This thesis focuses on the domain specific languages targeting network processing. Presented is a platform and associated language that provides an intermediate target for the higher level languages in order to reduce the design effort by bridging the gap between high-level description and low-level implementation. This platform and associated language allows the ability to define network processing applications that can then be automatically mapped to an FPGA. Shown in this chapter were many example applications that made use of FPGAs to provide tremendous speedups or improved functionality. This thesis focuses on the protocol handling aspect of the network processing and is not a target for implementing data flow oriented applications such as encryption. However, as they are an integral part of networking, two mechanisms exist where the protocol handling along with the data flow oriented functionality can be integrated. The first is a mechanism to include externally defined functions in the description of the system using the language presented in this thesis. The second is a use case where the language presented in this thesis is used to define a sub-system rather than the entire application. A "core" is generated for use in larger applications. In both cases, the ability to implement the protocol handling aspect efficiently provides a common mechanism across many tools.

CHAPTER 3

PROGRAMMING ABSTRACTIONS

3.1 Motivation

With the increasing size, feature set, and performance of FPGAs, there are many reasons to use them. However, hardware design for FPGAs can be a difficult task. FPGA design comes in two paths. One method is to use a hardware description language. Another method is to use a language or tool that is suited to a particular domain. Unfortunately, there is no dominant design language in networking.

A useful approach for design mapping would allow for different tools to make use of FPGAs while at the same time not requiring the tool to map directly to the FPGA hardware. Instead of a domain specific language, a domain specific design application programming interface (API) can be used, as is presented in this thesis. This API should enable the definition of functional as well as architectural features. The full benefits of FPGAs can only be realized when taking advantage of the flexibility that is provided.

Brebner described a computational model used to implement a mixed version IP router (MIR) based on threads that is used in this thesis. Threads are a prevalent unit of processing in the network-processing domain. As such, this is the model used by network processors. However, unlike in network processors where threads resemble traditional threads running on processors, the threads used in MIR can be implemented in hardware as well as in software. In a separate paper by Brebner, the theoretical framework for threads is discussed further [2]. Several issues were identified as to the nature of threads in hardware. One issue is activation and deactivation of threads. Different processing elements in the system have the capability to start and stop threads. Another issue was the communication between threads. As the FPGA provides synchronization through a global clock and reliable communication channels through wires the

threads were able to be made lightweight. A send-receive handshake protocol was not necessary since the clock edge could be used as the implicit handshake.

In order for the different threads to be able to write simultaneously as well as preserve low latency, a custom memory organization was used in the MIR work. Again, this architectural feature is adopted in the programming model proposed in this thesis. In MIR, as the packet arrives, the received words are broadcast to each of the threads. While the threads process the header of the message, the payload is streamed directly into a buffer. As it is unknown until a later point which port the packet is to be output on, the packet is initially written to all outgoing buffers. When the correct port is determined, the streaming to the other buffers is stopped. To cope with the data that is written to the incorrect output buffers, there is the notion of committing a packet. Only when a packet is committed does the buffer recognize that a packet is there. For those buffers that are not the destination port, the data simply gets overwritten by the next packet. This customization of the memory structure led to many of the performance gains that MIR was able to achieve.

The capability of mapping threads to hardware as introduced in MIR has allowed a commonly used computation framework to be used in FPGA applications. This, together with the architectural flexibility of FPGAs, has formed the basis of a platform for this thesis. While the platform is focused around the execution of threads there are many other building blocks that are necessary for a complete system. In addition to threads and the communication and synchronization between them, memories, external intellectual property, and interfaces to the external system are used to provide complete functionality. These features are programmable through the API.

3.2 Design flow

The programming interface is usable by designers desiring direct access to the platform as well as by tools that provide a higher level of abstraction. The platform is the programmable

soft architecture that includes programmable features such as threads and memories. As shown in Figure 3.1, the platform is an abstraction layer above the FPGAs. The high-level tools use the programming interface to access the platform. This is, in essence, the programming language for programming the platform. Example high-level tools include Teja C from Teja Technologies's [32], Click from MIT [13], and AnyWare from Novilit [31]. The platform then includes a mapping between it and a hardware description. This mapping is the compilation part of the API presented in this thesis. The back-end tools provided by the FPGA vendor are used to map the hardware description to the specific FPGA being targeted.

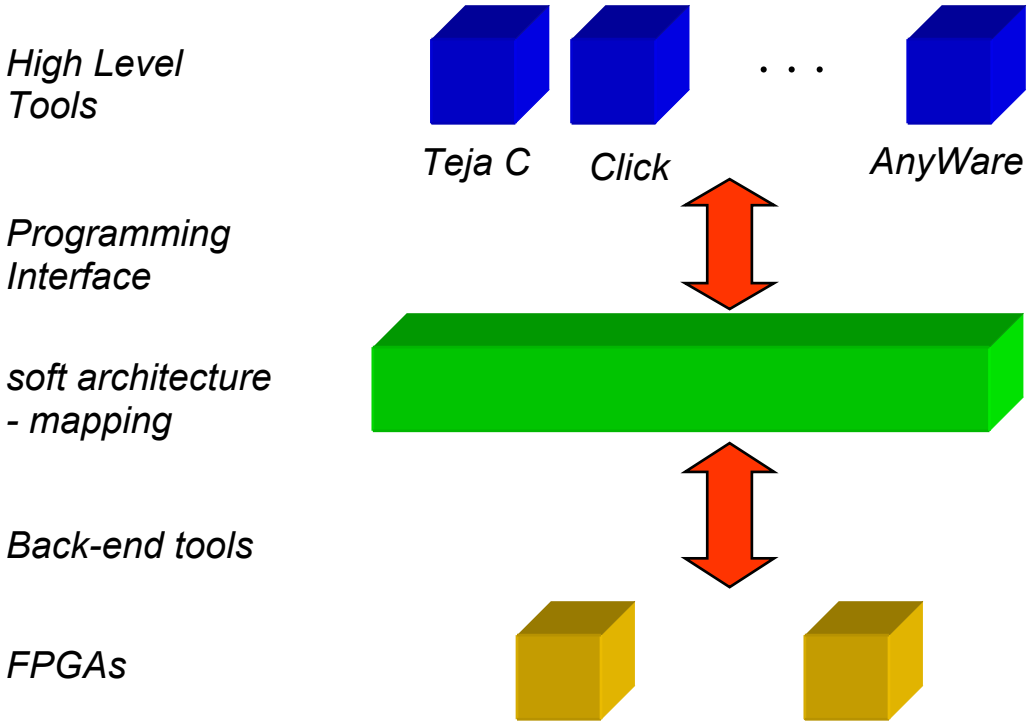

Figure 3.1. Abstraction layers for the programming model.

To support the dual-purpose usage model allowing both high-level tools as well as designers to program the platform, both a Java API and a textual intermediate representation are provided. The API can be useful in two ways. First, it provides access to the system from a

higher level tool's environment. Instead of requiring an intermediate format, the tool can directly call the functions in a library that the programming interface provides. Second, it provides a programmable method to create designs. This can allow easier parameterization, for example. The textual representation is useful for resolving incompatibilities between programming environments. If the API is written in one language and a higher level tool is written in another then using the API may be difficult. For this purpose, an intermediate format using the eXtensible Markup Language (XML) was created. XML has a standardized format and a number of available parsers. The parser then calls the API that will allow building the system. The document type definition (DTD) is a formal grammar to specify the structure and permissible values in the XML document. This is, in essence, the language that provides access to the available features of the system.

The targeted output of the API is a set of files that can then be fed to the Xilinx back-end tools. The back-end tools include synthesis, map, place and route, and bitstream generation. The main output file is a set of VHDL files that contains the design. Needed is both a simulatable version as well as a synthesizable version. The difference only shows up when using certain external intellectual property (IP) cores. For security reasons, certain cores are delivered as netlists instead of as a hardware description language source file. These netlists are inefficient to simulate. To compensate, the cores are also delivered with a behavioral model that can be used for simulation. Another useful output is a constraints file (UCF) that has information about the design that the back-end tool can use. Examples include timing-constraints to ensure the required clock frequencies are met. A final output is a simulation script file that can be used by the ModelSim simulator. This will provide test vectors as well as commands to load the necessary files into the simulator.

Summarized in Figure 3.2 is the design flow. There are three design entry methods, as made accessible by the API and intermediate format.

- The first method is a design entry tool that will output XML making use of the defined DTD to go with the system. The design entry tool can be, for example, one of the high-level languages shown in Figure 3.1. However, it can also be a simple text editor, as was used in the development of the programming interface for this thesis. The XML will then be read in by a parser which makes corresponding calls to the API.
- The second method is similar to the first. The user again enters the design in a high-level language or tool. That tool then makes use of the API directly to create the design.
- The final method is for a user to design using the API directly through a custom Java program.

In addition to the user interface, Figure 3.2 also shows the flow for mapping to the FPGAs. The output of the API is a set of files. One set of the files are VHDL files that can be used as input to the back-end tools. These tools will provide a mapping between the hardware description and the FPGA. The output of the back-end tools is the bitstream that is used to configure the device. The other set of files are the simulateable VHDL files that are then used with the simulator to create waveforms.

The focus of this thesis is on the first flow consisting of an XML file format read into a parser that makes use of the API to generate synthesizable VHDL. The XML is hand coded in all of the examples in this thesis. Chapter 5 discusses the method higher level tools would use to complete the first flow.

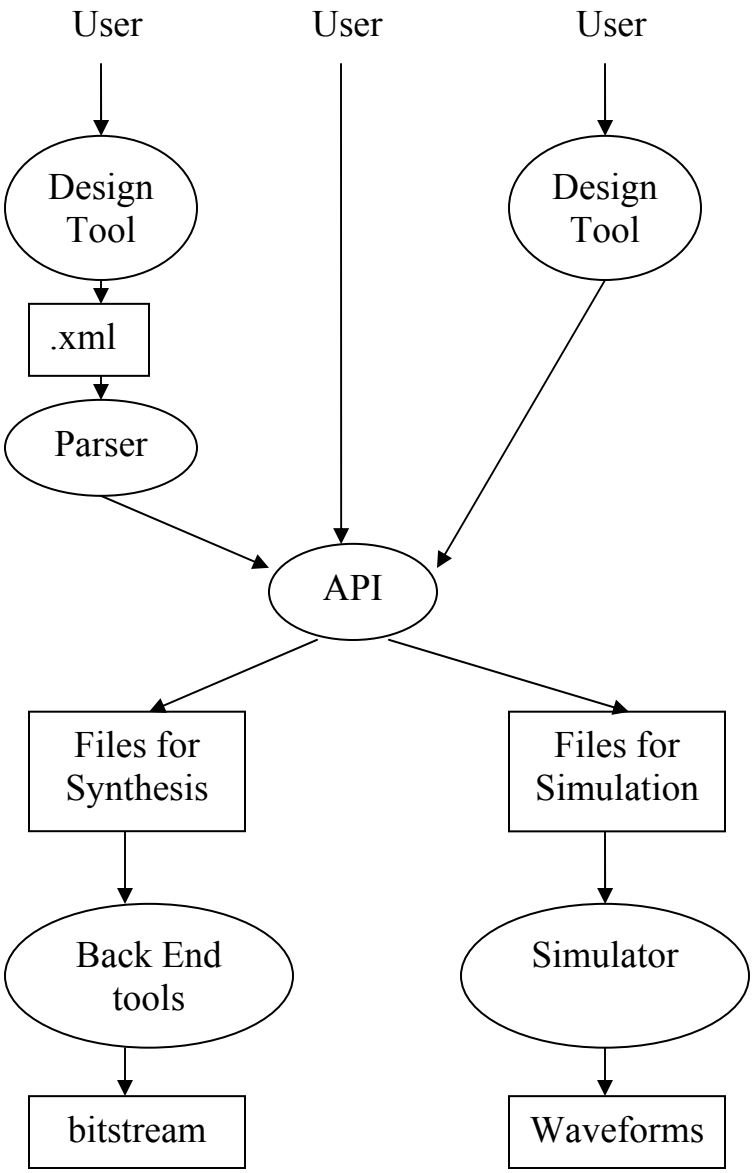

Figure 3.2. Design Flow.

3.3 Description of Programming Interface

The programming interface is an abstraction layer above the hardware. It is specific to the networking domain by using constructs that are particularly suitable to processing discrete packets of data. These constructs are the abstraction primitives that are used by the user to define

the application. As they are an abstraction, the programming interface will provide the mapping to the hardware implementation.

In general, the packets of data stream through the system. While in the system, processing is done through either simple manipulations to the header or complex manipulations to the payload. For this capability there are four main constructs that are needed, as shown with an example system in Figure 3.3. These include the interface to the external system, the memories for buffering of packets, threads for simple manipulations and control, and externally defined intellectual property for complex manipulations. Additionally, there are constructs for communication between threads, as well as the synchronization of the threads. Each of these constructs is discussed in more detail in the following sections.

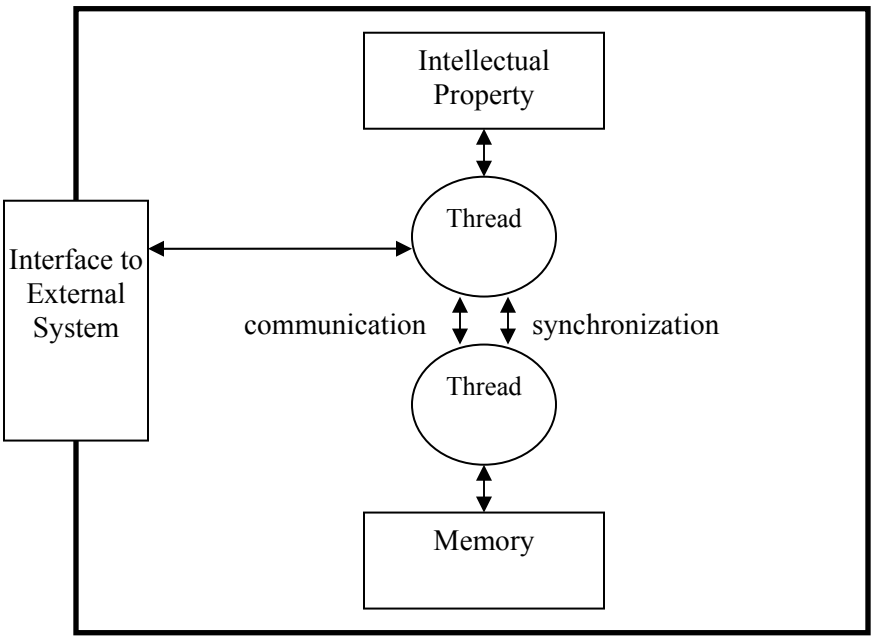

Figure 3.3. Graphical representation of the available constructs.

3.3.1 Threads

From a user's perspective, the programming interface provides a coding environment targeting multiple micro-engines that can operate in parallel. These micro-engines correspond to the threads as used in MIR. An instruction set is used to program the micro-engines. While

presenting a similar programming model, the micro-engine is not intended to operate as a traditional microprocessor – i.e. a fetch, decode, and execute pipeline feeding an ALU. Instead, it is implemented as a custom finite state machine. Each instruction used has a dedicated implementation. There is no additional support required for unused operations in the instruction set. Also, there is instruction level parallelism, since multiple instructions can be executed simultaneously. This is not limited by the architecture of the processing element, as would be the case in a microprocessor.

Shown in Figure 3.4 is the definition of a thread in the XML intermediate textual format that reads data from the receive port of a gigabit Ethernet and places it in a buffer. The XML can be automatically generated by high-level tools. Or, as is the case is this thesis, the XML can be hand coded. The definition starts with the variables as shown on lines 4 through 8 in Figure 3.4. Variables can be internal, input, or output. The input and output variables are used for inter-thread communication and are discussed in Section 3.3.5. However, the use internal to the thread is the same for each. They can be assigned to and read from. Internal variables are unique in that they retain their value. Included in the definition of each variable is the bit-width and default value. For output variables the default is the value that will be output unless the user explicitly outputs a different value. For internal variables it is the value that is to be assigned before the thread executes.

After the variables, the states must be defined as shown on line 9 in Figure 3.4. A tag that starts this definition tells which state is the start state. This is the state that when the thread is started will be the first to execute. An optional tag can be used to tell what the stop state is. By default stopping a thread will immediately put the thread in an idle state waiting for the start signal. However, a stop state can be given that allows some cleanup processing to be done. Another optional tag can be used to tell the thread to use an alternate signal to start the thread. This may, for example, be a data ready signal from an interface or a data available signal from a buffer.

After the tags defining the state machine control, each of the states is defined as shown on lines 10 through 14 and 15 through 43 in Figure 3.4. Within a state can be operations, conditionals, and transitions. Operations make use of the instruction set to define the functionality. They can use variables as the parameters or constants. The constants can be referred to symbolically or using numeric notation. Referring to a constant symbolically requires defining the constant when defining the system. The numeric notation can accept integer, hexadecimal, or binary values. Integers are simply the value. Hexadecimal values are prefixed with a 0x as in 0xF1234. Binary values are prefixed with a 0b as in 0b01101.

Conditionals, as shown on lines 16 through 42 in Figure 3.4, are groupings of operations, transitions, or other conditionals, that depend on a certain condition. This is a standard "if, else if, else" mechanism. Each branch of the conditional requires a condition as shown on line 17 of Figure 3.4. These are part of the instruction set and include instructions such as EQUALS and LESS_THAN. Also for each branch of the conditional, there can be operations, other conditionals and transitions.

Transitions tell what the next state to execute is. There can only be one transition per possible execution path in a given state. If not, this would lead to ambiguous behavior.

```
1: <FSM name="eth_rx_thread">
 2: <useinterface intname="RX" name="mygmac" port="rx"/>
 3: <usemem intname="PUT" name="ethrecv_buf" port="put"/>
 4: <variables>
 5:   <internal name="len" width="16"/>
 6:   <internal name="addr" width="11"/>
 7:   <internal name="goodFrame" width="1"/>
 8: </variables>
 9: <states start="startState" altstart="RX_dataValid">
10:   <state name="startState">
11:     <operation op="WRITE_DATA" params="PUT, RX_Data, 0, 4"/>
12:     <operation op="ASSIGN" params="addr, 4"/>
13:     <transition next="writeData"/>
14:   </state>
15:   <state name="writeData">
16:     <conditional>
17:       <condition cond="EQUAL" params="RX_badFrame, 1">
18:         <transition next="startState"/>
19:       </condition>
20:       <condition cond="EQUAL" params="RX_dataValid, 1">
21:         <transition next="writeData"/>
22:         <operation op="WRITE_DATA" params="PUT, RX_Data, 0, addr"/>
23:         <operation op="ADD" params="addr, addr, 1"/>
24:         <conditional>
25:           <condition cond="EQUAL" params="addr[1:0], 0b00">
26:             <operation op="SUB" params="len, addr, 4"/>
27:           </condition>
28:           <condition cond="else" params="addr[1:0], 0b01">
29:             <operation op="CONCAT" params="len, 0b00000, addr[10:2], 0b00"/>
30:           </condition>
31:         </conditional>
32:       </condition>
33:       <condition cond="EQUAL" params="RX_goodFrame, 1">
34:         <operation op="ASSIGN" params="goodFrame, 1"/>
35:         <operation op="WRITE_DATA" params="PUT, len[7:0], 0"/>
36:         <transition next="writeLen"/>
37:       </condition>
38:       <condition cond="else" params="">
39:         <operation op="WRITE_DATA" params="PUT, len[7:0], 0"/>
40:         <transition next="writeLen"/>
41:       </condition>
42:     </conditional>
43:   </state>
…
```

Figure 3.4. Example thread definition for interfacing to the gigabit Ethernet's receive (RX) port.

3.3.2 Included Intellectual Property

Many algorithms are not naturally described in terms of a finite state machine. For example, performing encryption using a description of the data flow is much more efficient, both

in terms of performance and specification, than expressing as a finite state machine. To accommodate these cases within the network processing system is the capability to include and make use of intellectual property cores. These cores would be defined using a separate manner and exist in the form of a netlist. In the programming interface, the interface of the block has to be defined as shown on lines 1 through 7 in Figure 3.5. This includes inputs and outputs. It is assumed that there will be a clock input as well as a reset signal, which do not need to be defined. In the programming interface the user then needs to make an instance of one of these cores as shown on line 8 in Figure 3.5. This requires a name as it will be referred to in the system and a type as defined in the definition of the interface.

```
1: <COPDEF type="Multiplier">
2:   <input  name="in1" width="32"/>
3:   <input  name="in2" width="32"/>
4:   <input  name="invalid" width="1"/>
5:   <output name="res" width="32"/>
6:   <output name="outvalid" width="1"/>
7: </COPDEF>
8: <COP type="Multiplier" name="mymult"/>
```

Figure 3.5. Example definition of an instantiation of included Intellectual Property.

3.3.3 Interfaces

At the perimeter of the defined system are one or more user defined interfaces. The interfaces in this system are well defined and allow for getting packets into and out of the system. They are not necessarily restricted to connecting to input or output pins of the FPGA. Instead they can define an exchange point between the network processing system defined using the programming interface and a larger design implemented on the FPGA that makes use of the defined network processing system.

The interfaces are not simply a grouping of signals. They also include functionality that enables the exchange of packets between the network processing system and the external system. One example of an interface is gigabit Ethernet [25]. This interface contains functionality to read

a stream of data off of the multi-gigabit transceivers and perform framing and error detection. It exists as a predefined core. Like gigabit Ethernet, all interfaces that are available for the system to use will exist as predefined netlists. To make an interface available for use involves a process of adding it to the list of available interfaces. A Java class is created to detail information about the interface. This will provide the programming interface, also written in Java, with information relating to the implementation. The first requirement is a list of signals that interface with the external system. This includes clocks as well as data and control signals. The second requirement is a list of ports that are used internal to the network processing system. A port is a grouping of signals that relate to each other. For example the transmit (TX) port of the gigabit Ethernet interface contains a data bus as well as control signals to signify, for example, the beginning of a frame.

Once this class is created, systems can then make use of it using the programming interface. In order to make use of the interface block, a simple "include" mechanism is used. Shown on line 1 in Figure 3.6 is the syntax to include an interface named "mygmac" that is of type "GMAChook."

After the inclusion of an interface in the system, it must then be associated with a thread. This is done through the "useinterface" tag within the definition of a thread as shown on line 3 in Figure 3.6. Each of the threads with an associated interface provides a way to handle messages entering and exiting the system. As each type of interface will have different protocols to read/write data, there is the need for varying functionality. One interface may be completely streaming, where once data starts it will receive a new value every cycle. Another interface may have flow control where there may be pauses in the data stream. The programming interface provides the ability to specify the functionality of the threads that attach to the interface block as well as telling which interface block to use.

Their implementation is similar to the micro-engines discussed in section 3.3.1. That is, a finite state machine based micro-engine executes a series of instructions to perform the

functionality. The main difference between the different micro-engines is that the interfacing threads have access to certain 'system' instructions that the other threads do not have. These are instructions to communicate with input/output interface logic blocks that communicate with another system (on the same chip or on a different one).

```
1: <interface name="mygmac" type="GMAChook"/>
…
2: <FSM name="eth_rx_thread">
3: <useinterface intname="RX" name="mygmac" port="rx"/>
…
```

Figure 3.6. Example syntax used to include and interface and associate with a thread.

3.3.4 Memories

Memory is a key component of many systems. In particular for a network processing system, memory enables buffering of packets, tables for lookup, and storage for state. Like the interface blocks, memories need to be both instantiated and associated with a thread. To instantiate a memory involves specifying the name to be used by the system as well as the type. Certain memories will be parameterizable and thus require additional specification. Shown on lines 1 through 5 in Figure 3.7 is an example. Each memory can have one or more ports, depending on the type. For example, a FIFO would have one write port and one read port. Each port has several signals that are used for access to the memory. To accommodate different processing rates, each port will have its own input clock. As with interface ports, threads can also be associated with a given memory port as shown on line 7 of Figure 3.7. Access to the ports involves read and write instructions, conditional instructions, and memory specific instructions. Finally, like interface blocks, memory elements exist as a predefined netlist.

```
1: <mem  name = "ethrecv_buf"  type="PutGetMem">
2:   <param name="size" value="18432" />
3:   <param name="putWidth" value="8" />
4:   <param name="getWidth" value="32" />
5: </mem>

6: <FSM name="eth_rx_thread">
7: <usemem intname="PUT" name="ethrecv_buf" port="put"/>
…
```

Figure 3.7. Example syntax to define a memory as well as attaching it to thread.

While memory elements can come in various types, sizes, and interconnections a general mechanism for supporting a range of memories is left as future work. Instead, a choice of four memory element types is given. Each is discussed further in the following sections.

3.3.4.1 FIFO Memory Element

The FIFO memory element is a common data structure used both in hardware and software. In this case it has one write port and one read port, both of which have parameterizable data width's that are not restricted to be of the same width. The writes occur in order and are read in the same order in a first in first out manner. Writes are done through the WRITE_DATA instruction and reads are done through the READ_DATA instruction. An address is not needed in either case.

3.3.4.2 PutGet Memory Element

The PutGet memory element is similar to the memory designed in MIR and was adapted by James-Roxby [28]. The PutGet memory element is an extension of the FIFO element. Like the FIFO, it has one write port (Put) and one read port (Get). However, whereas the FIFO memory element is a queue of single units of data (e.g. byte or word) the PutGet memory element is a queue of objects (e.g. packet). Each object contains multiple units of data. The queue

functionality is enabled by the use of a commit mechanism that allows the writer to specify that it has completed writing to the object and the queue can make it available to the read port. Writes within the object do not need to be in order as the read port will not be able to access it until after the writer has committed the object to memory. Because of this, the WRITE_DATA and READ_DATA instructions used to access the data include an address. This commit mechanism, accomplished through the COMMIT instruction, also enables the writer to speculatively write into memory.

3.3.4.3 SharedMemory Memory Element

The SharedMemory memory element allows the sharing of state information. It provides an interface with multiple ports that have read and write capability to access a single memory. Currently only two ports are supported as the embedded block RAM provides hardware dual-port support. Extra arbitration would need to be added to support more ports.

Also supported in the SharedMemory memory element is synchronization between threads through the ability to perform locks on variables. The LOCK instruction allows a thread to request a lock on a given memory location. The result of the request will be available a single cycle later. The GOT_LOCK condition allows the thread to test if it was granted the lock. The UNLOCK instruction allows a thread to release the lock for a given memory location.

The implementation of the locking capability, as seen in Figure 3.8, is through the use of an extra embedded block RAM to hold the lock status of every memory location. The embedded block RAM of the Xilinx Virtex II FPGAs have the capability for each write to memory to have as output either the value that was just written or the value that was in memory that got overwritten. For the locking capability, the value that was overwritten is of interest and that option was used. For this implementation a ‘1’ in memory signifies that the memory location is currently locked. A ‘0’ signifies that it is not locked. When a lock is requested, a ‘1’ will be written to the given memory location. The output of the memory, a single cycle later, will hold

the previous value. If the output is a '1', the lock request failed since it was already locked. If the output is '0', the request succeeded and the memory location now holds a '1' so any following requests will be denied. Shown in Figure 3.8 is a block diagram of the implementation. As with the data part of the SharedMemory element, the dual-ported capabilities of the embedded block RAM is used to provide the access to each of the ports. However, extra logic is required for the case when there are multiple requests to the same memory location. Based on the priority or scheduling, only one port's request will be attempted. The select line of the output multiplexer will be set to select the output of the memory. The other's request will be denied by setting the select line to select the constant '1'.

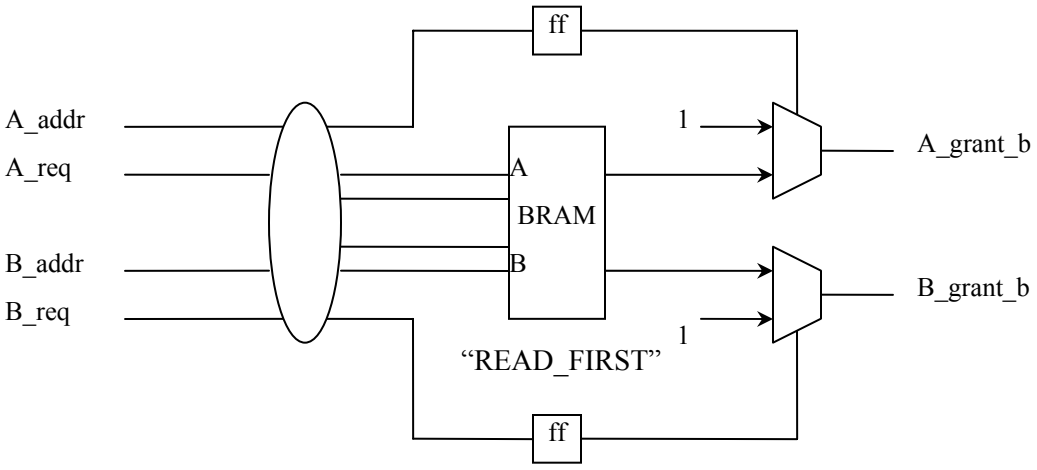

Figure 3.8. Diagram of implementation of locking in the Shared Memory element.

3.3.4.4 DPMem Memory Element

The DPMem memory element is a structure that provides multiple ports with access to multiple embedded RAM blocks. Each accessor has read and write capabilities through the READ_DATA and WRITE_DATA instructions. In addition to providing an address, the accessor also provides the ID of the memory element being accessed. This ID is the data pointer, or DP. Making use of the dual-ported nature of the embedded block RAM in Virtex FPGAs, two

ports can access the same memory at the same time. Each of the memories can be accessed simultaneously by different accessors, keeping single cycle access. Shown in Figure 3.9 is the structure of the DPMem. Only two memories being accessed by four ports are shown. Two of the ports, 0 and 1, map to the A port of the block RAM and the other two, 2 and 3, map to the B port of the block RAM.

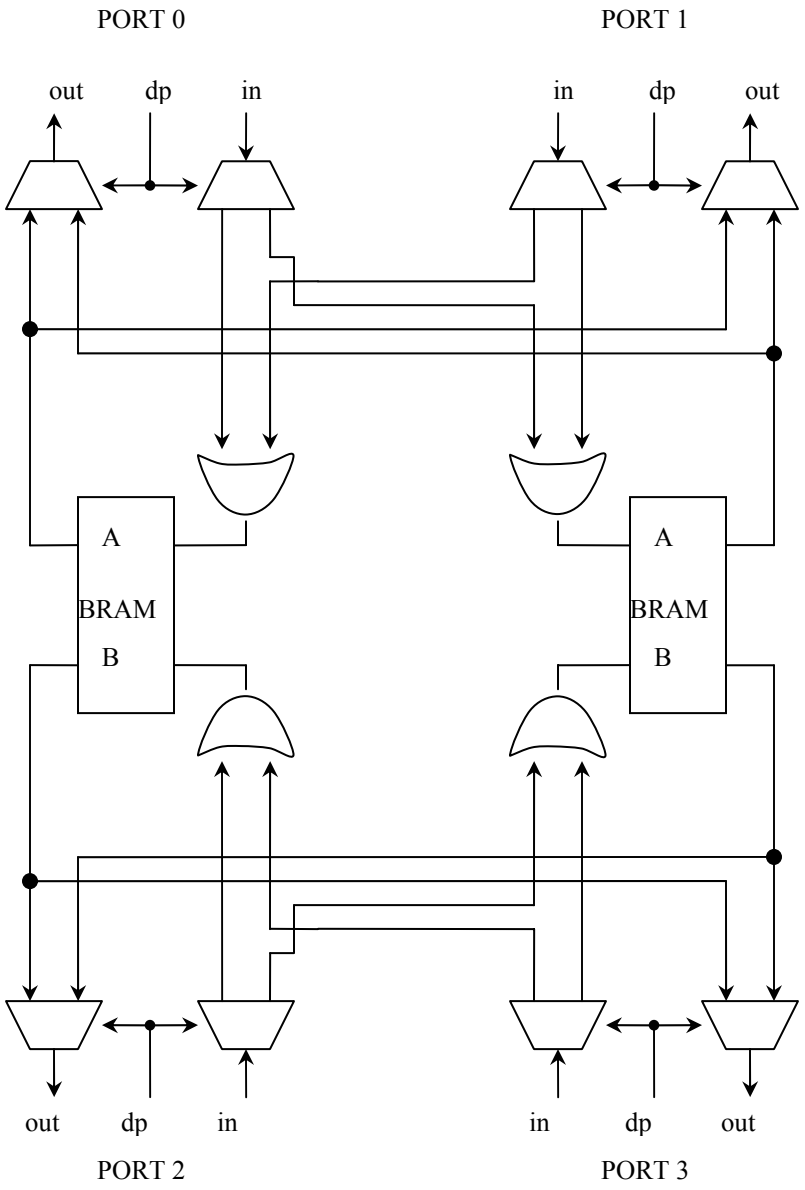

Figure 3.9. Structure of DPMem memory element's data access.

The shared port access is accomplished through multiplexing circuitry. In addition to providing an address, data, and write enable, each port provides the DP. This is used for the multiplexing. The input bits are demultiplexed with each output going to one of the memories. The value of the DP determines which output passes the value of the input to the output. The rest of the outputs are '0'. Since at most one port sharing a BRAM port will have a DP that points to a given memory, the values for that memory are logically or'd together and passed as the input to the block RAM. The outputs from the block RAM connect to a multiplexer for each of the ports. The port's DP then selects which memory value is passed to the output of the multiplexer.

The allocation and deallocation of data pointers is also a part of the DPMem memory element. An example use of the DP in a networking application would be that a receive thread obtains a DP, meaning obtains free buffer space, then receives the packet. It will then pass the DP to another thread to work on the packet. This in turn then passes the DP to a transmit thread which will transmit the packet and free the DP, meaning free buffer space.

Shown in Figure 3.10 is the implementation of the allocation scheme. A FIFO holds a list of available data pointers. Each of the ports has an allocate port which consists of an output DP value as well as an input control signal. At a high level the logic will inspect each of the registers and if any of them has a zero value then it will fill the register with a value from the free DP list. When a port decides to take the DP assigned to it, it will assert the control line which will clear the DP register. This informs the logic structure to allocate another DP. This means that a DP will be allocated to the port before it requests it and therefore there is no delay from when the first byte of a packet arrives. If there are no free buffers, then the register will be 0 and that will inform the accessor that there is no available buffer and that it must act accordingly (e.g. drop the packet).

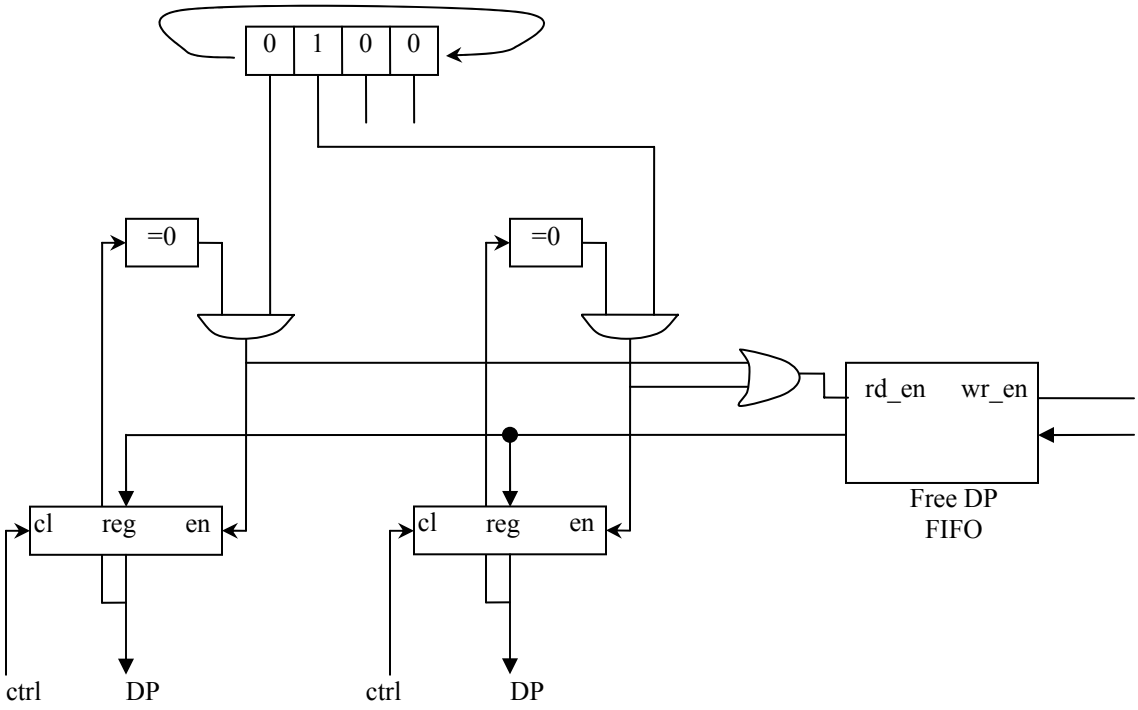

Figure 3.10. Structure of DPMem memory element's buffer allocation.

The deallocation of buffer space works in a similar manner to the allocation and is shown in Figure 3.11. Instead of reading from the free DP FIFO, the deallocation will write to it. The deallocation port includes an input DP and input control signal. The control signal is the enable signal for the register. The deallocation scheme also cycles through each of the registers. Instead of checking for registers that are zero, deallocation checks for registers that are not zero. When it finds one, it will then be written into the free DP FIFO.

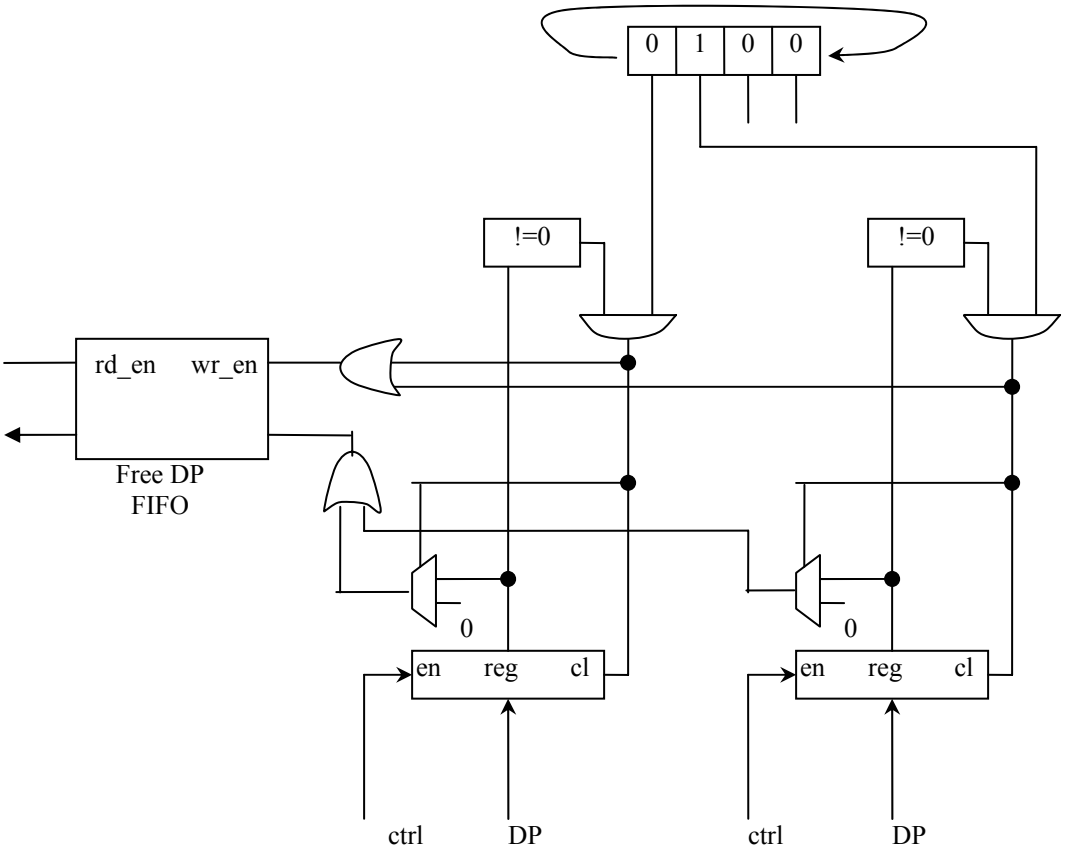

Figure 3.11. Structure of DPMem memory element's buffer deallocation.

3.3.5 Inter-Thread Communication

While memory provides one method for threads to communicate, it has a more general use. As such, it is not the most efficient method for communicating. The programming interface supports two additional forms of communication between threads. The first is a lightweight mechanism where there is simply a direct connection between two threads. This allows for the threads to handle any handshaking that is required. For example, one form of handshaking may involve a data valid signal that tells the receiver of the data that the data can be read. Another form may additionally involve an acknowledge signal which tells the sender that the receiver did receive the data. A final form of handshaking may involve no handshaking. It may be the case that due to the design, two threads may be able to pass data without requiring a handshake.

Using explicit connections provides for user specified communication that has no restrictions. It is also desirable to support a method of communication where the functionality lies in the communication rather than in the threads. For this, channels exist. The difference typically comes with the use. Direct connections are more useful for passing fields of a packet or results from a calculation between threads. Channels are more useful for streaming a packet, or any other ordered data, through the system to each of the threads. The streaming may be continuous and have data available every cycle or the stream may be such that data is not available each cycle.

3.3.5.1 Explicit Connections

The simplest form of communication is through direct connections between two threads. In the variables definition section of the threads a given signal can be defined as input or output. Then, through the programming interface, different variables can be connected. For each connection, a name, a source, and one or more sinks are given. The source and sinks are tuples that contain the name of the thread and the variable. An example is shown in Figure 3.12. As the threads will be synchronized, data transfer is very application specific. The sending thread will assign a value to the output variable. This will also mean that that value will appear on the receiving thread's input variable and can be used. The value on the receiver's input only is held as long as the sender's output retains that value.

```
<connection name="RPClen" width="16">
  <src element="RPC_THREAD" port="outLen"/>
  <sink element="IP_THREAD" port="inFromRPCLen"/>
  <sink element="UDP_THREAD" port="inFromRPCLen"/>
</connection>
```

Figure 3.12. Example of explicit inter-thread communication.

3.3.5.2 Channels

The other form of communication is through channels. This provides the capability for more complex data transfers. Just like memories, a list of channels exists to choose between. Each one exists as a predefined netlist. To make use of a channel, the system first needs to include it. This is using the same "include" mechanism that exists for the interface and memories and is shown on line 1 in Figure 3.13. The threads that make use of the channel, either on the sending or receiving side, use the "usechannel" tag within the thread definition as shown on line 3 in Figure 3.13. The thread must specify which port of the channel to connect to.

```
1: <channel name="bcastchan" type="AlignedChannel"/>
…
2: <FSM name="broadcaster">
3: <usechan intname="SEND" name="bcastchan" port="send"/>
…
```

Figure 3.13. Example syntax to define a channel as well as connect the thread to the channel.

One example channel is the AlignedChannel. This channel allows a sender to broadcast a non-continuous stream of data. It also allows the receivers to access the data in a pseudo-random access manner. Each new piece of data has an address associated with it. The address is calculated by the channel. Therefore, the sender simply sends data. The address enables the receivers to simply match address that the thread is accessing with the address of the data in the channel. Since the functionality is in the channel, the receivers do not have to count cycles or deal with data values that are not valid. The receivers simply specify which address to read and the programming interface will insert the proper circuitry to handle skipping over data.

An additional feature of the AlignedChannel channel, is that receivers can access the data on non-word (or non-data unit) boundaries. In addition to sending the current data unit, the channel also sends the previous data unit. This allows the receiver to pick out any non-aligned data. To see where this is useful consider and an IP packet encapsulated inside of an Ethernet

packet. The Ethernet packet size is 14 bytes. Therefore, assuming the width of the channel is 32 bits, a thread that needs to read the IP header and does not care about the Ethernet header will require reading starting at byte 15 which is not aligned to a data unit boundary.

3.3.6 Thread Synchronization

The usefulness behind the threaded model is not just in the computational model but also in the control. Just as with the threads in operating systems running on microprocessors, threads in this model can start, stop, and query other threads. Each of the control commands has an associated instruction: START(thread name) and STOP(thread name). The start command has an alternate use when a channel is used. In addition to the thread name, a starting offset is given. All addresses in a channel are offset from that value. Similarly, there is a query command to tell the status of the thread: IS_FINISHED(thread name). When a thread is stopped, it has the option of either going into the idle state or performing some cleanup before going into the idle state.

CHAPTER 4

COMPILATION

4.1 Hardware Generation

The design flow from Figure 3.2 is repeated again in Figure 4.1. Chapter 3 discussed the input to the programming tools, focusing on the XML intermediate format. This chapter discusses an overview of the mapping process from the users input to the FPGAs. Sections 4.2 and 4.3 detail the compilation process for generating a hardware description language representation of the design. This compilation is part of the functionality of the API. Section 4.4 details the process that takes the output of the API and generates a configuration bitstream.

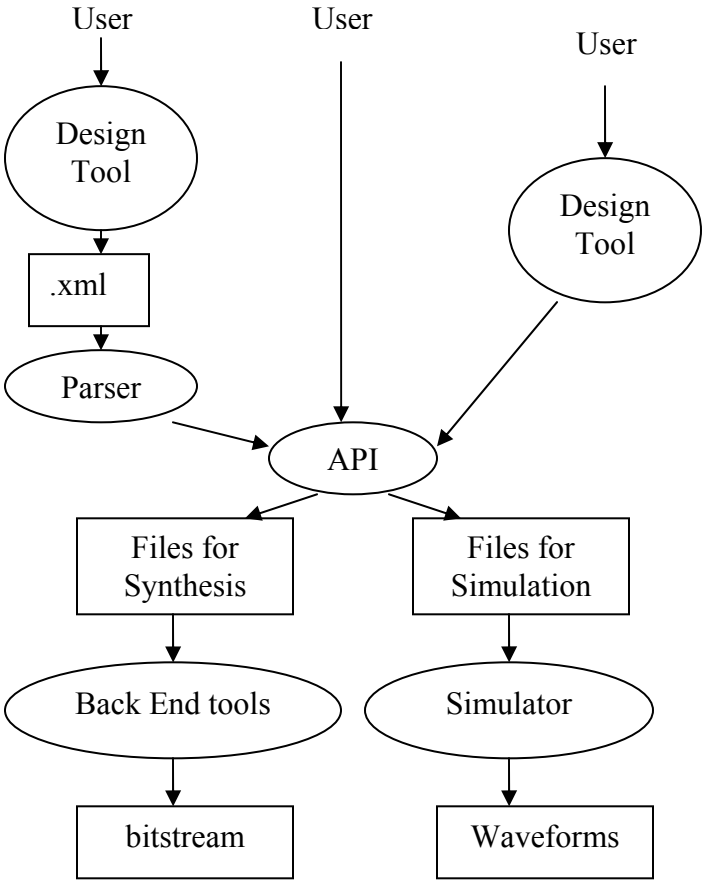

Figure 4.1. Design Flow.

Once the entire application has been defined using the programming interface, the hardware can be generated. This consists of the generation of a collection of VHDL files. There is one top level VHDL entity that instantiates all of the threads, memories, interfaces, and channels. For each of the threads, there is an additional VHDL file that contains the functionality of that thread. The memories, interfaces, and channels all exist as predefined IP, and are expected to be present.

The generation process is driven by the nature of VHDL. VHDL has both structural and behavioral components to it. Each element in the system is a module. When defining the elements, the behavior is described. When generating a system that uses the elements, the modules are connected together defining the structure of the system.

4.2 Top Level Design

The top level design is a design file that forms the root of the hierarchy for the design described in XML. The use here is to instantiate each of the components in the system. This includes defining the component and which signals attach to its interface.

The basic structure of the top level VHDL design file is shown in Figure 4.2. First the entity of the system is defined. This is the interface signals that will connect to pins of the FPGA or, if being used as a subsystem, the logic making use of the design created in XML. After the entity description is the architecture description. The first part of the architecture is the signals that are internal to the component. This only includes signals for this level of hierarchy. After the signals, the logic of the architecture is described. For the systems defined using the design flow presented in this thesis this includes the synchronization logic as well as the instantiatation of each component that appears in the design.

```
entity SYSTEM is
  port (
-- interface
  )
end SYSTEM;
architecture struct of SYSTEM is

-- signals

begin

-- synchronization logic

-- instantiate each component
   -- (interfaces, memories, threads, externally defined IP, channels)

end struct;
```

Figure 4.2. Structure of top-level generated VHDL file.

4.2.1 Interface

The interface is a set of signals described in the entity description that provides the interface between the component and the component a level higher in the hierarchy. Being the top level of the design, this is the interface to the external system. Note that the design created using the design flow presented in this thesis can be a complete FPGA design or a subsystem within a larger design on the FPGA. In the case where the design is a complete FPGA design, the interface defines the pins of the FPGA.

The interface is first assumed to have a global reset signal. The name defaults to "reset" but a different name can be specified in the System tag, shown on line 2 of Figure 4.3. The rest of the signals are determined solely based on the interface components that are present in the design. For each instance in the XML description using the "interface" tag as described in Section 3.3.3, a set of interface signals is created. Recall that each interface component has a set of ports that connect to the threads and also has a single port that defines the I/O signals. Each of the signals in the I/O port is added to the interface of the top level design prepended with the

name of the interface component. For example, the XML description in Figure 4.3 contains two interface components of type AuroraHook named aur0 and aur1. The corresponding VHDL entity description is shown in Figure 4.4. Each AuroraHook contains four signals - RXN, RXP, TXN, TXP - and one required clock - sysclk. While each interface specifies the clocks that are required, the generation of the clock signals at the correct frequency depends on the board being used and is currently left for the user to provide the correct clock.

```
1: <!DOCTYPE System SYSTEM "XMPE.dtd">
2: <System name="mysys" reset="reset">
3:   <interface name="aur0" type="AuroraHook"/>
4:   <interface name="aur1" type="AuroraHook"/>
5: …
6: </System>
```

Figure 4.3. XML code for interface.

```
 1: entity mysys is
 2:   port (
 3:     reset       : in  std_logic;
 4:     aur0_sysclk : in  std_logic;
 5:     aur0_TXP    : out std_logic;
 6:     aur0_TXN    : out std_logic;
 7:     aur0_RXP    : in  std_logic;
 8:     aur0_RXN    : in  std_logic;
 9:     aur1_sysclk : in  std_logic;
10:     aur1_TXP    : out std_logic;
11:     aur1_TXN    : out std_logic;
12:     aur1_RXP    : in  std_logic;
13:     aur1_RXN    : in  std_logic  );
14: end mysys;
```

Figure 4.4. VHDL code for interface.

4.2.2 Signals

The signals are defined at the beginning of the architecture section. These signals are wires that connect the different components in the system. There are three different constructs in the XML based system that cause signals to be created – synchronization, connections, and components other than threads.

The synchronization signals are control signals that allow threads to start, stop, and query the status of other threads. Currently the control signals include start, stop, startThread, stopThread, threadFinished, and isFinished. The signal startThread is a compound type that consists of a start signal as well as an offset. The offset is used for the systems which make use of channels and is left out otherwise. The signal start is an array of that compound type. For each of the threads defined, each of these signals is defined. For example, in Figure 4.5 shows 2 threads named TH_A and TH_B. This would cause the VHDL signals shown in Figure 4.6, lines 1-14, to be defined.

The connections are inter-thread signals that connect output variables in one thread to input variables in other threads. They are explicitly defined using the "connection" tag in the XML giving a name, width, a source, and a list of sinks. The source and sinks definitions are pairs consisting of thread name and variable. For each connection defined, a single signal is created. Figure 4.5 shows a single connection between the output variable outToB of thread TH_A and the input variable inFromA of thread TH_B. The connection is named c1 and is of width 16. The corresponding VHDL for this connection is shown in Figure 4.6, line 31.

Finally, the last set of signals includes the signals for each of the ports of the non-thread components in the system. These include the interface elements, memory elements, channels, and externally defined IP blocks. Note that the I/O port of the interface elements are not included in this list as they are used to define the interface of the top level as described in Section 4.2.1. Shown in Figure 4.5 is a system which uses a single interface element named aur0 of type AuroraHook. AuroraHook has two ports, RX and TX. Each of the ports has a data signal, a set

of control signals to indicate the validity of the data, and an output clock that is to be used by threads that connect to that interface. The corresponding VHDL for the AuroraHook defined in the system in Figure 4.5 is shown in Figure 4.6, lines 15-26.

```
 1: <!DOCTYPE System SYSTEM "XMPE.dtd">
 2: <System name="mysys" reset="reset">
 3:   <interface name="aur0" type="AuroraHook"/>
 4:
 5:   <FSM name="TH_A">
 6:   <variables>
 7:     <output name="outToB" width="16"/>
 8:   </variables>
 9:     …
10:   </FSM>
11:
12:   <FSM name="TH_B">
13:   <variables>
14:     <input name="inFromA" width="16"/>
15:   </variables>
16:   …
17:   </FSM>
18:
19:   <connection name="c1" width="16">
20:     <src element="TH_A" port="outToB"/>
21:     <sink element="TH_B" port="inFromA"/>
22:   </connection>
23:   …
24: </System>
```

Figure 4.5. XML code for signals.

```
 1: -- synchronization signals for THREAD_A
 2: signal TH_A_startThread      : START_TYPE;
 3: signal TH_A_stopThread       : std_logic;
 4: signal TH_A_threadIsFinished : std_logic;
 5: signal TH_A_isFinished       : std_logic_vector(1 downto 0);
 6: signal TH_A_start            : START_TYPE_ARRAY;
 7: signal TH_A_stop             : std_logic_vector(1 downto 0);
 8:  -- synchronization signals for THREAD_A
 9: signal TH_B_startThread      : START_TYPE;
10: signal TH_B_stopThread       : std_logic;
11: signal TH_B_threadIsFinished : std_logic;
12: signal TH_B_isFinished       : std_logic_vector(1 downto 0);
13: signal TH_B_start            : START_TYPE_ARRAY;
14: signal TH_B_stop             : std_logic_vector(1 downto 0);
15:  -- Aurora RX port signals for aur0
16:  signal aur0_RX_clk : std_logic;
17:  signal aur0_RX_data : std_logic_vector(15 downto 0);
18:  signal aur0_RX_rem : std_logic;
19:  signal aur0_RX_startOfFrameBar : std_logic;
20:  signal aur0_RX_endOfFrameBar : std_logic;
21:  signal aur0_RX_sourceReadyBar : std_logic;
22:  -- Aurora TX port signals for myaurora
23:  signal aur0_TX_clk : std_logic;
24:  signal aur0_TX_data : std_logic_vector(15 downto 0);
25:  signal aur0_TX_rem : std_logic;
26:  signal aur0_TX_startOfFrameBar : std_logic;
27:  signal aur0_TX_endOfFrameBar : std_logic;
28:  signal aur0_TX_sourceReadyBar : std_logic;
29:  signal aur0_TX_destReadyBar : std_logic;
30:  -- Connection named c1
31:  signal c1 : std_logic_vector(15 downto 0);
```

Figure 4.6. VHDL code for signals.

4.2.3 Synchronization Logic

Each thread includes a set of synchronization signals that allow it to control and query other threads and other threads to control and query it. The first set, those that allow it to control and query other threads, are each an array where the index into the array is the thread ID of the thread it is controlling or querying. For example, if there are three threads in a system, this array would be of size three where index 0 controls the first thread, 1 controls the second, and 2 controls the third. The second set, those that allow others to control and query it, are each a single value. The value in this second set is the logical or of each of the values with the same index. For example, shown in Figure 4.7 is the generated VHDL code of the each of the

stopThread signals for a 3 thread system with threads named TH_A, TH_B, and TH_C. While this may appear to create extra unnecessary circuitry, since not every thread will interact this way, this is not the case. Synthesis tools will remove unused logic and signals. For example if thread TH_A is the only one to start thread TH_B, then each of the other threads will always assign a logical '0' to the signal that will start thread TH_B. The synthesis tool can perform constant propagation which would eliminate the need for the OR gate for TH_B_stopThread. This feature of synthesis is relied upon.

```
 1: TH_A_stopThread <= TH_A_stop(TH_A_TID) or
 2:                    TH_B_stop(TH_A_TID) or
 3:                    TH_C_stop(TH_A_TID);
 4:
 5: TH_B_stopThread <= TH_A_stop(TH_B_TID) or
 6:                    TH_B_stop(TH_B_TID) or
 7:                    TH_C_stop(TH_B_TID);
 8:
 9: TH_C_stopThread <= TH_A_stop(TH_C_TID) or
10:                    TH_B_stop(TH_C_TID) or
11:                    TH_C_stop(TH_C_TID);
```

Figure 4.7. VHDL code for synchronization.

4.2.4 Instantiate Components

After the synchronization logic is defined, the components in the system must be instantiated. The components include interface elements, memory elements, channels, externally defined IP, and threads.

For interface elements, memory elements, channels, and externally defined IP, signals were created for each signal in their ports. Creating an instance simply includes specifying the name, as described in the XML description, and using the signals defined as described in Section 4.2.2. Additionally, the global reset signal is an input to all components. The interface element has the I/O port that uses the signals defined in the entity description rather than the internal signals. Shown in Figure 4.8 is the instantiation of an interface element named "aur0" that was defined in the XML description using the "interface" tag, as in line 3 of Figure 4.5.

```
aur0 : AuroraHook port map (
  reset              => reset,
  -- I/O port
  sysclk             => aur0_sysclk,
  RXN                => aur0_RXN,
  RXP                => aur0_RXP,
  TXN                => aur0_TXN,
  TXP                => aur0_TXP,
  -- RX port
  RX_clk             => aur0_RX_clk,
  RX_data            => aur0_RX_data,
  RX_rem             => aur0_RX_rem,
  RX_startOfFrameBar => aur0_RX_startOfFrameBar,
  RX_endOfFrameBar   => aur0_RX_endOfFrameBar,
  RX_sourceReadyBar  => aur0_RX_sourceReadyBar,
  -- TX port
  TX_clk             => aur0_TX_clk,
  TX_data            => aur0_TX_data,
  TX_rem             => aur0_TX_rem,
  TX_startOfFrameBar => aur0_TX_startOfFrameBar,
  TX_endOfFrameBar   => aur0_TX_endOfFrameBar,
  TX_sourceReadyBar  => aur0_TX_sourceReadyBar,
  TX_destReadyBar    => aur0_TX_destReadyBar     );
```

Figure 4.8. VHDL code for interface component instantiation.

The instantiation of threads involves four parts – reset and clock, synchronization signals, variables, and component ports. The reset is the global reset and is included in all threads. The clock is also always present. It uses one of the signals that was defined as a clock output of an interface element and is described further in Section 4.2.4.1.

The second part includes the synchronization signals. This is included automatically for every thread and uses the signals as described in Section 4.2.3.

The third part of the thread instantiation includes any variables defined as input or output. For each variable defined as input or output, the corresponding connection defined in the XML is found. As discussed in Section 4.2.2, the existence of a connection in the XML causes a signal to be defined. That signal is used as the input or output for that variable.

The third part involves including the signals associated with any ports that the thread is connected to. In XML, this is defined using one of the “useinterface”, “usemem”, or “usechan”

tags. Section 4.2.2 described that each signal for each port is defined as an internal signal. This is used as the input or output. Shown in Figure 4.9 is the XML definition of a thread that uses the RX port of an AuroraHook interface element as well as the PUT port of a PutGet memory element. Shown in Figure 4.10 is the corresponding instantiation of that thread.

```
 1: ...
 2: <interface name= "myaurora" type="AuroraHook"/>
 3:
 4: <mem  name = "a2e_buf"  type="PutGetMem">
 5:   <param name="size" value="18432" />
 6:   <param name="putWidth" value="16" />
 7:   <param name="getWidth" value="8" />
 8: </mem>
 9:
10: <FSM name="aurora_rx_thread">
11:  <useinterface intname="RX" name="myaurora" port="rx"/>
12:  <usemem  intname="PUT" name="a2e_buf" port="put"/>
13: ...
14: </FSM>
15: ...
```

Figure 4.9. XML code for instantiation of thread.

```
I_aurora_rx_thread : aurora_rx_thread port map (
  clk => myaurora_RX_clk,
  reset => reset,
  startThread => aurora_rx_thread_startThread,
  stopThread => aurora_rx_thread_stopThread,
  suspendThread => aurora_rx_thread_suspendThread,
  threadIsBlocked => aurora_rx_thread_threadIsBlocked,
  threadIsFinished => aurora_rx_thread_threadIsFinished,
  isFinished => aurora_rx_thread_isFinished,
  isBlocked => aurora_rx_thread_isBlocked,
  start => aurora_rx_thread_start,
  stop => aurora_rx_thread_stop,
  suspend => aurora_rx_thread_suspend,
  RX_data=> myaurora_RX_data,
  RX_rem=> myaurora_RX_rem,
  RX_startOfFrameBar=> myaurora_RX_startOfFrameBar,
  RX_endOfFrameBar=> myaurora_RX_endOfFrameBar,
  RX_sourceReadyBar=> myaurora_RX_sourceReadyBar,
  put_WE=> a2e_buf_put_WE,
  put_Commit=> a2e_buf_put_Commit,
  put_Data=> a2e_buf_put_Data,
  put_Flags=> a2e_buf_put_Flags,
  put_Offset=> a2e_buf_put_Offset );
```

Figure 4.10. VHDL code for instantiation of thread.

4.2.4.1 Clocks

Each of the threads, memories, channels, and interface blocks have clocks. However, it is not necessarily the case that they all operate from the same clock. In fact, it is most likely that they do not. With the given framework, the interface blocks play a key role in clock domain determination. The area of clock domains is one that the programming interface attempts to abstract. Each interface can require a different clock frequency so there will be multiple clock domains. Without careful design, this can cause problems. The programming interface will automatically determine which clock domain each thread belongs to. For processes in different domains to communicate, there needs to be special consideration – for example, inserting a memory element in between the processes. One method to calculate the clock domains is to start at the interface block and follow the connectivity graph. In the example in Figure 4.11, thread A connects to the interface block I/F X. Because of this, thread A is in the clock domain. Thread A

is connected through a custom interconnect using the “connection” tag to thread B, which adds thread B to the clock domain. This continues by adding threads C and D. Finally, a dual ported memory element is reached and port A is added to the domain. Likewise, a similar process starting at I/F Y will add threads E, F, G, and H, memory port B, and I/F B to clock domain 2.

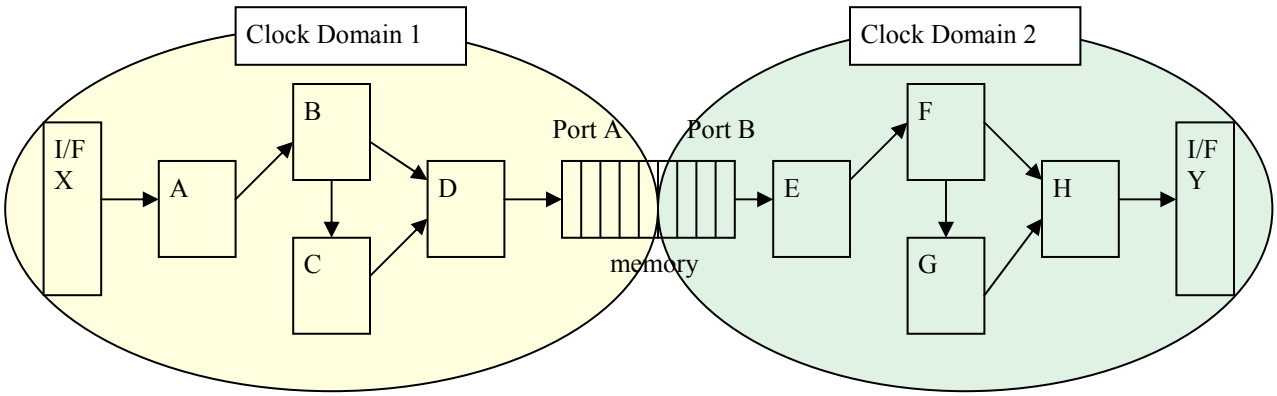

Figure 4.11. Example of clock domain determination.

4.3 Mapping Threads to Hardware

After the generation of the top level design file is complete, the individual generation of each of the threads is performed. Each thread is generated separately as a separate module rather combining the functionality into one, flat, system. The output is dependent on the nature of VHDL. An outline of the VHDL code structure used for each thread is shown in Figure 4.12. First the entity is defined to include the interface. After the entity description is the architecture of the thread. The first part of the architecture is the signals used internal to the module. After the signals, the behavior of the thread is described. The first part of this is the control logic to control the synchronization of the thread. After that is the combinatorial process that is a state machine with all of the combinatorial functionality needed. Following the combinatorial process is the synchronous process which is also a state machine, but includes synchronous behavior

rather than combinatorial. Finally, there is specialized circuitry that is required when using memories or channels.

```
entity THREAD is
  port (
-- interface
  )
end THREAD;
architecture behavioral of THREAD is

-- signals

begin

-- control logic

-- combinatorial process

-- synchronous process

-- special circuitry for memory reads and channel gets

end behavioral;
```

Figure 4.12. Structure of generated VHDL file for threads.

4.3.1 Interface

The first part of any VHDL file is the entity. An entity declaration is used to define the interface of the given hardware module. Included are the signal names, the signal type, the direction (in or out), and if relevant the width. In this case the module is a thread. There are many factors that go into determining the interface of a given thread. Each of the threads will have an input reset signal. As the threads are synchronous, each one will also have a clock input. Additionally, all signals of any attached memory ports are included. The direction is opposite to the direction defined in the memory, as expected. The name given to these signals are the names defined by the memory port prefixed with the internal name given to the connection to the

memory within the definition of the thread. The two parts of the signal name are separated with an underscore ('_').

Another component to the thread interface is the explicitly defined inputs and outputs as defined in the declaration of variables in the programming interface. These variables have a direct correspondence between the XML description and the generated VHDL.

The final component is the synchronization signals. This includes the inputs, startThread and stopThread, for control of this thread. This also includes the outputs, start and stop, for control of other threads. Additionally, the input, isFinished, is included for querying other threads. Finally, the output, threadIsFinished, is included for other threads to query this one. The synchronization signals are automatically included and are not defined by the user.

Shown in Figure 4.13 is an XML description of a design. Included is the description of an interface named aur0 of type AuroraHook, a memory named a2e_buf of type PutGetMem, and a thread named rx_thread. The thread rx_thread uses the RX port of the aur0 interface with an internal name R. It also uses the PUT port of the a2e_buf with an internal name P. It defines one input variable named in0 with a width of 16. Shown in Figure 4.14 is the corresponding VHDL entity description showing the generated interface for the thread rx_thread.

```
 1: ...
 2: <interface name= "aur0" type="AuroraHook"/>
 3:
 2: <mem  name = "a2e_buf"  type="PutGetMem">
 3:   <param name="size" value="18432" />
 4:   <param name="putWidth" value="16" />
 5:   <param name="getWidth" value="8" />
 6: </mem>
 7:
 8: <FSM name=" rx_thread">
 9:  <useinterface  intname="R" name="aur0"    port="RX"/>
10:  <usemem        intname="P" name="a2e_buf" port="PUT"/>
11:  <variables>
12:    <input name="in0" width="16"/>
13:    <internal name="dest" width="16"/>
14:    <internal name="myaddress" width="16"/>
15:  </variables>
16: ...
17: </FSM>
18: ...
```

Figure 4.13. XML code for interface of thread.

```
entity rx_thread is
  port (
    clk               : in  std_logic;
    reset             : in  std_logic;
    -- variables
    in0               : in  std_logic_vector(15 downto 0);
    -- Aurora RX port (int name is R)
    R_data            : in  std_logic_vector(15 downto 0);
    R_rem             : in  std_logic;
    R_startOfFrameBar : in  std_logic;
    R_endOfFrameBar   : in  std_logic;
    R_sourceReadyBar  : in  std_logic;
    -- PutGet PUT port (int name is P)
    P_WE              : out std_logic;
    P_Commit          : out std_logic;
    P_Data            : out std_logic_vector(15 downto 0);
    P_Flags           : out std_logic_vector(1 downto 0);
    P_Offset          : out std_logic_vector(9 downto 0);
    -- synchronization
    startThread       : in  START_TYPE;
    stopThread        : in  std_logic;
    threadIsFinished  : out std_logic;
    isFinished        : in  std_logic_vector(NUM_THREADS-1 downto 0);
    start             : out START_TYPE_ARRAY;
    stop              : out std_logic_vector(NUM_THREADS-1 downto 0)
  );
end rx_thread;
```

Figure 4.14. VHDL code for interface of thread.

4.3.2 Signals

After the declaration of the interface is the architecture declaration. This includes a section to declare any internal signals that are used. Like the interface, this includes both explicitly defined signals as well as implicitly defined signals. An explicit list of signals comes from any internal variable as defined in the variables declaration section of the programming interface. In addition to those, the state and nextState signals are required for controlling the state machines. The type of these signals is defined to be an enumeration of all of the possible states. This list is taken directly from the states defined by the designer. A final group of internal signals are required for the special circuitry as defined in Section 4.3.4.4. For each of the internal variables that are assigned to as part of a memory read or a channel get, two additional signals are defined. These are the name of the variable appended with "_sel_alias" and "_alias".

Shown in Figure 4.15 is thread defined in XML named tx_thread that has two internal variables. It also is connected to the GET port of a PutGet memory element. The thread has two states. In one of those states, an instruction reads from the memory port with the internal name G from the address 0 and places the returned data in the variable dest. Shown in Figure 4.16 are the corresponding generated VHDL signals for this thread.

```
 1: ...
 2: <interface name= "aur0" type="AuroraHook"/>
 3:
 2: <mem  name = "e2a_buf"  type="PutGetMem">
 3:   <param name="size" value="18432" />
 4:   <param name="putWidth" value="8" />
 5:   <param name="getWidth" value="16" />
 6: </mem>
 7:
 8: <FSM name=" tx_thread">
 9:  <useinterface  intname="T" name="aur0"    port="TX"/>
10:  <usemem        intname="G" name="e2a_buf" port="GET"/>
11:  <variables>
12:    <internal name="dest" width="16"/>
13:    <internal name="myaddress" width="16"/>
14:  </variables>
15:  <states start=”startState”>
16:    <state name=”startState”>
17:    ...
18:    </state>
19:    <state name=”s2”>
20:      ...
21:      <operation op="READ_DATA" params="G, dest, 0"/>
22:      ...
23:     </state>
24:   </states>
25: </FSM>
26: ...
```

Figure 4.15. XML code for signals of thread.

```
1: -- state machine signals
2: type stateType is (startState,s2);
3: signal state,nextState : stateType;
4: -- internal variables
5: signal dest      : std_logic_vector(15 downto 0);
6: signal myaddress : std_logic_vector(15 downto 0);
7: -- alias signals for dest (it is dest in READ_DATA)
8: signal dest_sel_alias  : std_logic;
9: signal dest_alias      : std_logic_vector(15 downto 0);
```

Figure 4.16. VHDL code for signals of thread.

4.3.3 Control

As the threads are based on state machine implementations, included in all of the threads is a process that controls the state machines. This control forms the basis of the synchronization of the threads. Within each of the states of the XML description the designer defines transitions. Within the combinatorial process, as described in section 4.3.4, this gets implemented as assignments to a variable named nextState. The control circuitry is then used to assign nextState to the current state variable named state on each rising clock edge. In addition to normal synchronous control of the states, there is a reset signal. The stop signal is used as a reset to change the state. As discussed in Section 3.3.6, if a stop state is defined the reset stop signal will assign that state to the state signal. Otherwise, the start state will be used by default. Finally, the threadIsFinished signal gets assigned to notify other threads of the threads state. The threadIsFinished signal goes low, e.g. it is not finished, when either the thread is not in the defined start state or when the startThread signal is high. Shown in Figure 4.17 is an example of the VHDL code that is generated. Shown is for a thread that has a start state named startState and a stop state named stopState.

```
 1: fin : process(state, startThread)
 2: begin
 3:   if ((state /= startState) or (startThread = '1')) then
 4:     threadIsFinished <= '0';
 5:   else
 6:     threadIsFinished <= '1';
 7:   end if;
 8: end process fin;
 9:
10: update: process (clk, reset, stopThread)
11: begin
12:   if reset = '1' then
13:     state <= startState;
14:   elsif clk'event and clk = '1' then
15:     if stopThread = '1' then
16:       state <= stopState;
17:     else
18:       state <= nextState;
19:     end if;
20:   end if;
21: end process update;
```

Figure 4.17. VHDL code for control.

4.3.4 Combinatorial and Synchronous Processes

Each of the threads is implemented as a state machine with both a combinatorial and synchronous part to it. Combinatorial is where the signals are not registered and synchronous is where the signals are registered. As the two implementations are similar in structure based on the XML description, they are described together here. The main difference is the signals that they affect. All internal variables as defined in the XML are part of the synchronous process. All outputs are part of the combinatorial process. Outputs can include memory signals, explicit outputs, channel connections, interface signals, and external IP connections.

The basic structure of a combinatorial process is shown in Figure 4.18. The process begins by declaring the sensitivity list. This is a list of signals where a change in value can affect another signal that is being assigned to in the combinatorial process. Then the process declares default assignments to all of the signals. Later assignments to those signals in the process will overwrite the default value. Once the defaults are assigned, the states are each defined using a case construct in VHDL. The signal being switched on is the state variable. Each of the states

that are defined in the XML description has a corresponding state in the VHDL description. The start state, as defined in the XML description, is a special case state. To add the ability to start a thread, the state machine must be idle until it has been started. For this, the start state is wrapped with a conditional where if the start condition is met, the state gets executed. Otherwise, the state loops back to itself without executing any operations. This is reflected on lines 6 through 10 in Figure 4.18.

Within each of the states is the functionality. This is discussed later in Sections 4.3.4.1 through 4.3.4.4.

```
 1: COMB: process (sensitivity list)
 2: begin
 3:   -- default assignments
 4:   case state is
 5:     when state_1 =>
 6:       if (startThread = '1') then
 7:         -- state functionality
 8:       else
 9:         nextState <= state_1;
10:       end if;
11:     when state_2 =>
12:       -- state functionality
13:     ...
14:     when state_N =>
15:       -- state functionality
16:   end case;
17: end process;
```

Figure 4.18. VHDL code for structure of combinatorial process.

The structure of the synchronous process, shown in Figure 4.19, is similar to the combinatorial process. The main difference is that it is sensitive only to reset and a rising clock edge. The default values are assigned to signals when the asynchronous reset is high. On rising clock edges, the state machine's case statement gets executed.

```
 1: SYNC: process (clk, reset)
 2: begin
 3:   if (reset = '1') then
 4:     -- default assignments
 5:   elsif (clk'event and clk='1') then
 6:     case state is
 7:       when state 1 =>
 8:         if (startThread = '1') then
 9:           -- state functionality
10:         end if;
11:       when state 2 =>
12:         -- state functionality
13:       ...
14:       when state N =>
15:         -- state functionality
16:     end case;
17:   end if;
18: end process;
```

Figure 4.19. VHDL code for structure of synchronous process.

There are three main components of the XML description that get mapped to the state functionality section, each of which is discussed in the following sections:

- operations,
- conditionals, and
- transitions.

4.3.4.1 State Functionality: Operations

Operations are defined in the XML based description by specifying an instruction along with parameters to that instruction. A complete listing of the currently supported instructions and parameters is available in Appendix A. There are currently four types of instructions: general, synchronization, channel specific, and memory specific. General instructions are basic operations such as add, subtract, assign, and concatenate. Shown in Figure 4.20 is a listing of general instructions along with the generated VHDL. It can be seen that there is a one to one correlation between the XML based description and the generated VHDL. Note that the assigned to variable, x in each case, is used to determine if the operation goes in the combinatorial process or the synchronous process.

```
<!-- XML -->
<operation op="ADD" params="x, x, 1"/>
<operation op="SUB" params="x, y, z"/>
<operation op="ASSIGN" params="x, y"/>
<operation op="BIT_INV" params="x, y"/>
<operation op="concat" params="x, x[7:1], 0"/>
```

```
1: --VHDL:
2: x <= x + 1;
3: x <= y - z;
4: x <= y;
5: x <= not y;
6: x <= x(7 downto 1) & '0';
```

Figure 4.20. XML code (top) and VHDL code (bottom) for general operations.

With synchronization operations, the parameter is the name of the thread that the specified operation, start or stop, is to be performed on. In the generated VHDL, each of the threads is assigned a constant integer identifying the thread. This ID is used in the VHDL operation to perform the synchronization. The start and stop signals are arrays where each entry in the array is indexed by the thread ID. Shown in Figure 4.21 is an example of the input XML and the generated VHDL. The stop operation in XML simply corresponds to setting, in the combinatorial process, the bit in the stop array to '1'. The start array is a bit more complex. For the case where channels are not used, all that is needed is the operation shown on line 6 of the VHDL code in Figure 4.21. When using channels, the start operation can be used to pass a start address to the thread that is being started. This start address is relative to the starting thread's start address. The signal baseAddress defines the base offset and the signal offsetSelect defines the alignment for that thread. For example, if a channel is four bytes wide and gets started with a base address of fourteen, in terms of bytes, then it would have an offsetSelect of two.

```
 1: <!--XML -->
 2:
 3: <operation op="STOP" params="broadcaster"/>
 4: <operation op="START" params="IP_THREAD, 14"/>
```

```
 1: -- VHDL
 2:
 3: -- for the STOP operation
 4: stop(BROADCASTER_TID) <= '1';
 5: -- for the START operation
 6: start(IP_THREAD_TID).START <= '1';
 7: start(IP_THREAD_TID).OFFSET <= "0000000000001110" +
 8:                                baseAddress - offsetSelect;
 9:
10: --where in the definitions of constants, this line appears:
11: constant BROADCASTER_TID  : integer := 2;
12: constant IP_THREAD_TID    : integer := 4;
```

Figure 4.21. XML code (top) and VHDL code (bottom) for synchronization operations.

There are six memory instructions: read, write, commit read, commit write, lock, and unlock. The write instruction assigns the offset, data, write enable, and, if the memory is a DPMem, the data pointer signals in the combinatorial process. The commit instruction, specific to the PutGet memory, assigns the commit signal along with the offset in the combinatorial process. The offset is used by the PutGet memory as the length of the commit. The lock and unlock instructions, specific to the SharedMemory memory, assign the offset, lock request, and write enable signals in the combinatorial process. For lock, the lock request signal is '1', for unlock it is '0'. The read instruction is unusual in that it assigns signals in both the combinatorial and synchronous process. Section 4.3.4.4 discusses special circuitry that is required for memory reads to enable them to execute in a single cycle. From this, a signal that is a select input to a multiplexer must be assigned in the synchronous process. In the combinatorial process, the address is assigned. Shown in Figure 4.22 is an example of the various memory instructions using the XML based syntax along with the generated VHDL.

```
<operation op="READ_DATA" params="GET, mydata, addr"/>
          -- in combinatorial process
          GET_Offset <= addr;
          GET_WE <= '0';
          -- in synchronous process
          mydata_sel_alias <= GET_ID;

<operation op="WRITE_DATA" params="PUT, mydata, addr"/>
          -- in combinatorial process
          PUT_Data <= mydata;
          PUT_Offset <= addr;
          PUT_WE <= '1';

<operation op="COMMIT_WRITE" params="PUT, len"/>
          -- in combinatorial process
          PUT_Offset <= len;
          PUT_WE <= '0';
          PUT_Commit <= '1';

<operation op="COMMIT_READ" params="GET, len"/>
          -- in combinatorial process
          GET_Offset <= len;
          GET_Commit <= '1';

<operation op="LOCK" params="SHM, addr"/>
          -- in combinatorial process
          SHM_Offset <= addr;
          SMH_WE <= '1';
          SMH_lockReq <= '1';

<operation op="UNLOCK" params="SHM, addr"/>
          -- in combinatorial process
          SHM_Offset <= addr;
          SMH_WE <= '1';
          SMH_lockReq <= '1';
```

Figure 4.22. XML code and VHDL code (listed after each line of XML) for memory operations.

Like accessing memory, accessing channel data is also done through special operations. In the case of channels there are three: channel put, channel put first, and channel get. Channel puts are like memory writes without the address. They include a data value, a data valid, and a start of message signal. The CHAN_PUT_FIRST instruction is used for the first data value in a message and for it, the start signal gets set. Otherwise, including when CHAN_PUT is used, the

start signal is deasserted. Shown in Figure 4.23 is example XML code for the channel put instructions and the generated VHDL code.

```
1: <!--XML -->
2: <operation op="CHAN_PUT_FIRST" params="SEND, mydata" />
3: <operation op="CHAN_PUT" params="SEND, mydata" />
```

```
 1: -- VHDL
 2:
 3: -- for CHAN_PUT_FIRST
 4: SEND_Data <= mydata;
 5: SEND_start <= '1';
 6: SEND_valid <= '1';
 7:
 8: -- for CHAN_PUT
 9: SEND_Data <= mydata;
10: SEND_start <= '0';
11: SEND_valid <= '1';
```

Figure 4.23. XML code (top) and VHDL code (bottom) for channel put operations.

The CHAN_GET instruction is used to read a value from a channel at a given offset. The desired effect is for the state machine to effectively block until the address of the channel matches the address of the instruction. When they do match, the value on the channel should be accessible through the variable given. The blocking behavior is achieved through wrapping the state in a conditional statement. When the addresses match, the operations in the state are executed. Otherwise, the state does not change and no operations get executed. A special case exists when the CHAN_GET instruction is used in the start state. As previously discussed, the start state is also wrapped with a condition. In that case the channel condition is inside the start condition. However, since the start signal only needs to be held for a single cycle to start the thread, if the channel condition is not met at that time the state machine jumps to an alternate start state. This is automatically generated and is equivalent to the start state except that it is not wrapped with the start condition. Once the channel condition is met, the operations can execute. Shown in Figure 4.24 is example code of a state wrapped in a condtional as the result of a CHAN_GET instruction.

```
 1: <!-- XML -->
 2:
 3: <operation op="CHAN_GET" params="CHAN, temp, 8"/>
```

```
 1: -- VHDL
 2:
 3: when stateX =>
 4:    if (CHAN_ptr = (baseDP + "0000000000001000") and
 5:              CHAN_valid_internal='1') then
 6:        temp_sel_alias <= CHAN_DATA_INTERNAL_SEL_ID;
 7:        -- rest of state functionality
 8:       else
 9:         nextState <= stateX;
10:       end if;
```

Figure 4.24. XML code (top) and VHDL code (bottom) for channel get operations.

To achieve the desired affect of the variable that was read into being able to be used in those operations requires extra circuitry similar to the case of reads from memory. However, a difference exists on how and when the select line is assigned to. In the case of a memory read it is assigned on the next rising clock edge. In the case of a channel read it is assigned when the condition is met. That way, the operations in the state can reference the variable by name and not by the channel signals. This is discussed in Section 4.3.4.4.

4.3.4.2 State Functionality: Conditionals

Conditionals are used to control which operations within a state get executed based on the conditions. In the XML based language the standard if, else if, else type structure is supported. As VHDL also has this construct, the translation is direct. There are eight conditional instructions currently supported. Examples using these eight conditional instructions and the generated VHDL are shown in Figure 4.25.

```
 1: <!-- XML -->
 2:
 3: <condition cond="EQUAL" params="a, b">
 4: <condition cond="NOT_EQUAL" params="a, b">
 5: <condition cond="BETWEEN_INCLUSIVE" params="a, b, c">
 6: <condition cond="LESS_THAN" params="a, b">
 7: <condition cond="GREATER_THAN" params="a, b">
 8: <condition cond="GOT_LOCK" params="SHM">
 9: <condition cond="DATA_AVAIL" params="GET">
10: <condition cond="IS_FINISHED" params="ETH_THREAD">
```

```
 1: -- VHDL
 2:
 3: if (a = b) then
 4: if (a /= b) then
 5: if ((a >= b) and (a <= c)) then
 6: if (a < b) then
 7: if (a > b) then
 8: if (SHM_lockGranted = '1') then
 9: if (GET_nonEmpty = '1') then
10: if (isFinished(ETH_THREAD_ID) then
```

Figure 4.25. XML code (top) and VHDL code (bottom) for conditionals.

4.3.4.3 State Functionality: Transitions

Transitions control the flow of execution between states. Each state defines which state to execute next. This can be within a conditional if the transition is dependent on a certain condition. To implement this in VHDL is to simply assign to the signal nextState in the combinatorial process. Shown in Figure 4.26 is an example of XML with a transition and generated VHDL.

```
1: <!-- XML -->
2:
3: <state name="s1">
4:   <transition next="s2"/>
5: </state>
6: <state name="s2">
7:   <transition next="s1"/>
8: </state>
```

```
1: -- VHDL
2:
3: when s1 =>
4:   nextState <= s2;
5: when s2 =>
6:   nextState <= s1;
```

Figure 4.26. XML code (top) and VHDL code (bottom) for transitions.

4.3.4.4 State Functionality: Special Circuitry for Memory and Channels

One side effect of reading from memory is that it could affect timing. In the case where a user wants a single-cycle access to memory, this could be specified with an instruction of the form:

READ_DATA port, data, address

where port is the internal name of the port from which the read is to occur, data is the variable where the value from memory should be placed, and address is the location in memory of the data to fetch. The sequence of events would first be to present the address to the memory. On the next clock edge the value would be at the output of the memory. However, it would be desirable then to be able to reference the value using the variable name chosen and not the memory signals. It would also be desirable to be able to manipulate the variable (for example "ADD data, data, 1") in the next cycle. On one cycle the outputs of the memory represent the variable data and on the next cycle the outputs of a register represent data. This can be solved through some multiplexing, as shown in the Figure 4.27. This circuitry is automatically inserted when the given thread is attached to a memory. Figure 4.27 shows a register that selects which data value is output. This

is set depending on the state that the state machine is in. A feedback loop exists to keep the value from memory in the variable data even if the address to the memory changes.

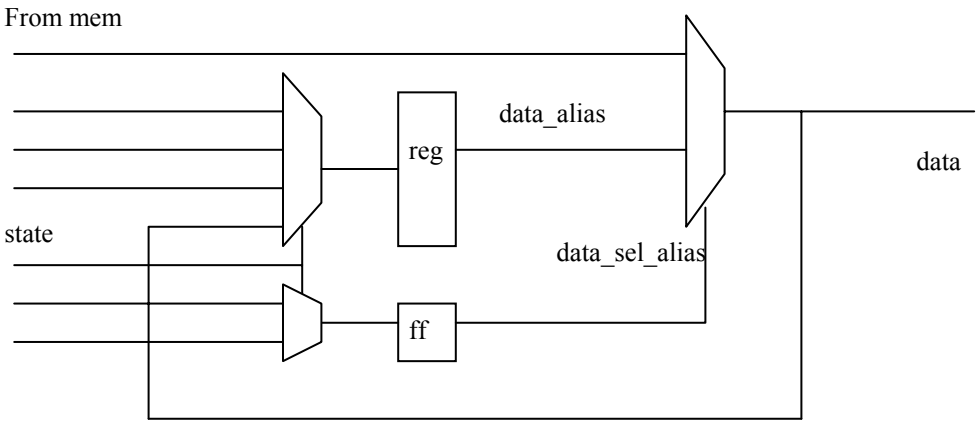

Figure 4.27. Circuitry used to support Alias signals.

Figure 4.28 shows a timing diagram for the circuit shown in Figure 4.27 using an example. In one state there is a read operation, as specified by the user, from the location 0xAA11 into a variable data. This generated hardware for this operation immediately puts the address on the line. At the next rising clock edge the output of the memory has the data value at 0xAA11, which is 13. The generated hardware will also set the select line for the variable data at that rising clock edge so that the data selects the value from memory and not from register data_alias. The next user defined instruction reads a value from 0xFF22 and places it in the variable named other. This instruction does not affect this circuit. The previous value of data is now registered in data_alias. The select line goes low and data gets the value from data_alias as desired. The next user defined state then increments the variable data. This is easily handled as the state selects which value data_alias gets next. In this case it is the output of an adder that adds 1 to data_alias. Note that data_alias is used as the input to the adder even though the user specified data in the instruction. The user only sees the three instructions. The underlying mechanism is automatically generated.

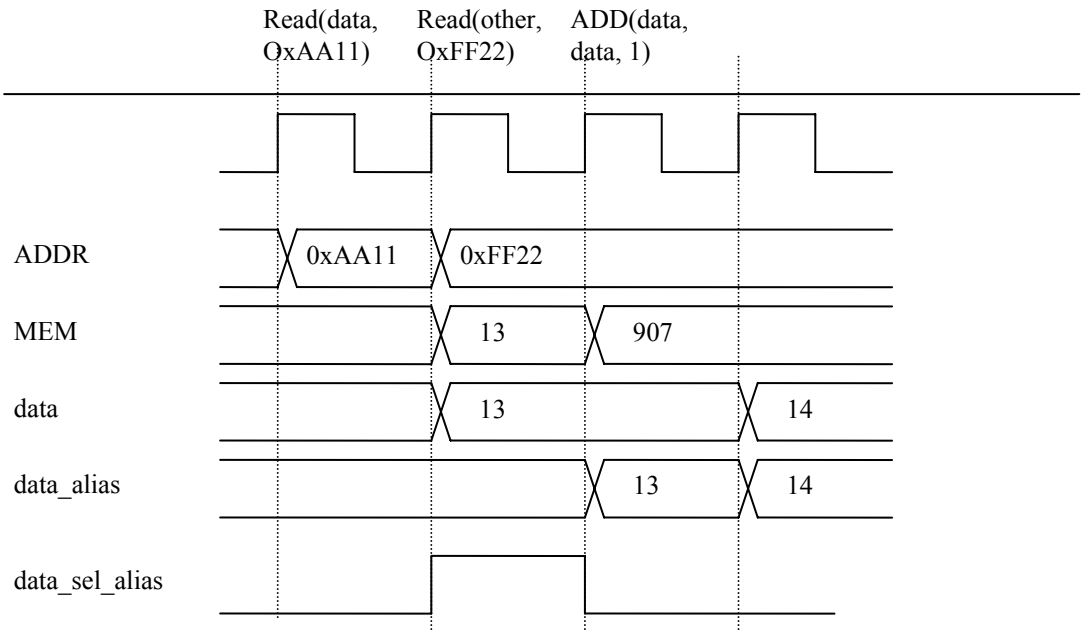

Figure 4.28. Timing Diagram for reading from memory.

This circuitry is necessary. It is not possible to simply replace any references to the variable with the signal name of the memory signal. This is due to the one cycle latency between when the memory read occurs and when the assignment to the variable occurs. To illustrate, consider three states S0, S1, and S2. S0 and S1 both transition to S2. In S0 there is a read from memory and in S1 there is not. The value read from memory will be accessible in state S2. Since state S2 could have been entered from S1 as well, the selection of the source of the variable will be different.

Like memory reads, channel gets have similar multiplexing circuitry. An additional consideration for accessing channels, though, is handling the alignment of the data. When a thread is started, it is given an offset. This offset is used in addressing when comparing against the channel address. For example if the offset is 12 then when a thread reads a value at address 4, as specified by the designer, it is actually reading address 16. For values that are aligned to the data unit size boundary, this addressing is simply the sum of the address and offset. For

unaligned offset values, extra circuitry is needed. This is simply a multiplexer with the lower two bits of the offset selecting which values from the channel pass. As the implemented channel is a 32-bit channel, this allows for byte alignment.

4.4 Bitstream Generation

Once the top-level VHDL module is created by connecting together each of the system components, the design can then be fed into the Xilinx back end tools. Shown in Figure 4.29 is a representation of the process involved in the back end tools. The synthesis tool, XST, is first run on each of the files to convert the design from a behavioral description to a netlist of hardware components. The mapper will then map those components to primitives specific to the architecture. It will also perform optimizations to utilize the hardware efficiently. The place and route tools are then run to lock the primitives to a particular location as well as use wiring to connect them. Finally, a bitstream can be generated using the bitgen program. This bitstream is the configuration that is used to program the FPGA.

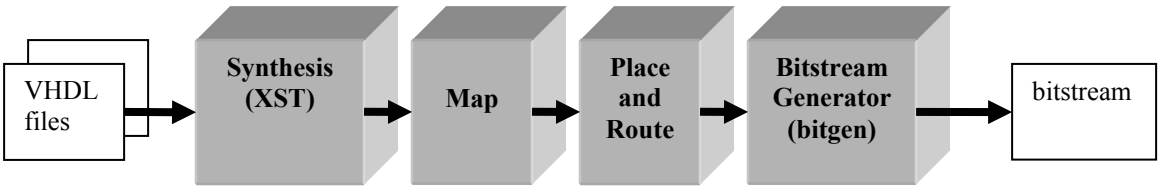

Figure 4.29. FPGA back end tools design flow. .

CHAPTER 5

HIGHER LEVEL TOOLS

5.1 Mapping Higher Level Tools to FPGAs

It has been asserted in this thesis that the programming interface presented is an abstraction layer that bridges the gap between higher level tools that target the network processing domain and implementations on FPGAs. In this chapter two such tools are considered. It should be noted that what is described here has not been implemented. The discussion is only meant as an examination of the method by which one could implement the mapping of these languages to an FPGA using the XML based language. This allows for an assessment of the abstractions provided.

The first language to be considered is Click, developed at MIT for creating modular software routers. The second tool is Teja Technology's application development environment used for designing applications on network processors. Both are currently targeted at a microprocessor-based system. An overview of the tool or language is provided for each followed by a discussion on the use of the abstractions presented in this thesis to map these two tools to FPGAs. While this discussion mainly focuses on the process the tool developer would need to follow in order to incorporate an FPGA flow into the tool, it also includes a discussion of the impact to the end user of the tool.

5.2 Click

Click is a system for creating routers in software from modular components targeting a PC running Linux and more recently FreeBSD. The components, called elements in Click, are C++ classes that form a library for users to choose from and create routers with. However, the use of C++ is only important due to the target platform. It is the Click scripting language that is

used for defining the graph of elements that is independent of the target implementation platform. Conceptually, a packet flows through the graph with each element modifying the packet, inspecting the packet, or deciding the path the packet is to take. Shown in Figure 5.1. is a two port router that is a simplified version of the commonly used example in papers about Click [13]. FromDevice is responsible for receiving the packet from the network interface. When a packet has arrived, the message is passed to the CheckIPHeader element which verifies if the packet is a valid packet. Example checks include checking that the version is IPv4 and checking that the checksum is correct. When that is done, the packet is passed to the Lookup element. The Lookup element has multiple outputs. In this case the IP address is used to determine which path to follow. The process continues until the message is transmitted, dropped, or stored.

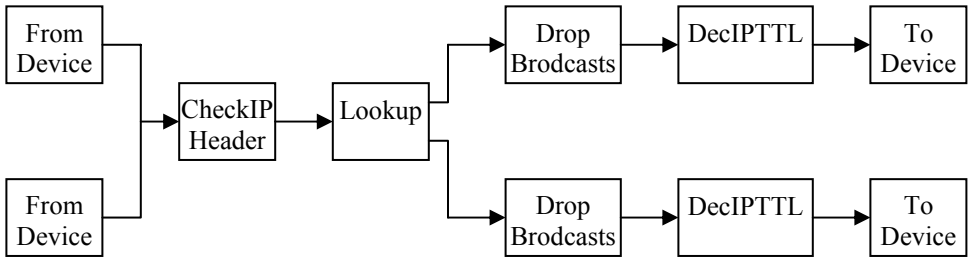

Figure 5.1 Click graph of a 2 port IP router.

Mapping this to an FPGA can be done with hardware description languages as demonstrated by CLIFF. The XML based programming model presented in this thesis could have aided the development of CLIFF and produced a more optimal solution.

Shown in Figure 5.2 is a summary of the steps required to generate a hardware representation of the Click graph using the XML based language presented in this thesis. The flow starts with the click graph (as a .clk file). A series of independent transformations are then made to the graph, each producing a new click graph. This then gets used as input to a tool making use of the XML based language presented in this thesis to generate the hardware system.

First, a base system is created using a one to one mapping between the click elements and the XML based threads. The elements come from the library written in XML. The individual threads are then merged where possible. Finally, the execution of the threads is then optimized to run them in parallel where possible. The output is a complete system using the XML based language.

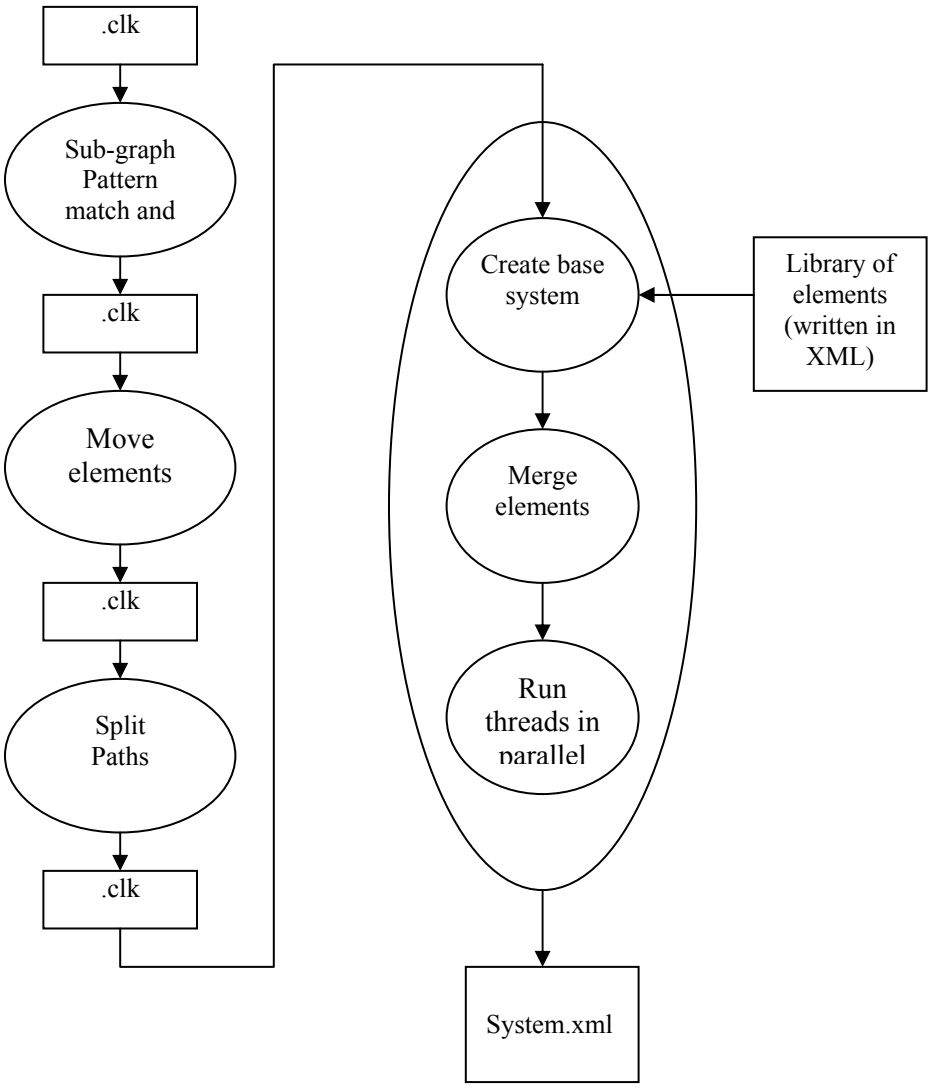

Figure 5.2. Summary of process to map Click to an FPGA using the XML based language presented in this thesis.

As previously mentioned, the use of C++ is not central to the Click system. It is only the means used to implement given the target platform on Linux. For an FPGA solution, the XML based language presented in this thesis is used to describe the Click elements. The interface includes a memory port to connect to the DPMem memory element for accessing the packet. Additionally, if the element requires state information to be kept, then it would instantiate a SharedMemory, though not shared, and add a "usemem" statement to the description. Finally, the functionality of the element would be described using the instructions provided by the instruction set. At this point, the elements are very similar to the Verilog described elements in CLIFF with one noticeable difference. In CLIFF the elements also describe the synchronization between elements consisting of a 3-way handshake.

The base system would be a pipeline making use of the DPMem memory element described in Section 3.3.4.4. The control of the pipeline would be through the built in synchronization mechanisms – starting/stopping the thread next in the pipeline and using the isFinished status flag. The existence of the memory library allows the higher level tool to ignore the issue of creating a suitable memory. This base system is similar to the CLIFF implementation but is only meant as a starting point.

Shown in Figure 5.3 is a diagram of the implementation using the abstractions presented in this thesis. Each of the elements gets mapped to an equivalent thread. The existence of the FromDevice and ToDevice then requires that an interface element be created. The interface GMACHook is shown in this example. Each of the threads has one interface to a shared memory with the stored packets. The Lookup element has an additional interface to a local memory that holds the lookup table. Finally each of the threads has control signals as well as one connection between them. The isFinished and start control signals are shown along with the connection that has a pointer to the packet. The pointer and the start signal are shown as one line going from a thread to the next thread.

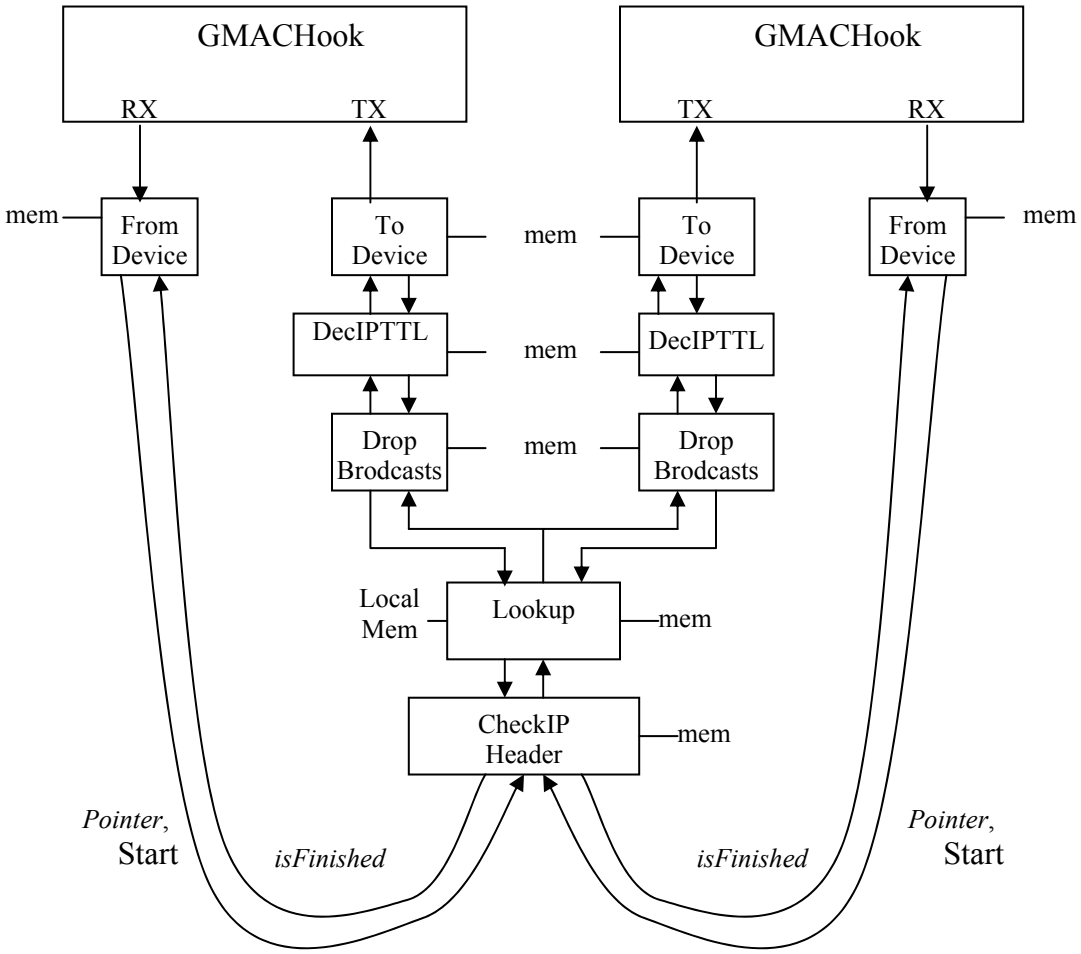

Figure 5.3. XML based implementation of a two port IP router. .

Since the interface and functionality can be flexibly moved by some method calls using the Java API or moving a line in the XML code, optimizations can then be made. A model of performing optimizations in Click through the use of independent tools has been previously established [46]. In this model, a click graph is the input to an optimization tool and a new click graph is the output. This model will be followed as well as allowing for implementation optimizations to be performed by the tool that transforms the Click graph to an FPGA implementation.

Five optimizations are discussed to demonstrate the use of the XML based language in implementing the optimizations. The first three are independent of the XML based language as

they only perform modifications on the arrangement of the Click graph. These modifications to the graph can make it easier to perform the last two optimizations. The last two require modifications to the elements themselves in order to improve performance. The simplified version of the IP router, shown in Figure 5.1, will be used to give an example for each optimization.

5.2.1 Sub-graph Pattern Match and Replace

Click-xform is a tool for Click that will search a Click graph for certain sub-graphs and replaces them with alternate sub-graphs, which may be a single element [46]. Due to the modular nature of Click there is overhead in the generality of the elements. Replacing commonly found element groupings with equivalent alternate sub-graphs can produce a more efficient implementation. Using the alternate sub-graph is not advised since it reduces the generality. This optimization technique is valid in a hardware implementation as well. Overheads exist in the state machine control and synchronization mechanism that can be reduced using this technique. While this could be done automatically for simple cases, as discussed in Section 5.2.4, a better implementation will be generated with this approach. However, it is not a general solution since the replacement rules must be specified and appear exactly. The ability to use a tool created for the software version of Click shows the benefit for using the model of using a set of separate tools for optimizations. While not all optimization tools targeted at the software version will make sense in hardware, there will be some that make sense in both domains.

Shown in Figure 5.4 is a trivial example transformation of the simplified IP router from Figure 5.1. The elements DropBroadcasts and DecIPTTL are both very simple elements and can be combined to reduce overhead. A rule given to Click-xform would specify that any time the sub-graph of DropBroadcasts followed by DecIPTTL appears it should be replaced with OptimalElem. OptimalElem must already exist as an element

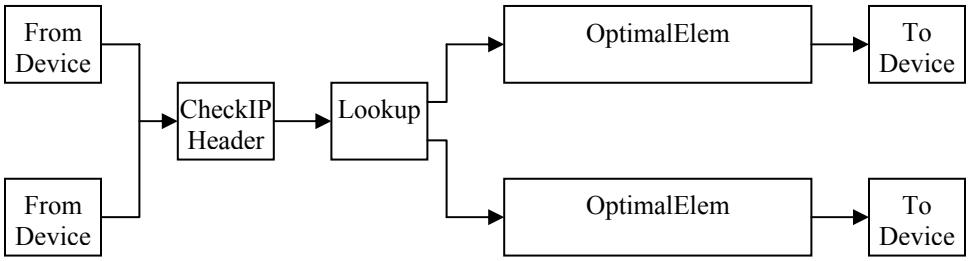

Figure 5.4. Transformed Click graph for the two port IP router using sub-graph pattern match and replace. .

5.2.2 Move Elements

As long as the functionality is not altered, the order of element execution is unimportant. As long as for a given input the same output is obtained, the Click graph can be rearranged. For elements that can output to one of many ports, such as a Lookup element, a dependency exists between that element and everything after it. The elements after the splitting element may or may not get executed depending on the decision of the splitting element. For this reason, it must only execute after the decision has been made. However, if both paths have elements in common there is the possibility for a transformation because no matter what path is taken, the element will get executed. The transformation performed in this step provides benefits only in hardware and only when combined with other optimization steps. Simply moving the element does not provide much benefit other than presenting future steps with a graph more suitable for optimization.

Using the graph in Figure 5.1 as an example again, the DropBroadcast and DecIPTTL elements exist in both paths and therefore will always get executed. Moving them before the Lookup element, as shown in Figure 5.5, will not change the functionality but will enable further optimizations as discussed in following sections. Care must taken to check dependencies between the splitting element and the moved elements. For example if there was an element between Lookup and DropBroadcasts in one of the paths that caused a dependency between that and DecIPTTL then the DecIPTTL could not be moved since it must get executed after that other element.

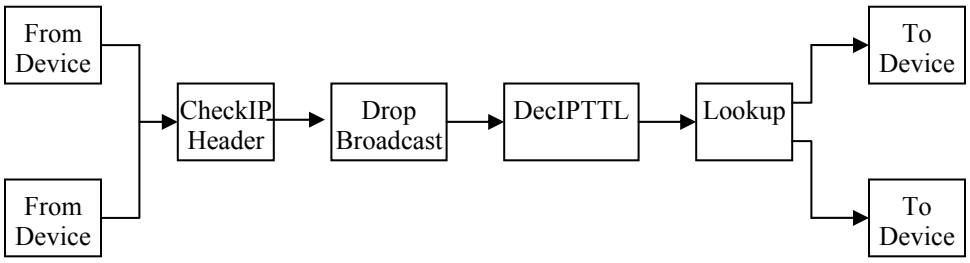

Figure 5.5. Transformed Click graph for the two port IP router using technique to move elements.

.

5.2.3 Split Paths

While the sequential nature of the graph in Figure 5.1 matches the sequential nature of microprocessors, an implementation on an FPGA affords the opportunity to create parallelism. Duplicating a path is trading off area for performance and can be done at the Click graph level. Shown in Figure 5.6 is the duplication of the entire sequential path from Figure 5.1. The tags in the element source code will specify whether duplication is advisable or not. For example an encryption element that takes a large area of the chip will not be a good candidate for duplication, whereas an element that has a single operation such as DecIPTTL, is a good candidate.

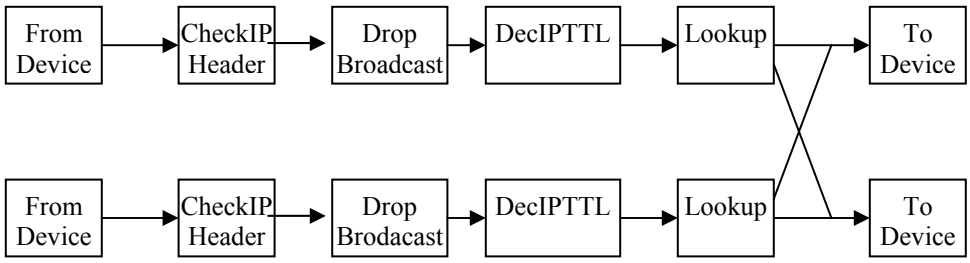

Figure 5.6. Transformed Click graph for the two port IP router using technique to split paths. .

5.2.4 Merge Elements

Under the programming model presented in this thesis, as elements are implemented as state machines, the overhead of state machines needs to be balanced. Elements implemented as state machines in hardware require:

- Circuitry to generate the outputs and internal registers,
- an extra register to hold the current state,
- extra circuitry to multiplex, based on the current state, among the many possible sources that generates either the output or internal register values,
- extra circuitry to determine the next state,
- an interface to the packet data (either an interface to memory, which includes a register holding a pointer to the packet, or an input data register), and
- extra states for handshaking to control the pipeline.

For fine grain elements, the control of the state machine overwhelms the actual functionality. For example, consider the element DecIPTTL that simply decrements the time to live (TTL) field in the IP header. While the operation requires a single adder circuit that can execute in a single cycle, the state machine is more complex and includes the overhead listed above. The overhead of individual elements is multiplied by the number of elements in the system. Additionally, more elements will then require more interfaces to memory, which will require more multiplexing circuitry.

Using state machines that are of coarser granularity may create a more optimal design both in terms of performance and area. Since the elements in Click can be fine grain, instead what is required is the ability to map multiple elements to a single state machine rather than using a one-to-one mapping. The target in this optimization is a series of sequentially executed elements. For example CheckIPHeader, DropBroadcasts, and DecIPTTL in Figure 5.1. Merging these would produce a new graph as seen in Figure 5.7. The new element would be automatically generated and the new graph would be output that makes use of that element. This element is not intended to be reusable, but may be a target for template match and replace as discussed in Section 5.2.1. It should be noted that it is assumed that the tags embedded in the elements

documentation that can be used by tools may include information about whether or not the element may be merged. For example, in this example, the Lookup element would include such a tag.

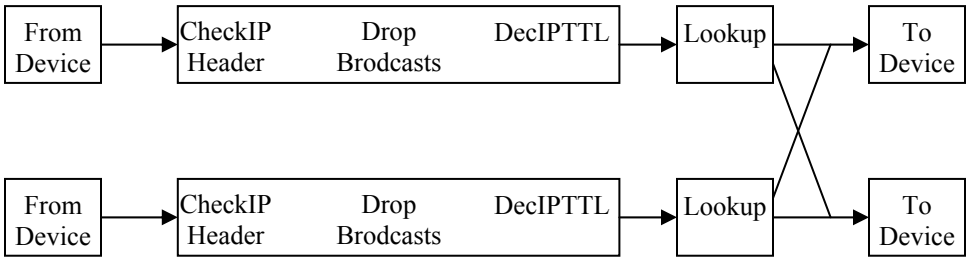

Figure 5.7. Transformed Click graph for the two port IP router after merging elements. .

The most simplistic way to implement the merging of threads is to execute the operations in each element sequentially within a state machine. While simplistic, this may be a reasonable implementation if each of the operations is a simple operation (i.e. single cycle execution) that works on different parts of the message. In that situation the state machine would get a data value from memory, modify it, and write it back. Simple overlapping of the read of the next data value with the modification of the previous data value can improve performance. This situation is especially valuable when the message can be worked on while it is arriving, and before it even gets written to memory. More complex elements would require more analysis of control flow (e.g. looping) and dependencies.

The description of elements in the XML based language includes the states and operations within the state as a node in a tree. The description includes minimal interface and control information in the description. Merging thread A into thread B would require the following steps:

1. Remove states from B and append to A.
    a. If any name conflicts arise, append a unique identifier to each of the “A” states and a different unique identifier to each of the “B” states.

2. Replace any “transition” in the “A” states that are to the start state of A to the start state of B.
3. Replace any transition in the “B” states that are to the start state of B to the start state of A.
4. Remove variables from B and append them to the variables section of A.
    a. If any name conflicts arise, check the definition of the conflicting variables (e.g. the size). If they match, no further correction needed. If they don’t match, append a unique identifier to each of the “A” variables and a different unique identifier to each of the “B” variables.
    b. For each of the variables with modified names, replace any operations that use that variable in “B” states with the variable name that was unique to B and similarly to the operations in “A” states.

From this point, evaluating the memory read and write patterns as well as the order of transitions can optimize the state machines further.

5.2.5 Run Elements in Parallel

Ideally, all optimizations could be done on the Click graph to follow the established model. However, there does not exist any way to express parallelism between elements that have the same control flow in Click. In Click there exists a mechanism to express multiple branches in a pipeline (for example each of the FromDevice elements are inherently parallel). However, what is desired is also the ability to express the parallelism of elements operating on a single message within the same pipeline flow. In other words, execute two stages of a pipeline in parallel. Shown in Figure 5.8 is an implementation of the graph in Figure 5.6 where instead of executing Lookup after the CheckIPHeader/DropBroadcasts/DecIPTTL element they are executed in parallel on the same message.

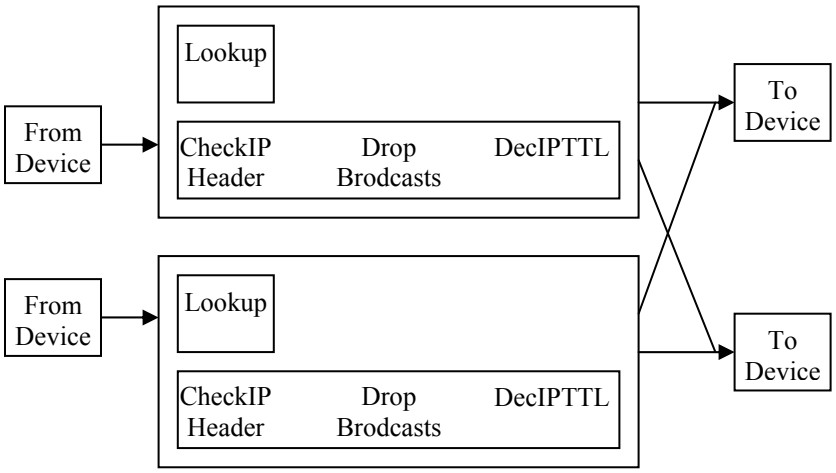

Figure 5.8. Transformed Click graph with implementation details for elements executing in parallel for the two port IP router. .

Since this requires implementation details, this optimization can only be done during the step of transforming a Click graph to an FPGA implementation. To achieve this using the programming model presented in this thesis is trivial and only requires the timing of the thread control to be modified. In this case FromDevice starts both the Lookup thread as well as the CheckIPHeader/DropBroadcasts/DecIPTTL thread instead of only starting the CheckIPHeader/DropBroadcasts/DecIPTTL thread. An additional state may be needed in the last element in the sequence, Lookup in this example, to make sure all of the other elements are finished.

5.3 Teja

5.3.1 Teja Application Design Environment

Teja Technologies develops a tool that currently targets application development for network processors [32]. In its most generic form, users can write arbitrary C/C++ code targeting a multiprocessor system with shared memory. A more targeted use of the tool allows users to create applications using modular data structures and program logic. It then allows users to

define the software architecture, define the hardware architecture, and finally define the mapping of the abstractions in the software architecture to the available resources in the target hardware architecture. There are four stages to development, each of which is discussed further in following sections:

- Software Library – defines the data structures and program logic.
- Software Architecture – defines instances of the data structures and program logic from the software library and communication between instances.
- Hardware Architecture – defines the resources and structure of the target platform. For example, in a network processor, the hardware architecture includes resources such as microengines and busses.
- Hardware Mapping – defines the mapping from the software architecture abstractions to the hardware architecture resources.

5.3.1.1 Software Library

It is in the software library that the user defines the data structures along with the functionality of the system. The Teja environment comes with a library that can be extended or used directly. There are three main constructs in the software library:

- Data structures
- Components
- Events

Data structures are objects that hold state and functionality to manipulate that state. To take an example, a queue provides a mechanism to store data in a first-in first-out mechanism. Abstractly, data is pushed into the queue at one end and pulled out at the other end. The functions “enqueue” and “dequeue” are the user’s interface to manipulate the data structure. Users can create new data structures and add them to the library or extend the functionality of existing data structures.

Components provide the user the ability to specify the functionality of one element in the system. It is implemented and designed as a state machine. The state machine has states with blocks of C code attached to them. It also defines the conditions and transitions between states. The user uses a GUI to define each of the aspects of the state machine, thus preventing the use of arbitrary and complex code.

Events are the communication mechanism between components. Events can be either multicast or unicast, synchronous or asynchronous, and can be used to pass data or for control purposes.

5.3.1.2 Software Architecture

Each of the elements in the software library is an object in an object oriented programming methodology. The software architecture defines the instances of the objects as well as defining the memory pools, irrespective of the target hardware platform. A memory pool is an abstract view of memory that can contain multiple memory elements as well as what objects reside in those memories. After defining the instances, the software architecture then defines another level of abstraction to provide containers for the instances. Threads are the containers for the functionality and memory spaces are the containers for data structures. Memory pools map to memory spaces and component instances map to threads. Finally the communication abstraction is also defined. A channel provides a mechanism for threads to communicate, using events, with each other.

5.3.1.3 Hardware Architecture

Just as the software architecture is independent of the target hardware, the hardware architecture is independent of the application that will run on the hardware. In this step, the target platform is defined in terms of hardware resources and connectivity of the resources. Resources include memory banks, busses, and chips. While described using board level terms, the hardware architecture can be a description of a "system on chip" architecture such as a network processor

which has multiple processors, busses, and memory all on the same device. In most instances this hardware architecture simply exists as an importable design. A given network processor is fixed in terms of hardware architecture and the user can reuse the description rather than redefining it each time.

5.3.1.4 Hardware Mapping

In the final step of development, the hardware architecture and software architecture are brought together. The user must map the abstractions in the software architecture to the resources in the hardware architecture. The mapping of threads and memory spaces is straightforward. One or more threads get mapped to chips (processors) and one or more memory spaces get mapped to memory banks. The complexity lies not in the gap between software abstraction and hardware resource, but rather in the placement. The placement of the threads can improve performance of the system by placing them such that the processor utilization is high. Furthermore, on architectures with memories local to the processor, it is desirable to place a data structure close to the thread that will use it.

The channels are more complicated in that there is not a single hardware resource to map to. Channels have several implementations to choose from and are architecture dependent. One example is the NNRingChannel. This is specific to the Intel IXP2xxx family of network processors by making use of the hardware support for next neighbor connections between microengines. Another example channel is the SignalChannel which only wakes up the associate thread rather than sending data as well. Other channels exist making use of shared memory or any other mechanisms available to communicate.

5.3.2 Mapping Teja to FPGAs

On first inspection, it does not appear that Teja's tool environment would be able to make use of the programming model presented in this thesis for mapping to FPGAs. It is an environment that currently only targets multiprocessor systems and makes use of the C/C++

language. The targeted platform and the targeted language are more closely matched with the hard and soft microprocessors available for FPGAs.

However, there are similarities between Teja's model and the model used in this thesis that upon further inspection provides a match that may lead to more efficient mappings to FPGAs. One obvious similarity lies in the targeted application domain of network processing. While targeting the same application does not mean that the design environments are similar, it does mean that the processing style could potentially match.

In Teja's environment, the programming model is based on state machines operating on data structures resident in memories. The memories can be distributed and there can be communication between threads. This programming model matches the programming model presented in this thesis closely.

Shown in Figure 5.9 is a summary of the proposed design flow for the Teja environment including the necessary modification needed to use the XML based flow presented in this thesis to map to an FPGA. The process starts with defining the software library. A compiler is needed to compile the C code for the state machines to the XML based instructions. Together with the preexisting data structures library, written in the XML based language, this forms the software library. Next, the user designs the software architecture using a GUI and the software library. This will be saved in an internal format and requires no modifications. The hardware architecture is then defined by the user using the GUI and the available resources such as threads, DPMem, SharedMemory, FIFO, and AlignedChannel. This is also saved in an internal representation requiring no modifications. Finally, the user defines the hardware mapping using the GUI. This will map the software architecture to the hardware architecture. It is in this step that the code from the data structure library is inserted into the code from the thread library. After the data structure code is inserted, the mapping can take place. Mapping may require merging multiple threads if the user specified this behavior. The output is the complete system described in the XML based language.

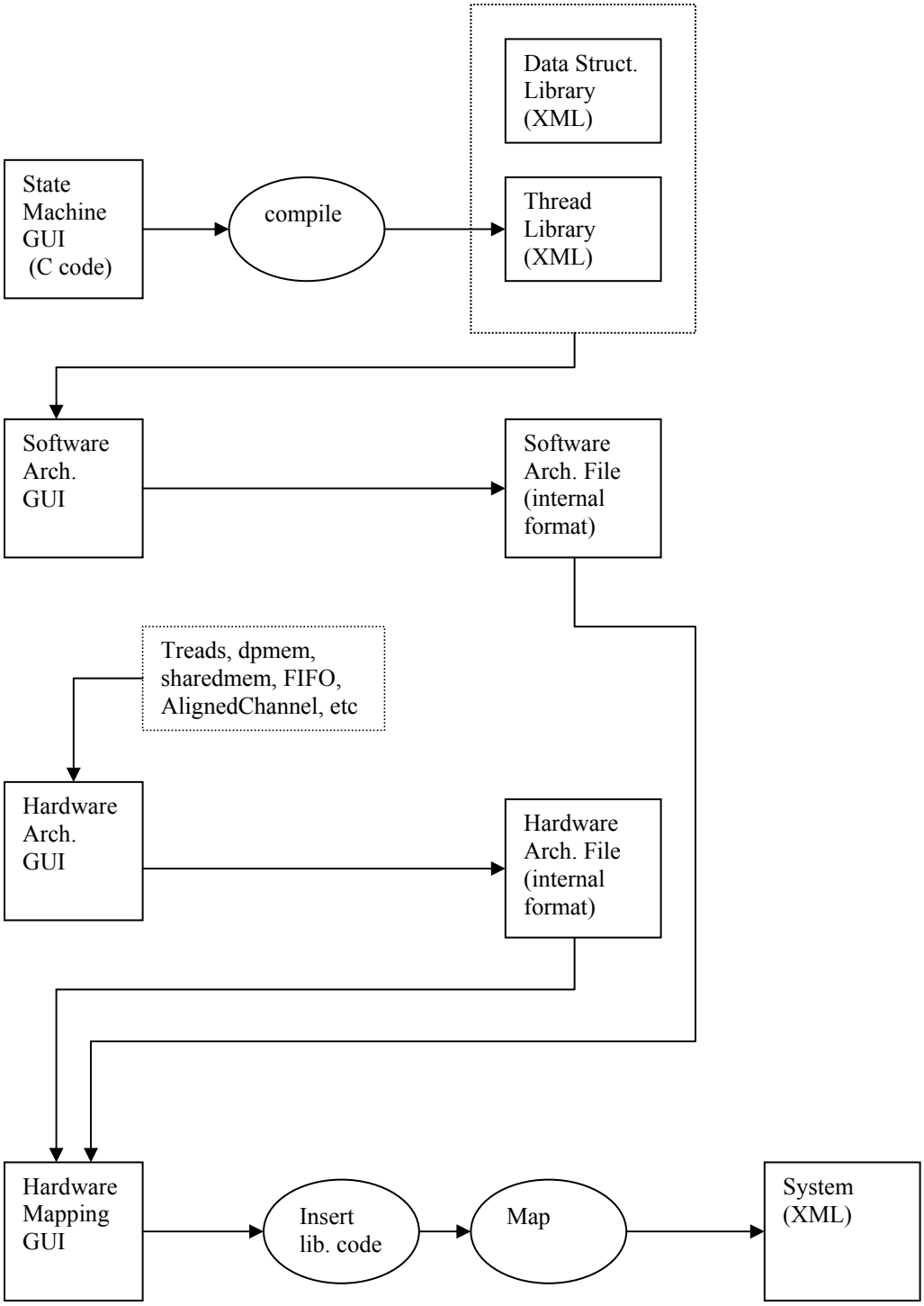

Figure 5.9. Summary of process to map Teja to an FPGA using the XML based language presented in this thesis.

5.3.2.1 Software Library

Recall that the first step in the Teja environment is to create the software library. A set of common data structures already exists in the library. Users can make use of the library elements directly or extend it and add functionality. Extending data structures simply makes use of the object oriented design features such as inheritance. Since the language presented in this thesis is not a C++ based language, minor modifications to this model are required. For mapping to FPGAs the standard library of common data structures would be rewritten in the XML based language. Each of the data structures would be written as if it has its own memory element. The implementation of the data structure depends on the hardware resource that is being used. For mapping to network processors Teja assumes a shared memory type model whereas in FPGAs there are different options. For example a queue object that gets mapped to a FIFO or PutGetMem would have different use than one that gets mapped to a shared memory. In the FIFO case, the pointers and pointer updating are implemented in hardware and do not need code to do that. Enqueue or dequeue operations simply write or read the value from memory. Assuming the included library is extensive enough, for an initial implementation a restriction can be put in place that the data structures are not extensible. Of course, users that need to write new data structures can do so in the XML based language. This differs from the current design environment for network processors in that users write new data structures using the XML based language instead of using C/C++.

The user also defines components in the software library. These are designed as state machines with blocks of C/C++ code attached to each state, possibly including calls to the interface of the data structures. This step would require compilation. Instead of relying on the compiler for the targeted microprocessor, a simple compiler needs to be created to target the XML based language. The individual states would be compiled separately to simplify the task. Since the XML language presented in this thesis is assembly like the compilation step is not that complicated. Also since the implementation of the instructions of the XML based language is

dedicated hardware, the compiler does not need to do optimizations that are based on pipelined microprocessors that execute instructions stored in memory such as branch prediction or loop unrolling. Instead, the operations in the code, such as the add operator '+', can be mapped directly. Registers can be created as necessary and are not limited to the number available on the target processor. Loops can be handled through the conditional statements and transition statements in the XML based language that will transition to one state if the condition is not met and another if the condition is met. Conditionals can be handled directly in XML based language through the conditional statements. Memory accesses would be implemented with the use of load and store instructions, called READ_DATA and WRITE_DATA in the XML based language. Each component appears as if it has its own memory space. So a "usemem" tag would be defined and used. Finally, the API calls to the data structures would inline the code from the data structure in with the compiled code. However, this will be left until the hardware mapping stage as the library functionality may change depending on the hardware element used.

5.3.2.2 Software Architecture

The software architecture step does not have any particular features that are unique to the programming model presented in this thesis and therefore does not appear to require any modifications from its current form.

5.3.2.3 Hardware Architecture

The hardware architecture step allows the user to define the architecture of the target platform. For network processors, this exists as a loadable file since they are fixed platforms. For FPGAs, this step plays a more important role. Defining the architecture is the key difference between mapping to FPGAs and mapping to network processors. FPGAs are flexible in terms of hardware architecture and users can define one more closely matching their application. The Teja environment would need to add hardware elements that are available in the programming model

presented in this thesis: threads, different memory elements, interfaces, and connections/channels. Since the terms thread and channel already exists in Teja's environment, the versions targeting the abstractions presented in this thesis would need to be called something different.

5.3.2.4 Hardware Mapping

The hardware mapping step is fairly straightforward. The mapping from Teja threads to XML based threads can be a direct mapping since by this point they are the same. However, Teja for network processors has the ability to map multiple threads to a single microengine. Since the threads implemented in an FPGA are custom logic, mapping multiple Teja threads to a single hardware thread is not necessary. However, as discussed in Section 5.2.4 in the context of the Click state machines, it is possible to perform the mapping of multiple Teja threads to a single hardware thread in order to minimize the state machine control overhead. Instead of algorithmically determining when to do the merging, as was described for Click, this is done under the user's control.

Memory spaces get mapped to memory elements. This is also straightforward. Before this step, the "usemem" tags are associated with an object, acting as if there is one memory per object. Once the particular instance of the memory element where the object is located is determined by the hardware mapping process, the "usemem" tags for a thread would be modified to use that memory element. It is also at this point where the Teja tool would compare the object and memory type and generate the code accordingly, for example use the queue library making use of a hardware FIFO rather than implementing it using pointers to a shared memory.

The final elements that get mapped are Teja's channels. These are the most complicated in that they are the least generic. As previously mentioned, for Intel's IXP2000 family of network processors there are channels in Teja's environment that match the hardware resources such as next neighbor registers. A similar set of channel types available for mapping to FPGAs would also have to exist. These can make use of memory elements, such as a FIFO, direct

thread-to-thread communication making use of the connections tag, or make use of the channels available in the XML based programming model. Each one has different instructions used to send data, and therefore all communication before this point would need to be specified using Teja specific placeholder instructions. These would be replaced at hardware mapping time with the proper XML based language instructions. This is possible since the structure and validity of the XML based code, i.e. unknown instructions, does not matter until the actual design is generated when the system is built after hardware mapping.

5.4 Summary

This section discussed two example higher level tools that can make use of the programming model presented in this thesis for mapping to FPGAs. They both target network processing, giving a common domain with the model presented. Click originally was originally targeted at a uniprocessor system running an operating system such as Linux. Teja's environment is targeted at network processors. Both environments have unique features. Click is a modular design environment plugging together blocks. Teja's environment focuses on a system where a user defines state machines to manipulate data structures. Both, however, are based off of C code. Instead of relying on compilation of C to gates, an alternate flow was proposed for each environment making them more suitable for mapping to FPGAs using the XML based language while minimizing the impact to the user. Both implementations make modifications to the objects in the XML based programming environment, such as operations and interfaces of the threads. Due to this, the use of the API based flow would be useful as modifying a textual representation would be inefficient.

CHAPTER 6

EXPERIMENTAL APPROACH

6.1 Methodology

In order to test and demonstrate the capabilities of the programming interface and test the efficiency of the compilation process, four example applications were implemented. The designs were implemented using the XML intermediate format with direct coding of the XML. The VHDL files were then generated by a tool that parses the XML and uses the API to generate the hardware description. Using the generated VHDL files, the designs were then simulated with ModelSim to test functionality. After simulation and after running the output files through the back-end tools, the resulting bitstream was downloaded to an ML300 board [48] which has an XC2VP7 Virtex-II Pro FPGA on it [27]. The FPGA's multi-gigabit transceivers were connected, via an optical cable to a Netgear GA621 gigabit Ethernet network interface card inside a workstation. The workstation was running RedHat Linux 7.1. All traffic on the workstation was captured with the Ethereal packet sniffer. The workstation did not have any other network connections, so all traffic was with the FPGA. This chapter discusses the applications and Chapter 7 discusses the results.

6.2 Gigabit Ethernet to Aurora Bridge

The first application implemented was a gigabit Ethernet to Aurora bridge. The bridge enables incoming frames received on the receive port of Aurora, a Xilinx proprietary point to point link layer protocol [26], to be translated to gigabit Ethernet frames, and then to be transmitted on the gigabit Ethernet's transmit port. The reverse flow is also supported. Received

gigabit Ethernet frames are then translated to Aurora frames and transmitted. Both Ethernet and Aurora make use of the multi-gigabit transceivers on the FPGA as the physical interface.

6.2.1 Gigabit Ethernet

Gigabit Ethernet is a standard defined in the IEEE 802.3z spec [25]. While mostly used for point-to-point communication, Ethernet does support broadcast capabilities. Due to this, addresses are required in order to determine the destination. The header of each frame consists of a 6 byte source address as well as a 6 byte destination address. The header also contains a two byte field that tells the type of the message. For example, a value of 0x800 would mean that an IP packet is contained in the payload of the frame. In addition to the header, the frame also contains a trailer that consists of a cyclic redundancy check value. This allows for error detection. Through the use of special frames, called pause frames, flow control can be supported. Flow control exists in order for a receiver to notify a sender that it has become overloaded. The sender will then refrain from sending any more frames until the receiver can handle them. This flow control operates on a per frame basis. A receiver cannot stop transmission in the middle of a frame. Instead, it will apply to future frames.

Shown in Figure 6.1 is the interface of the Gigabit Ethernet interface block that was used in the experimentation. The gigabit Ethernet Media Access Controller core from Xilinx was manually wrapped up to give an interface that is simplified for use in the programming interface. The programming interface could then make use of the preexisting netlist. Rather than using the entire interface of the core, the interface was restricted to what is shown in Figure 6.1. On the left hand side are the signals to the external system. The RXP, RXN, TXP, and TXN are the signals that attach to the multi-gigabit transceivers. The other two signals are the necessary clocks. On the right hand side are the signals that are used internal to the system defined by the programming interface. The signals prefixed with RX form the RX port. Similarly, the signals prefixed with TX form the TX port. When a frame is received, the GMAC core from Xilinx will check the

CRC. If there is an error, the badFrame signal will be raised. If there is not an error, the goodFrame signal will be raised. The data come in 8-bit values and are valid only when dataValid is high. The output clock is the clock that should be used by the thread that connects to the port. Similar signals exist in the TX port. The only new one is the ack signal. This is used by the core to tell when it is ready to accept a frame for transmission. Once it goes high, a byte of data has to be provided each cycle.

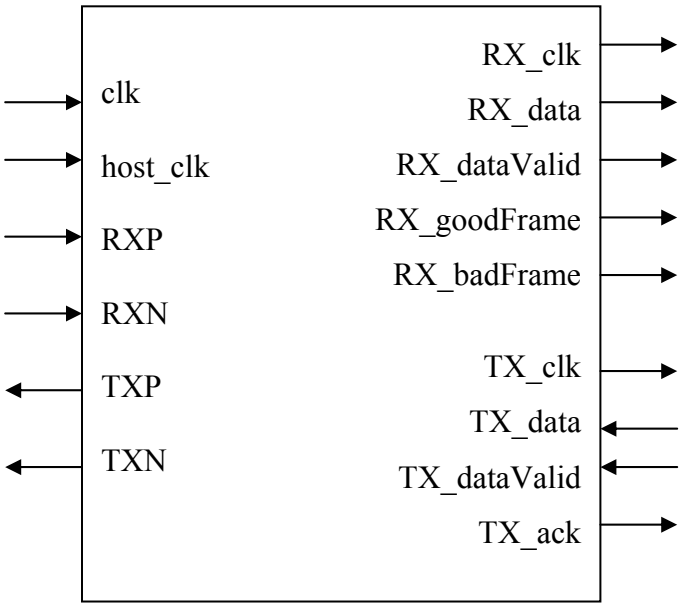

Figure 6.1. Gigabit Ethernet Interface.

6.2.2 Aurora

Aurora [26] is a protocol designed by Xilinx as a convenient way to make use of the multi-gigabit transceivers that are available. Unlike Gigabit Ethernet, Aurora is only meant for point to point transmissions. As such there is no addressing. Flow control is available. However it is on a per byte basis. With this capability, it is possible to pause the transmission of a frame. Due to the transmission delay, the pause may not happen for a few cycles.

As was done with the gigabit Ethernet core, the Aurora 201 core from Xilinx was manually wrapped up to create a netlist with an interface that is simplified for use in the

programming interface. The interface is shown in Figure 6.2. The startOfFrame signals are used to indicate the first data unit in a frame. The endOfFrame signals are used to indicate the last data unit of a frame. When the endOfFrame is high, the value on the rem signals will indicate the number of bytes that are valid in that last data unit. This allows for frames of odd size. The srcReady acts as a data valid signal and destReady notifies the sender that the Aurora core is ready to accept data for transmission.

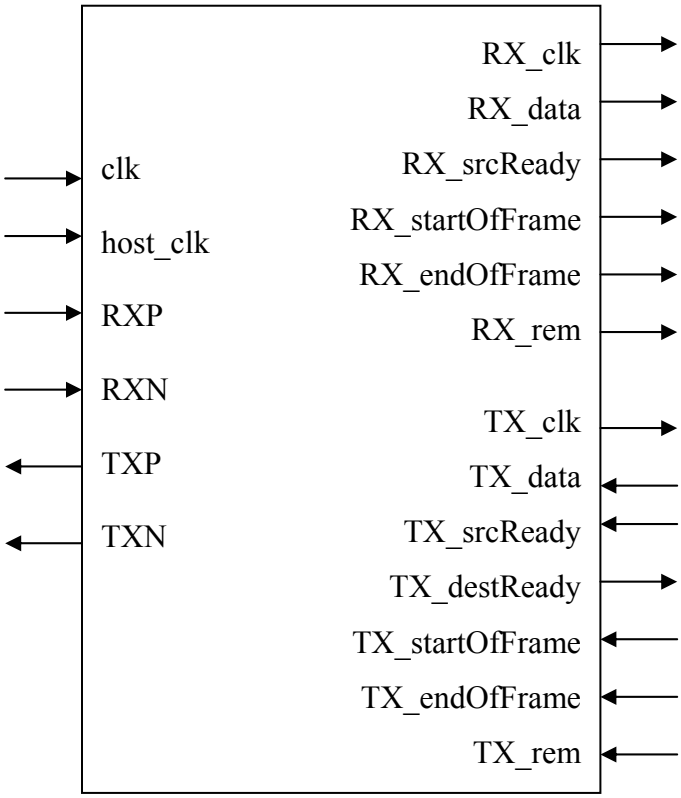

Figure 6.2. Aurora Interface

6.2.3 Architecture

Modeled after James-Roxby's implementation [28], the bridge is an example that uses many of the features of the programming interface. The XML intermediate format was used to implement this design by describing the functionality of the threads as well as the memory and interface architecture. As shown in Figure 6.3, there are two interface blocks: Aurora and Gigabit Ethernet Media Access Controller (GMAC). There are also four threads: the Aurora RX and TX

threads and the GMAC RX and TX threads. Each of the RX threads is responsible for reading the input message from the interface block using the signaling protocol of that interface and placing the message in a buffer. The threads are each started by their respective “data ready” signals from the interface blocks. When an entire and valid message arrives, the message is "committed" to memory. Otherwise the message is dropped and the values written to memory will be overwritten and never seen by the TX threads. When the commit occurs, the memories will have data available and the “nonEmpty” signal will be raised. This will trigger the TX thread to execute, if not already running. By doing this, the TX thread waits until a message is ready to be transmitted. It is then responsible for streaming the message out through the interface block with the proper frame format. The length of the message is known by the TX threads from the first value in memory. The RX thread will put the length in the first memory location and then the message in the subsequent locations.

For each of the two flows, there is a PutGet memory used. Since the Aurora interface uses a 16-bit data width and the GMAC interface has an 8-bit data width, the Put16Get8 and Put8Get16 variants of the memory are used. That means, using Put16Get8 as an example, that 16-bit data units are put into the memory and 8-bit data units are used on the get side.

As there are four ports in this example, there exist four independent clock domains. Each of the ports corresponds to a clock domain. The memories, being dual ported, serve as a boundary between the clock domains. Since the RX and TX ports of each of these particular interfaces have the same clocking requirements, Aurora at 62.5 MHz and GMAC at 125 MHz, the RX and TX threads can make use of the same clock.

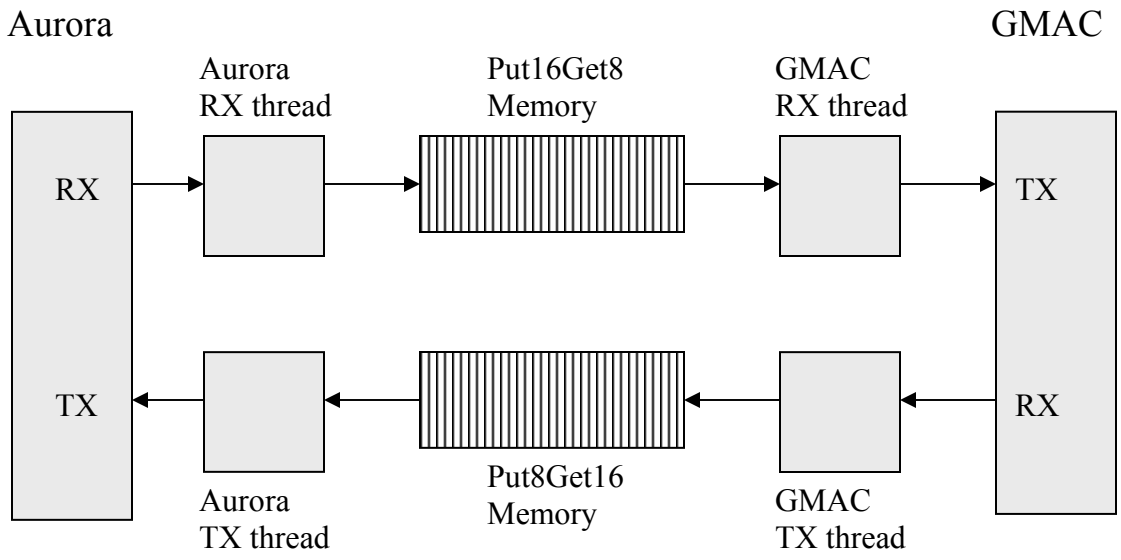

Figure 6.3. Gigabit Ethernet to Aurora Bridge Architecture.

6.3 Remote Procedure Call

The second application that was implemented to demonstrate the use and capabilities of the programming model was a remote procedure call server. To understand the processing associated with this application, an overview of the protocol is given. The discussion focuses on the functionality as found in software implementations, as that is currently the dominant implementation.

One model of communication relies on the exchange of messages between end systems. With this, however, the communication is central to the application and is not hidden at all. To obtain transparency, Birrell and Nelson introduced remote procedure calls (RPC) [24]. This model of communication allows a program running on one machine to call a procedure on another machine. In the common case of two workstations this is implemented using a client-server model. The calling function is the client and the called function is the server. When the client program makes a function call it places the parameters on the stack and jumps to the procedure. In the case of RPC, this procedure, known as the client stub, will remove the parameters from the stack and form a message to be sent over a network. The form of the

message adheres to the RPC specification [22]. Included in that specification is the format of how data types are represented. As one machine may represent integers in big-endian format and another represents it in little-endian formation, a common representation is needed. Once the message is ready to send, using UDP or TCP, the message is sent to the server. Upon reception, the operating system delivers the message to the server program. The entry point on the server is known as the server stub. This stub will unpack the message, place the parameters on the stack, and then jump to the procedure. When the procedure returns the server stub places the result in a message and sends it to the client. The client stub receives the result, unpacks it and then returns. To the application, it was as if the procedure was local. The diagram in Figure 5.4 sums up the process.

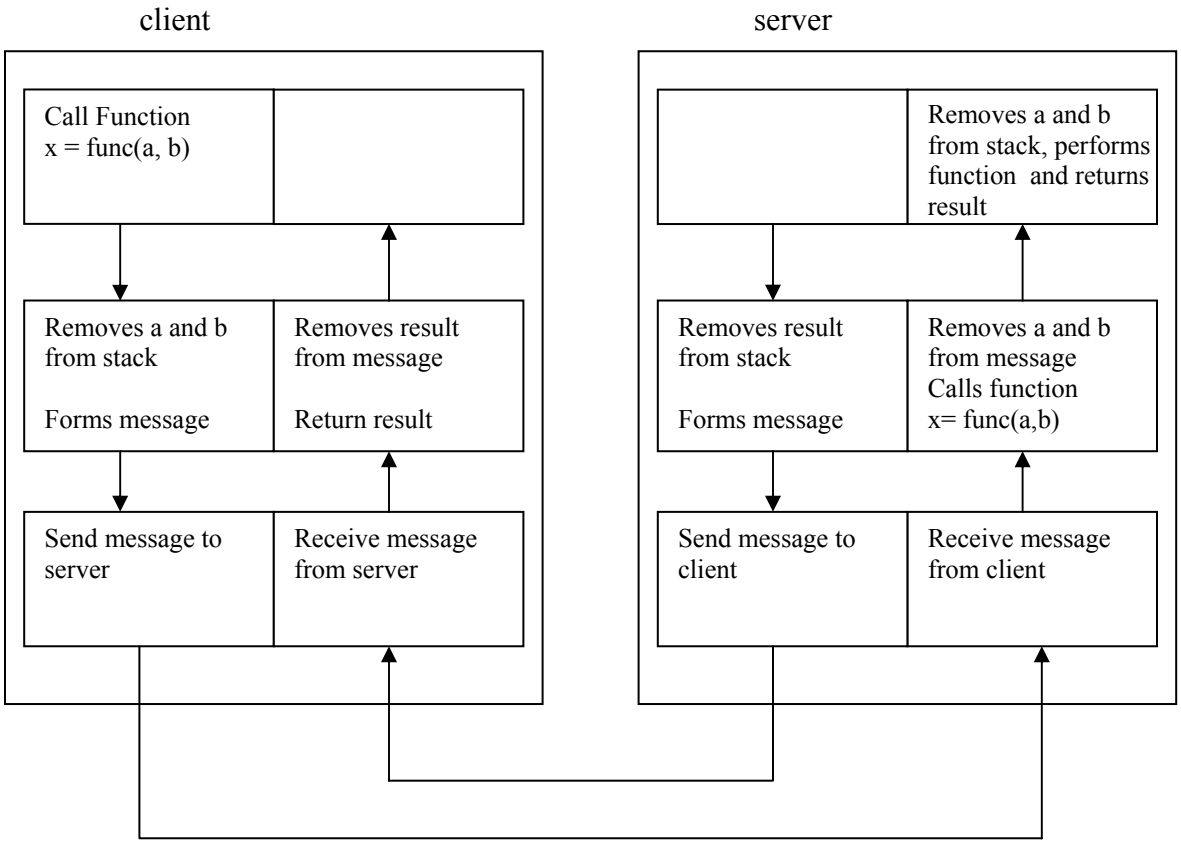

Figure 6.4. A step-by-step explanation of an RPC call.

The development of the RPC functionality was very centered on a workstation only distributed system. However, with the increasing use of FPGAs, it is reasonable to extend this model to FPGAs. This would provide for a truly heterogeneous distributed system by enabling hardware systems to use functionality that is only available to software systems. To explain an example, it would be useful to consider the network file system (NFS) [23], which is the most common use of the RPC model of computation. The procedures in NFS include opening a file, appending to a file, reading from a file. An FPGA that is capturing data could make use of RPC to provide a file system access. With the FPGA as the server, a workstation could see the status using UNIX commands such as more. With an FPGA client, the FPGA could log the status every so often to a file on the server workstation. Either method would enable this without any special software running on the workstation. In addition to embedded systems, this may have implications in storage area networks.

6.3.1 Implemented Design

In addition to the usefulness of RPC as a demonstrator of FPGAs, it also is a good demonstrator of the many features of the programming interface presented in this thesis. It makes use of several features that were not necessary in the bridge example. These include channels, direct communication between threads, an alternate stop state for cleanup, and the use of external intellectual property.

The XML intermediate format was used to define the implementation of an RPC server that has the arbitrarily chosen functions “int add(x, y) and “int mult(x, y).” In addition to the user defined function, the functionality of “find port” is also implemented [22]. Find port enables the client to lookup the port number of the program that implements the called functions. This is a single entry point for every possible program with RPC that is running. One mode of operation of find port is to simply return the port number associated with the requested function. Another mode is for the find port routine to actually call the function directly and return the result. Both

methods are supported. The networking protocols used are gigabit Ethernet, IP, and UDP. These provide the communication basis underlying the RPC functionality.

On the client end, the UNIX program rpcgen was used to create template software code that the client can use to call the functions. The functions and parameters were defined in the XDR definition language on the Linux workstation. This is then used by rpcgen to create the templates. While a version of rpcgen could be created that generates the FPGA code as well, that was crafted by hand.

6.3.2 Architecture

To motivate the architecture, as shown in Figure 6.5, the functionality of the system needs to first be described. A host workstation will make a function call using RPCs. This will then create an RPC message encapsulated in a UDP segment, an IP packet, and an Ethernet frame. The RPC message itself contains header information about the procedure being called. It then includes the parameters to the procedure. After the message arrives in the FPGA via gigabit Ethernet, the headers must be inspected and stripped off. The inspection is to verify the message did not get corrupted as well as to verify the destination of the message. After the header of the RPC message is processed, the parameters are passed to the hardware implementing the procedure. When this is finished, the result is then encapsulated in an RPC message, UDP segment, IP packet, and Ethernet frame for transmission.

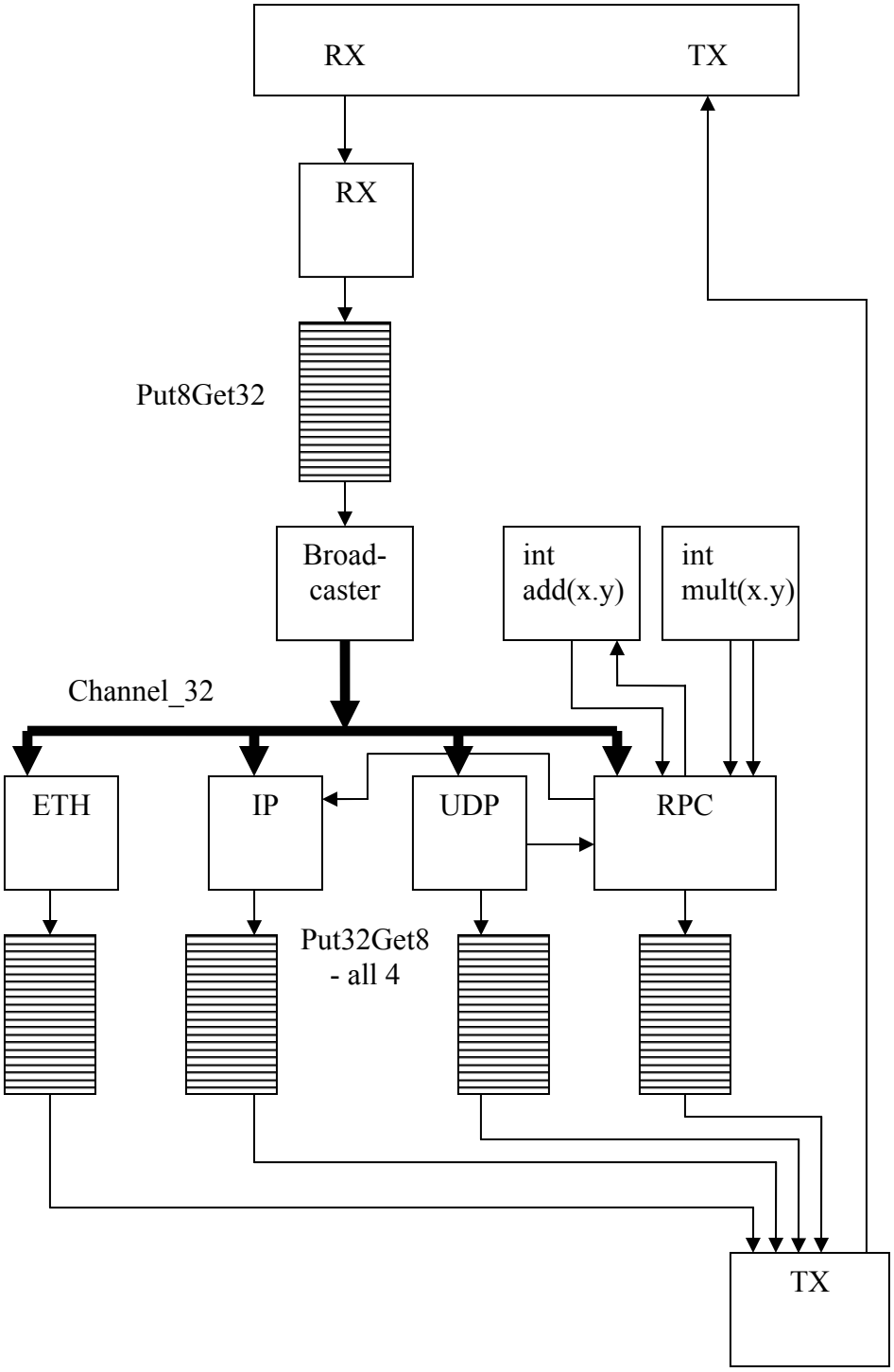

Figure 6.5. Architecture of RPC implemented on FPGA.

6.3.3 Component Descriptions

As can be seen from the architecture in Figure 6.5, the implementation is more complex than the bridge. Each of the labeled components is described with XML and gets compiled to a block of hardware. The network interface used for this application is the gigabit Ethernet, as discussed in the example of the bridge. The RX thread has the same functionality as the bridge. It is responsible for reading data from the interface and placing it in a buffer. The buffer is a Put8Get32 memory as the Gigabit Ethernet interface has an 8-bit data width and the threads were chosen to have a 32-bit data width. When the message is committed in memory, the broadcaster thread is activated from the nonEmpty signal going high. The broadcaster is simply responsible for reading from the memory and sending it through a channel. The thick link coming out of the broadcaster thread is the channel. The broadcaster thread sends 32-bit values through the channel to each of the other threads. The channel will deliver to the threads both the value that the broadcaster is sending as well as the previously sent value. It also outputs the address of the current data value. This allows the threads to wait for a specific field and not have to count cycles and possibly, as in the case where flow control is used, keep track of data validity. Sending the previous data value allows for the threads to read data on non-aligned word boundaries. For example, a thread that needs to read a word starting at byte 2 will read from the channel when the address is 4. This way the previous word will include bytes 2 and 3 and the current word will contain bytes 4 and 5.

The broadcaster starts the ETH thread when it starts sending data. The ETH thread is then responsible for checking the Ethernet header. It will check that the destination is correct as well as that the type is an IP packet. If the header is not valid, then the ETH thread will stop the broadcaster. The broadcaster thread makes use of an additional cleanup phase when it is stopped. To skip over the rest of the current packet in the buffer, the broadcaster will perform a commit operation on the Get port of the memory. This allows it to skip over any remaining data in the message and then wait for the next one. If the Ethernet header is valid, then the ETH thread starts

the IP thread. As previously discussed, starting a thread involves both a signal starting it but also a base offset for use with comparisons to the channel address. Meanwhile the ETH thread will produce the outgoing Ethernet header and place it in a buffer. However it will not commit it until it knows the complete message, not just the Ethernet part, is valid.

The IP thread is responsible for checking the IP header. Upon an invalid header, it will stop the broadcaster and ETH threads. When the header is valid it will start the UDP thread. It will then generate as much of the IP header that is possible and put it in a memory. One field is the length of the entire message. This is unknown and depends on the function called. Thus, it waits for the RPC thread to tell what the size is.

The UDP thread will check the UDP header. Included in this is checking the destination port. This information is used to tell which group of functions to call. In other words two different functions with the same ID can be in different programs. The UDP port is used to distinguish between then. This port is given to the RPC thread when the UDP thread starts it, as shown by a line between the two threads. Like the ETH and IP thread, the UDP thread will generate the outgoing UDP header.

The RPC thread will decode the RPC message and generate the return RPC message. The decoding involves deserializing the parameters and passing them to the function. This is through the included externally defined IP block functionality as discussed in Section 3.3.2. When the RPC thread can determine the length of the return message, which may be before the function is completed, it tells the IP thread. This is shown as a line connecting the RPC thread and the IP thread. Finally, when all four threads – ETH, IP, UDP, and RPC – commit their respective parts of the message into the memories, the TX thread then transmits the response. First, the TX thread reads the length of each of the partial headers from each of the four buffers. This can be done in parallel. It will then sequentially step through each buffer. First transmitting from the ETH buffer, then the IP buffer, then the UDP buffer, and finally the RPC buffer. The length is used to determine how many bytes to transmit from each buffer.

6.3.4 Improvements for the RPC Implementation

There are several improvements that can be made to the architecture. The first is that the buffer at the input is not necessary. The RX thread could directly send through the channel to each of the other threads. This was avoided for simplicity. Another improvement comes at the buffers at the transmit side. There is no need for each of the threads to have an output buffer associated with it. However, what is needed is the ability for each thread to write to memory, potentially simultaneously. This would require a shared memory, and as memory is left as future work to this thesis, it was not done.

6.4 IP Router

The third application that was implemented using the XML based language presented was an IP router. As with the previous examples, the XML was the input to a tool that automatically generated VHDL. The motivation for implementing IP routing is twofold. First, IP routing is one of the fundamental functions in networking. At a high-level, routers receive packets on one interface, perform checks, decide which interface to send the packet out on, and finally send the packet out on the chosen interface. Second, this thesis presents an intermediate platform and API for use by high-level tools in the networking domain to FPGAs. One such tool discussed is Click. There are a couple of example implementations on different platforms for an IP router. For this reason, it provides a means for comparison as is shown in Section 7.3.

6.4.1 Implemented Design

The implemented functionality for the IP router was based on two implementations in the literature that were based on Click. The first implementation was a simplified two port IP router based on the two port router from Koehler[13]. The two port version from Kulkarni, et al, used the functionality of the router from Koehler as a guideline but made a few simplifying assumptions [3]. The implementation by Kulkarni, et al uses a simple lookup algorithm that

succeeds in a single cycle, it does not generate any ICMP error message, and does not support ARP. The sixteen port version from Shah, et al, had the same simplifications as Kulkarni, et al, except that it had more complex IP address lookup [47]. Shah's version used a static routing table of roughly 1000 entries.

6.4.2 Architecture

The architecture of the two port IP router is shown in Figure 6.6. There are six total threads, two gigabit Ethernet interfaces, and four PutGet memory elements. The RX threads read the packet from the interface and place it in each of the attached PutGet memories. It also passes the packet to the corresponding Func thread. The Func thread will perform the necessary checks on the IP header such as verifying the checksum, decrementing the time to live field, and updating the checksum. For packets where an error has occurred, the Func thread will send a signal to the RX thread indicating this. Since the packet can still be arriving, the RX thread will continue to receive the data but when the last bit has arrived, it will not be committed to memory. This is equivalent to dropping the message. For packets that are not in error, the Func thread will perform a very simple lookup that can complete in a single cycle. It will then pass on the result to the RX thread. The RX thread will, at the end of the packet arrival, commit the packet in only the memory corresponding to the port chosen by the Func thread.

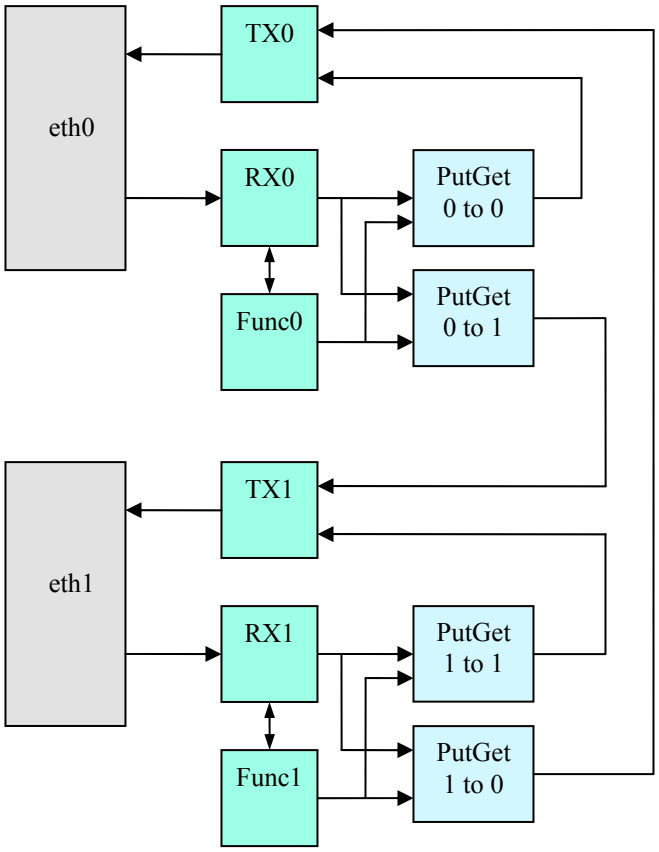

Figure 6.6. Architecture of a 2-Port IP Router implemented on an FPGA.

Due to the fact that there are only two ports, a fully connected memory structure was used with O(N^2) memory elements being required. For the router with sixteen ports, this was unreasonable. Instead, the DPmem memory structure was used. This is reflected in the diagram in Figure 6.7 showing the implementation of the sixteen port router. Note that only two ports are shown for clarity. In this version of the IP router, the RX thread will first obtain a free buffer area through the allocate mechanism of the DPmem element. When a packet arrives, the RX thread writes the data to the DPmem through its access port to the memory. It will also pass the received data to the Func thread. As with the two-port IP router, the Func thread will perform the verification of the IP header. However, the Func thread no longer performs lookup. It instead passes the address to the Lookup thread and gets the result back after a few cycles. The Lookup thread performs a simple binary tree search of the routing table that resides in a shared memory.

The shared memory is shared among four Lookup threads and thus there are a total of four for the sixteen-port router. When the lookup is finished the Lookup thread passes the result back to the Func thread, which in turn passes the result to the RX thread. The RX thread then passes the pointer along with the output port number to the buffer to the DP switch thread. The DP switch thread then places the pointer into the FIFO memory for the corresponding port. This causes the nonEmpty flag to go high. The TX thread will read the pointer from the FIFO. It uses that in memory accesses to the DPmem to read the packet from memory and transmit it. When the TX thread is done transmitting it will free the buffer for future use using the deallocate mechanism of the DPmem.

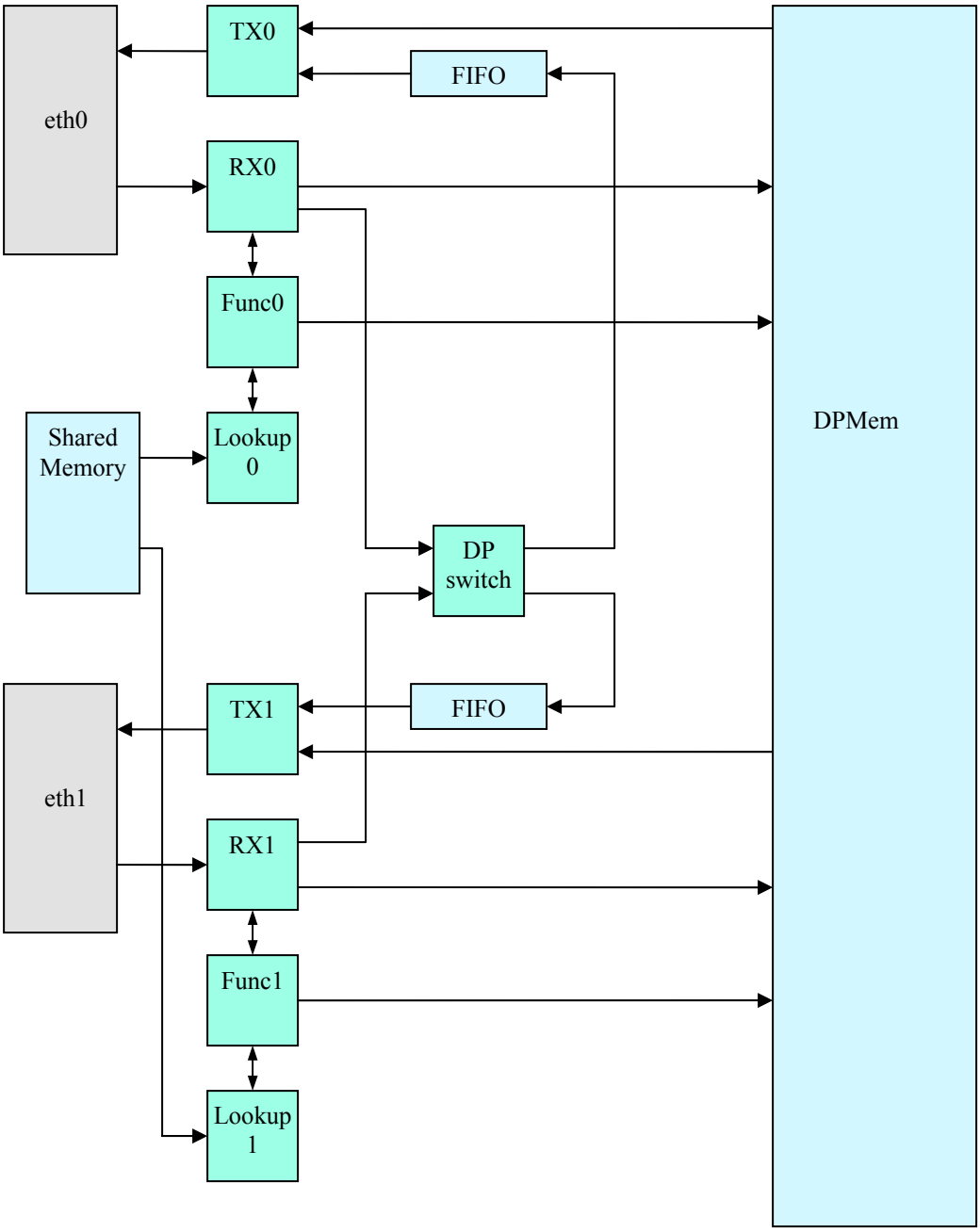

Figure 6.7. Architecture of 16-port IP Router implemented on an FPGA.. Shown are 2 representitive ports.

6.5 Network Address Translation

The fourth application that was implemented using XML and a tool to generate VHDL was a simplified network address translation (NAT). Just as with the IP router, NAT was chosen because of an implementation in Click that was previously mapped to an FPGA using the CLIFF tool [3]. A device that implements NAT provides a boundary point between an internal network and the external Internet. NAT will hide the internal network by translating the IP addresses from one used internally by the hosts to an externally visible address. Shown in Figure 6.8 is a diagram showing the functionality using a diagram based on available Click elements. The FromDevice element reads the packet from the network interface and passes it to the IPAddrRewriter element. The IPAddrRewriter implements the basic network address translation functionality rewrites. The IPFilter element in this case is similar to address lookup in that it filters packets depending on whether they are destined for the external network or the internal network. It also allows certain destination IP addresses to cause the packet to get dropped instead of forwarded.

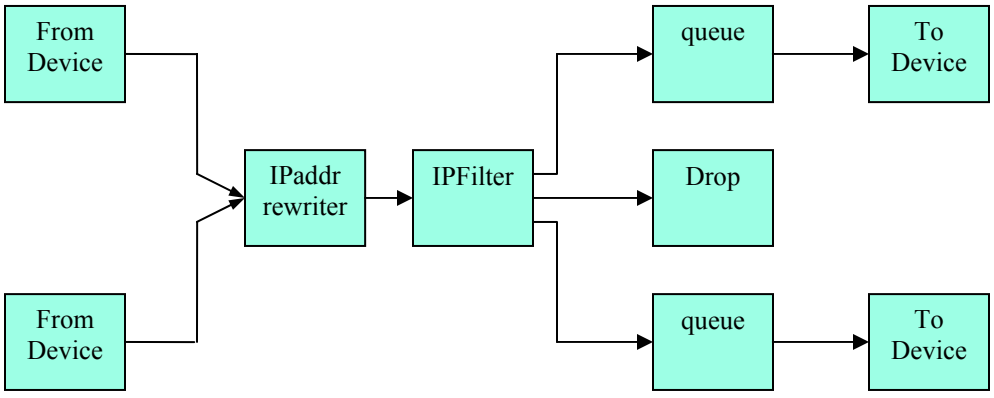

Figure 6.8. Diagram of click graph of the NAT application.

6.5.1 Implemented Design

The functionality was chosen to match the implementation by Kulkarni, et al [3]. Shown in Figure 6.9 is the block diagram of the XML based implementation. The GMAC interface was used for eth0, the internal network, and eth1, the external network. RX is a thread that reads from the interface and puts the packet in the outgoing PutGet memory element. RX also passes the packet header to the Func thread which performs all of the NAT functionality. The Func thread also writes to the corresponding PutGet memory element. The Func thread will also determine if the packet should be dropped and will send a signal to the RX thread to signify this. Since the packet can still be arriving, the RX thread will continue to receive the packet. To handle the dropping, it will simply not commit the packet to memory when the last bit arrives. If the RX thread does not get signaled to drop the message, it will commit the packet to the PutGet memory when the last byte of the packet has arrived and will return to the idle state waiting for the next packet. The PutGet memory will then contain the packet and the nonEmpty signal will go high. At this point, the TX thread will transmit it.

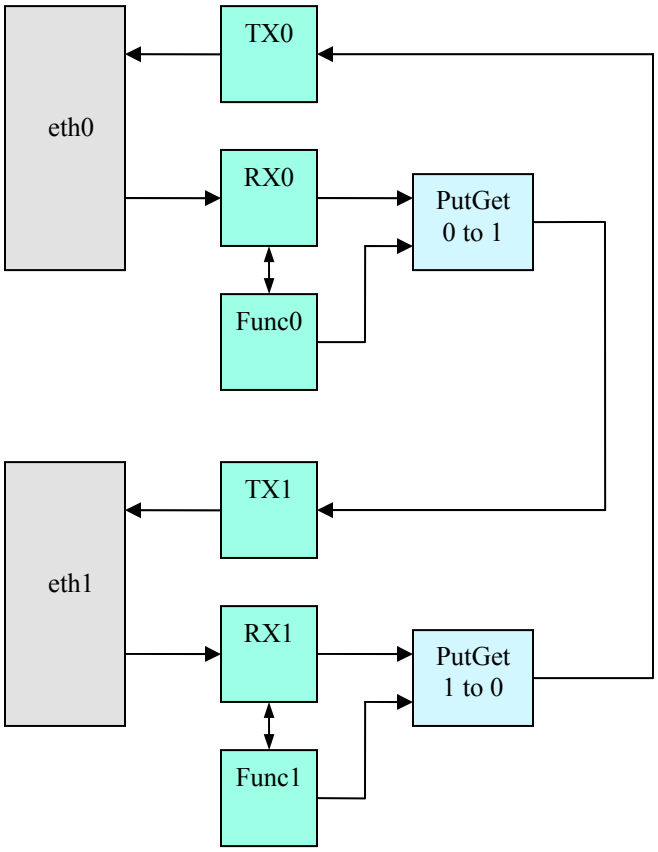

Figure 6.9. Architecture of NAT implemented on an FPGA6.9

CHAPTER 7

RESULTS

7.1 Introduction

In evaluating the effectiveness of both the tool and the architecture, many parameters are involved. Throughput is a common metric used in networking applications. In the case of each of the designs, the throughput is fixed at 1 Gbps due to the target interfaces. Further, the interfaces determine the performance requirements, i.e. clock frequencies, and the system only needs to go fast enough to meet those requirements. However, for evaluation purposes this requirement was relaxed and the maximum clock frequency obtainable was measured. All of the designs target an XC2VP7 Virtex II Pro FPGA with a –6 speed grade. This was chosen due to the fact that it is the FPGA available on the ML300 board which was used for all testing purposes. The ML300 board includes four optical connections that connect to the gigabit transceivers. A faster speed grade for the Virtex II Pro FPGAs exists that provides higher obtainable clock frequencies by roughly 15%.

Another metric that is often important is latency. For store and forward devices, latency is defined as the time from when the last bit of data arrives to when the first bit of data is transmitted [37]. This definition is used for all of the designs except for the RPC, which will instead use response time – defined as the time from when the first bit arrives to when the last bit is transmitted.

In addition to performance metrics, implementation on an FPGA also creates the need to consider area. Of interest, is the number of LUTs, flip-flops, and block RAMs used.

7.2 Gigabit Ethernet to Aurora Bridge

For store and forward devices, latency is defined as the time from when the last bit of data arrives to when the first bit of data is transmitted [37]. For the Aurora to GMAC flow the Aurora RX thread requires two extra cycles to add the length and commit the write to memory. The GMAC TX also requires two cycles to read the length. The total latency is therefore four cycles, two of which are at 125 MHz and two of which are at 62.5 MHz. This means the latency for the Aurora to GMAC flow is 48 ns.

In the GMAC to Aurora direction the latency also is four cycles, however the time is different. The GMAC RX thread requires an extra two cycles to write the length since it is writing a 16-bit value using an 8-bit interface to memory. An extra cycle is also needed to commit the packet to memory. Each of these cycles is for a 125 MHz clock. The Aurora TX thread requires one cycle to read the length using a 16-bit interface to memory with the clock frequency of 62.5 MHz. This totals 40 ns.

In addition to performance metrics, implementation on an FPGA also creates the need to consider area. Of interest, is the number of LUTs, flip-flops, and block RAMs used. The results are summarized in the Table 7.1. Also included in Table 7.1 are the results from James-Roxby's VHDL implementation of the bridge [28].

Comparing to the VHDL implementation, the throughput and latency were the same. The area was slightly greater using the API developed for this thesis. The overhead comes from the inability in the XML-based language to specify combinatorial circuits. Each of the operations are implemented within a state machine. This means there is extra multiplexing that would not be required if describing the functionality outside of the state machine. The maximum clock frequency obtainable was greater in the XML based version, though both were well above the required 125 MHz.

	Device	LUT	FF	BRAM	Freq.	Throughput	Latency (ns)
XML	XC2VP7-6	2,159	1,492	2	152.8	1 Gbps	40-48
VHDL [28]	XC2VP7-6	2,091	1,494	2	144.9	1 Gbps	40-48

Table 7.1. Comparison of XML implementation of the Aurora to gigabit Ethernet bridge using the programming interface from this thesis and the VHDL implementation from James-Roxby.

7.3 Remote Procedure Call

As in the case of the bridge the performance, latency, and area will be used as the relevant metrics for the implementation of the RPC protocol. In this case, there is not an equivalent implementation on an FPGA that can be used for comparison purposes. One way to analyze the performance is by comparison to the performance of an RPC call on a workstation, as that is the most common target for an RPC call. The results are summarized in Table 7.2.

For the FPGA implementation, the design required 4345 LUTs, 2084 flip-flops, and 5 BRAMs. There are two clock domains for this design. At the interface, the data arrives and is transmitted eight bits per cycle. To meet gigabit rates, a 125 MHz clock was required. Internally to the implementation, a 32-bit data width was used. To keep up with the data arrival rate, only a 31.25 MHz frequency was required. The obtained frequencies are shown in Table 7.2 showing the design runs at 127 MHz at the interface and 31.5 MHz internally. The final metric listed on the table is the response time. Response time is the time from when the first bit of data arrives to when the last bit of data is transmitted. This implies that the size of the packet impacts the response time. For the implemented functions, this packet size was fixed at 90 bytes and therefore the response time was calculated to be 2.16 us for each RPC call. This only includes the processing time on the FPGA and does not include the overhead on the caller.

	Device	LUT	FF	BRAM	Freq.	Throughput	Response Time
XML	XC2VP7-6	4,345	2,084	5	127 / 31.5 MHz	1 Gbps	2.16 us
Linux Stack	Pentium 4 (512 MB RAM)				2 GHz	1 Gbps	18.80 us

Table 7.2. Comparison of XML implementation of the RPC protocol using the programming interface from this thesis and the software implementation as part of Linux.

To compare the performance to a workstation-based server, two workstations with NetGear GA621 network cards were directly connected through a fiber optic cable. The client workstation then made 900 calls to the server workstation, which had a 2 GHz Pentium-4 processor and 512 MB of RAM. The total time for the 900 RPC calls was 192 ms. However, this includes the processing time on the client as well. To determine the overhead of the client, 900 calls were made to the FPGA-based system consisting of an ML300 board from the same client. This was also via optical fiber through the NetGear GA621 network card. The measured time was 177 ms. Using the above knowledge of the FPGA processing time (1.94 ms total for 900 calls), the overhead of the client workstation, which was substantially slower than the server workstation, was calculated to be 175.06 ms. This implies that the average response time of the workstation-based server was 16.94 ms total for 900 calls or 18.80 us per call. That represents an 8.7x speedup for the FPGA implementation over a high powered workstation. As the chosen functions (add and mult) are extremely fast with either technology, this processing time essentially only represents the protocol handling time.

7.4 Click Comparisons

In this section, the three applications based on Click will be grouped together. This includes the two-port IP router, the sixteen-port IP router, and the network address translation.

To evaluate the two-port router and NAT applications, the results are compared to the results from using CLIFF to map the Click descriptions to an FPGA [3]. The results reported for

CLIFF are in terms of slice count and did not attempt to maximize clock frequency as the required 125 MHz for gigabit Ethernet was met. For this purpose, the source code was obtained from the author and run through the tools for the purpose of this comparison. The results for both the two-port IP router design and the NAT design are summarized in Table 7.3. It can be seen that the area for the XML-based flow was smaller than each of the CLIFF implementations by 45% for both designs. The flip-flop count for the CLIFF implementations reflect the use of large registers containing the entire the entire header of the packet, and therefore for comparison purposes the LUT count was used to compare area. There are several factors for the substantial reduction in logic. The first reason is that there are less state machines in the XML based implementation than the CLIFF implementation. For the XML-based design, 6 state machines were used to implement the functionality that was done in CLIFF with 14 state machines corresponding to a one to one mapping between element and state machine. In addition to there being less state machines, there is also less state info for each state machine. The XML-based implementation works on the data as it arrives and therefore not much intermediate state is needed. In contrast, the CLIFF implementation can only start processing when the data has fully arrived. Finally, there is no pipeline registers passing the data pointer for each element in the XML-based implementation that CLIFF has.

The maximum obtainable frequencies for the CLIFF implementations were greater than the XML based implementation. This reflects the register based read and write access of the packet data in CLIFF that is passed between threads. Whereas the XML based implementation accesses data through the embedded block RAM. Access to memory has a higher delay than access to a register.

The final difference is in the latency where the XML based implementations are able to transmit the data a single cycle after the last bit arrives. In CLIFF, a long pipeline with handshaking caused a longer latency.

As an added comparison point, the two port IP router that was implemented on a Linux workstation was included in Table 7.3 [46]. The reported throughput was in terms of minimum sized packets (64 bytes) per second. Translating this to bits per second led to a throughput of 228 Mbps, or roughly 4 times lower than the FPGA implementations. The latency was also noticeably higher requiring 2500 ns to forward a packet compared to 8 ns for the XML-based implementation. Also shown is the performance of a higher powered dual processor system. The higher power system consisted of an AMD Athlon MP dual processor running at 1.6 GHz. Using the higher powered system increased the throughput to 379 Mbps. The latency for that system was not reported. It should be noted that the Click system is the full system and does not include the simplifications. While this will affect the performance of the FPGA implementation, it would be minimal since much of the removed functionality does not exist in the forwarding path. The lookup does not affect the XML-based implementations greatly as seen in the discussion of the 16-port router that does implement lookup. The area of the FPGA implementations would be more greatly affected.

	Design	Device	LUT	FF	BRAM	Freq. MHz	Thru. (Gbps)	Lat. (ns)
XML	IP Router 2 port	XC2VP7-6	4,052	2,402	8	138.7	1	8
CLIFF (org 1) [3]	IP Router 2 port	XC2VP7-6	7,385	9,063	2	144.3	1	224 to 248
Click [46]	IP Router 2 port	Pentium III	N/A	N/A	N/A	700	0.228	2500
Click [46]	IP Router 2 port	AMD Athlon-MP	N/A	N/A	N/A	1600	0.379	Not Reported
XML	NAT	XC2VP7-6	3,970	2,369	4	139.9	1	8
CLIFF [3]	NAT	XC2VP7-6	7,304	7,650	2	144.1	1	160

Table 7.3. Comparison of implementations of the Click designs IP router and NAT.

Shah, et al, implemented the functionality of the sixteen port router targeting an Intel IXP1200 network processor [47]. Their results are summarized in Table 7.4. The FPGA implementation written using the XML based language required 50,201 LUTs, 25,111 flip-flops,

and 57 BRAMs. Note that this, unlike the rest of the designs, required a larger device. A faster speed grade was also used. Even with the faster speed grade, the design was only able to obtain a clock frequency of 77.4 MHz due to the physical limitations of being able to access all 32 embedded block RAM. This falls short of the 125 MHz requirement. However, increasing the data path to 16 bits rather than 8 bits, would only require a 72.5 MHz frequency which could be possible. As the 8 bit version was implemented, the throughput is therefore scaled down for the slower clock speed. Comparing this to the throughput from Shah's design still represents roughly a speedup of 7.1. It should be noted that the design by Shah, et al, was implemented on an IXP1200. The latest network processor from Intel, the IXP2800, has higher performance. However, this design has not been ported to the IXP2800. It should also be noted that Shah reported the throughput in terms of system throughput over sixteen ports. The throughput used in the rest of this chapter have reported in terms of a single port and therefore Shah's results are divided down to be a per port number.

The latency for the FPGA implementation required anywhere from two cycles to eighteen cycles for passing the pointer through the switch to the output port, since the switch used a round robin scheme to check each of the input ports for a value to write to an output port. This translates to a range of 25.8 ns to 232.2 ns. The latency was not reported by Shah, et al.

	Device	LUT	FF	BRAM	Freq.	Thru. (Mbps) (per port)	Lat. (ns)
XML	XC2VP70 -7	50,201	25,111	57	77.4 MHz	619.7	25.8-232.2
Hand Coded in IXP-C [47]	IXP1200	N/A	N/A	N/A	232 MHz	87.5	Not Reported

Table 7.4. Comparison of XML implementation of the 16-port IP router using the programming interface from this thesis and the implementation by Shah.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

Presented in this thesis is a programming model for implementing networking applications on FPGAs. The model involves a programming language, presented as an API as well as an XML grammar, which provides the designer with abstractions using threads as the central construct. The use of threads is due to past research that has shown the suitability to networking applications. In addition to being able to specify functionality in terms of threads, the programming interface also provides the ability to specify architectural features. The features are restricted to those that are relevant to networking. Most notably, these include the interface for communication with the external system, the memory for buffering of messages, and communication between threads. The programming model also involves a compilation process that will automatically generate hardware from the specification using the programming language aspect of the programming model.

The programming interface, presented as an API or XML based language, is intended for use by domain specific high-level tools. This would allow these tools to map to FPGAs with reduced effort by providing an abstracted view of the FPGA.

Evaluating such a model is difficult as it addresses the subjective metric of ease of use. It is assumed that with the correct abstractions the effort will be reduced. However, in order to provide a quantitative evaluation, the efficiency of the mappings to the FPGA is key. Using the abstractions provided by the programming interface resulted in similar area and performance as the implementation using VHDL. This demonstrates that it is efficient. When compared to the CLIFF tool, a direct implementation from the Click language to an FPGA implementation, the

area was smaller by 45% for the implemented designs. The performance was slightly less, but both comfortably met timing constraints. The latency was lower for the XML based implementation than the CLIFF implementation.

A publication based on the work in this thesis titled "Programming a Hyper-Programmable Architecture for Networked Systems," has been accepted for publication at the IEEE International Conference on Field Programmable Technology (FPT). The conference is to be held December 6-8, 2004 in Brisbane, Australia.

8.2 Future Work

Since the programming interface is immature, it is assumed that future work will involve improvements. This involves expanding the functionality, improving the efficiency of mapping to the FPGA resources, and further refinement of the soft architecture. The most obvious omission from this work includes the abstraction and flexibility of the memory architecture. As buffering and data access is a key to performance and usefulness, this is an important addition that will need to be addressed.

In addition to expanding functionality, the abstractions provided by the programming interface need to be evaluated. As a goal of the programming interface was to enable higher level domain specific languages to target FPGAs, that will demonstrate the usefulness. This will serve to enable the mapping from that particular language to the FPGA as well as provide an evaluation of the interface. Numerous candidates exist that can be used. Examples include CloudShield's RAVE[30], MIT's Click [13], Novilit's Anyware [31], and Teja's Teja C [32].

Finally, one assumption that was made in this thesis was the benefits of using FPGAs. While this was assumed from observation of past research, there has not been a study comparing network processors, the most likely candidate for alternate implementation, to FPGAs. This

would serve to demonstrate if the flexibility of FPGAs provides enough benefits to outweigh the raw clock speed of network processors.

APPENDIX

LANGUAGE REFERENCE

A.1 Language

While the programming language is implemented using XML based syntax, a more syntax independent description is given. Following this description, the details on how this is implemented in XML will be given. The language is a tree of elements. Each element can have attributes associated with the element as well as children. The exact attributes and children depend on the type of element that it is. The attributes are discussed in Section A.2. To explain the possible children, shown in Figure A.1 is a graphical representation of the language. Each of the nodes represents an element. The characters '*', '?', and '+' attached to each node signify the cardinality of the element type. A node without any additional marking represents that there must be exactly one element of that type. A node that is marked with the '*' character represents there can be zero or more elements of that type. A node with the '+' character represents there must be one or more elements of that type. Finally, a node with the '?' character signifies that there must be either zero or one element of that type.

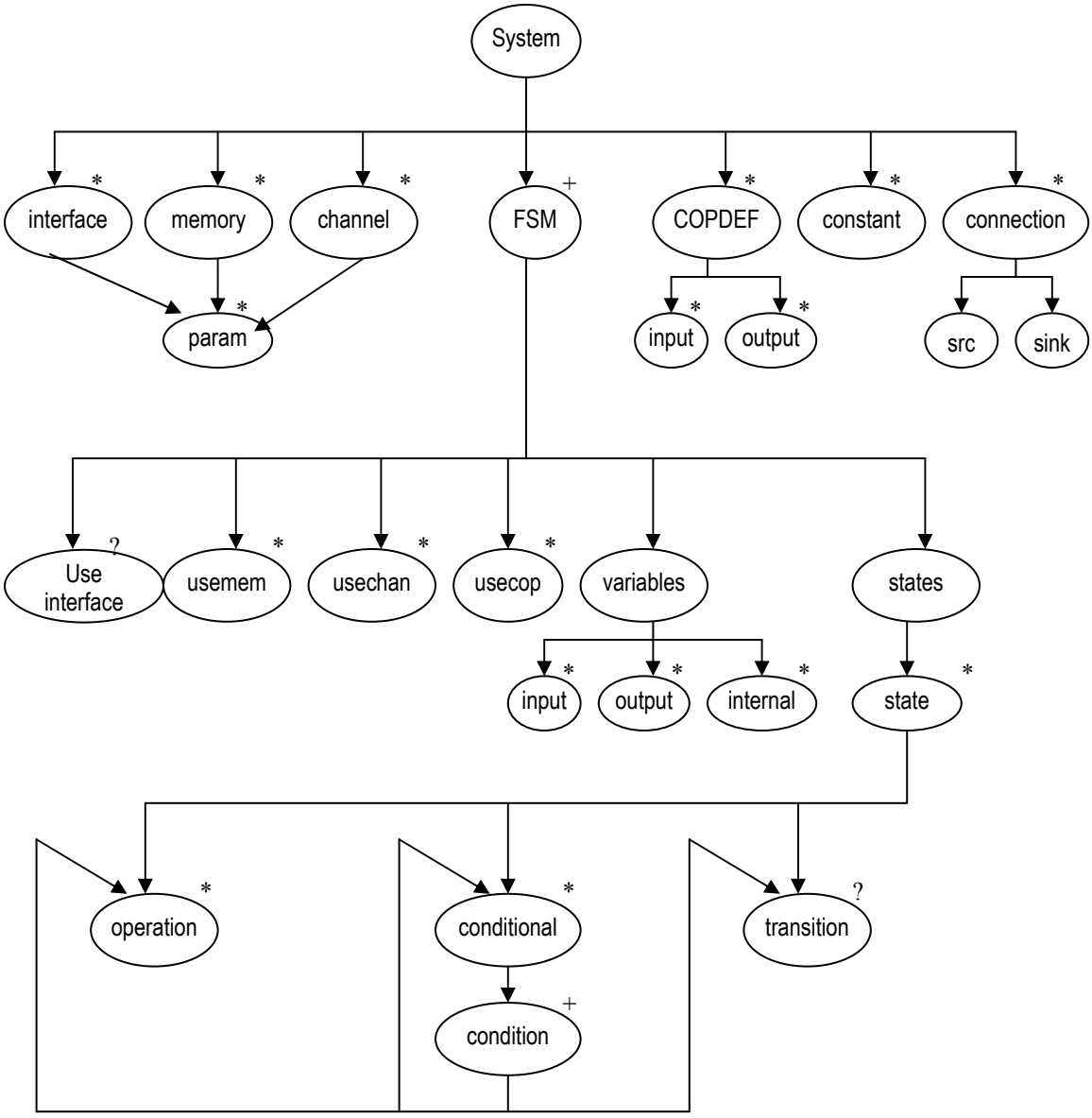

Figure A.1 Graphical view of programming language.

A.2 Details of Node Types

system

Expl: This is the top-level description of the application.

Attr: name – the name of the design

Attr [reset] – the name of the reset signal. It is optional and if not specified, the name "reset" is used.

interface

Expl: The interface is the method by which the system defined using this methodology communicates with the external environment (either on the same FPGA or off chip). It is a predefined block to be included in the system.
Attr: type – the name of a predefined interface block. Current choices include "GMAC" and "Aurora"
Attr: name – a string specific to the application that represents the given instance of the interface in the design.

memory

Expl: This is a block of memory. It is a predefined block to be included in the system.
Attr: type – the name of a predefined memory block. Current choices include "PutGet" or "SharedMemory".
Attr: name – a string specific to the application that represents the given instance of the memory in the design.

channel

Expl: This is a communication mechanism between threads where the complexity lies in the communication medium. It is a predefined block to be included in the system.
Attr: type – the name of a predefined channel. The sole current choice is "AllignedChannel".
Attr: name – a string specific to the application that represents the given instance of the channel in the design.

param

Expl: Parameters that can be passed to an element. The possible parameters are unique to the block that it is associated with. In the current implementation, only the PutGet memory is parameterizable.
Attr: name – the name of the parameter. For the PutGet memory the number of bits is specified with the "size" parameter, the bit width of the put port is specified with the "putwidth" parameter, and the bit width of the get port is specified with the "getwidth" parameter.
Attr: value – the value to be assigned to that parameter.

COPDEF

Expl: Defines the interface for a complex operation (cop) that is implemented as a predefined block.
Attr: type – the name of the predefined block that this interface describes.

COP

Expl: Provides a way to instantiate a complex operation (cop). This predefined block will be included in the system.
Attr: type – the name of the predefined block that is being instanced. Note that this must match one of the COPDEF element's type attributes
Attr: name – the name of the complex operation as it will be referenced by the system.

constant

Expl: Used to symbolically reference constant values in the system description.
Attr: name – the string that will be used to reference the constant.
Attr: val – the value of the constant. It can be in decimal, binary (prefix number with 0b) or hexadecimal (prefix number with 0x).

Attr: [width] – the number of bits used to represent the constant in hardware. For hexadecimal and binary, this is determined from the constant. For decimal, this must be given.
Attr: [type] – the type of the constant. If not specified the type will be either "std_logic" if the width was determined to be 1 or "std_logic_vector" if the width was determined to be greater than 1. In cases where the default is not correct, for example if the type is an integer, then the type must be given.

connection
Expl: An explicit communication connection between threads.
Attr: name – the name of the net that represents the connection.
Attr: width – the number of bits that the connection represents.
Attr: [type] – the type of the connection. If not specified the type will be either "std_logic" if the width was determined to be 1 or "std_logic_vector" if the width was determined to be greater than 1. In cases where that is not correct, the type must be given.

src
Expl: The source of a connection detailing the thread and port providing the data.
Attr: element – the name of the thread that will be driving the connection.
Attr: port – the port of the thread that will be driving the connection. The port must exist in the thread's output list as specified in the variable section.

sink
Expl: A sink of a connection detailing the thread and port receiving the data.
Attr: element – the name of the thread that will be driving the connection.
Attr: port – the port of the thread that will be driving the connection. The port must exist in the thread's input list as specified in the variable section.

FSM
Expl: The threads that provide all of the functionality in the system.
Attr: name – the name of the thread.

useinterface
Expl: The way to specify that a thread connects to an interface (i.e. the thread uses the interface).
Attr: intname – the internal name of the interface as it will be referenced in the rest of the thread (e.g. in operations).
Attr: name – the name of the interface as specified when instancing it with the instance node.
Attr: port – the port of the interface that the thread will communicate with. For the GMAC and Aurora interfaces there exists an RX (receive) and a TX (transmit) port.

usemem
Expl: The way to specify that a thread connects to a memory (i.e. the thread uses the memory).
Attr: intname – the internal name of the memory as it will be referenced in the rest of the thread (e.g. in operations).
Attr: name – the name of the memory as specified when instancing it with the memory node.
Attr: port – the port of the memory that the thread will communicate with. For the PutGet memory the ports are named PUT (for the write side of the buffer) and GET (for the read side of the buffer). For the SharedMemory memory the ports are named A and B and each have both read and write capabilities.

usechan

Expl: The way to specify that a thread connects to a channel (i.e. the thread uses the channel).
Attr: intname – the internal name of the channel as it will be referenced in the rest of the thread (e.g. in operations).
Attr: name – the name of the channel as specified when instancing it with the channel node.
Attr: port – the port of the channel that the thread will communicate with. For the PutGet memory the ports are named PUT, for the write side of the buffer, and GET, for the read side of the buffer. For the AlignedChannel channel the ports are named src, for the write side of the channel, and sink, for the read side of the channel.

usecop

Expl: The way to specify that a thread connects to a cop (i.e. the thread uses the cop). Note that a port is not specified, as is the case in usemem, usechan, and useinterface. With cop elements, there exists only a single interface and it is not partitioned into ports.
Attr: intname – the internal name of the cop as it will be referenced in the rest of the thread (e.g. in operations).
Attr: name – the name of the cop as specified when instancing it with the cop node.

variables

Expl: The tag that allows for variables to be defined.

input

Expl: An input to a thread.
Attr: name – the name of the variable as it will be reference in the rest of the thread (e.g. in operations).
Attr: width – the number of bits needed to implement the variable in hardware.
Attr: [type] – the type of the variable specified the type will be either "std_logic" if the width was determined to be 1 or "std_logic_vector" if the width was determined to be greater than 1. In cases where that is not correct, the type must be given.

output

Expl: An output of the thread.
Attr: name – the name of the variable as it will be reference in the rest of the thread (e.g. in operations).
Attr: width – the number of bits needed to implement the variable in hardware.
Attr: [default] – an optional value that specifies what the variable should be assigned to in cases where the operations do no explicitly assign a value to it. The default value is 0.
Attr: [type] – the type of the variable specified the type will be either "std_logic" if the width was determined to be 1 or "std_logic_vector" if the width was determined to be greater than 1. In cases where that is not correct, the type must be given.

internal

Expl: A local variable of the thread that is only used internal to the thread.
Attr: name – the name of the variable as it will be reference in the rest of the thread (e.g. in operations).
Attr: width – the number of bits needed to implement the variable in hardware.
Attr: [default] – an optional value that specifies what the variable should be assigned to when it is reset. The default value is 0. Note that unlike output variables, internal variables retain their value even when not explicitly assigned to.

Attr: [type] – the type of the variable specified the type will be either "std_logic" if the width was determined to be 1 or "std_logic_vector" if the width was determined to be greater than 1. In cases where that is not correct, the type must be given.

states

Expl: The tag that specifies the behavior of the state machine control.
Attr: start – the start state of the thread. This is the state that will be executed first. It must be the name of one of the state elements that are children of the states element.
Attr: [altstart] – an alternative start signal. By default the thread can only be started by another thread through the synchronization signal start. This allows an alternative signal to start the thread. One example would be a data available signal from a FIFO.
Attr: [alwaysrun] – a tag that allows the thread to be always running when set to "true". In other words there is no starting or stopping this thread. In this case the start state only has meaning when the thread starts up. It is default to "false"
Attr: [altstart_activelow] – for the alternate start signal, this allows for the signal to be active low. By default only when the signal goes high is the thread started. With this attribute set to true, the thread starts when the start signal goes low.
Attr: [stop] – the optional state that the thread will jump to when being stopped. Allows for a destructor-type behavior.

state

Expl: The tag that starts a state definition.
Attr: name – the name of the state.

transition

Expl: Specifies which state will execute next.
Attr: next – the name of the state that will be executed next.

operation

Expl: The instructions that implement the functionality of the threads.
Attr: op – the instruction that is to be executed.
Attr: params – the parameters to the instruction.
(see table A.1 f or a full description of valid op values along with the associated params)

Instruction (op)	Parameters (params)	Explanation
GENERAL OPERATIONS		
ADD	dest, src1, src2, src3…	Adds all of the sources (src1, src2, src3, …) and places the result in dest
SUB	dest, src1, src2, src3…	Performs a series of subtractions on the sources as listed from left to right and places the result in dest. The subtraction is (src1 – src2 – src3 – …)
CONCAT	dest, src1, src2, src3…	Concatenates each of the sources and places the result in dest. The source listed first (src1) will be the high order bits.
ASSIGN	dest, src	Assigns src to dest.

BIT_INV	dest, src	Performs a bit-wise inversion of src and places the result in dest.
CHANNEL SPECIFIC OPERATIONS		
CHAN_PUT	channel, value	Puts a value into the channel. As a given thread can have multiple channels, the internal name given to the port that connects to the channel is first given.
CHAN_PUT_FIRST	channel, value	Puts a value into the channel. Using this instruction signifies that it is the first value and should be assigned address zero. As a given thread can have multiple channels, the internal name given to the port that connects to the channel is first given.
CHAN_GET	channel, dest, address	Gets a value at address from the channel and places it into the variable dest. The thread will wait until the channel has the data for address before continuing execution. As a given thread can have multiple channels, the internal name given to the port that connects to the channel is first given.
THREAD - SYNCHRONIZATION OPERATIONS		
START	thread_name, [offset]	Starts the thread named thread_name. The optional parameter offset allows the thread to specify a base address to use when reading from a channel. If not specified, it defaults to zero.
STOP	thread_name	Stops the thread named thread_name. Depending on how thread thread_name is defined, this could stop it immediately or cause it to jump to a destructor state before stopping completely.
MEMORY SPECIFIC OPERATIONS		
WRITE_DATA	memory, data, [flag], addr	Writes the value in data to the memory specified by memory at the address addr. An optional value can be given and used as a program specific flag. Note that the parameter memory is the internal name of

		the memory port as used by the thread.
READ_DATA	memory, data_dest, [flag_dest], addr	Reads the value from memory specified by memory at the address addr into the variable data_dest. If the optional flag was written into the memory then it can optionally be read into the variable flag_dest. Note that the parameter memory is the internal name of the memory port as used by the thread.
COMMIT_WRITE	memory, num	Commits num data units into the memory identified by memory on the write port. Note that the parameter memory is the internal name of the memory port as used by the thread.
COMMIT_READ	memory, num	Commits num data units into the memory identified by memory on the read port. Note that the parameter memory is the internal name of the memory port as used by the thread.
LOCK	memory, addr	Locks the data value at addr in the memory named memory. Note that the parameter memory is the internal name of the memory port as used by the thread.
UNLOCK	memory, addr	Locks the data value at addr in the memory named memory. Note that the parameter memory is the internal name of the memory port as used by the thread.

Table A.1. Details the attribute values for the operation element type. The first column ("Instruction") is the string used as the value for the op attribute. The second column (Parameters) is the string used as the value for the params attribute. Note that the given string (e.g. "dest, src") needs to be replaced with one that is specific to the design. For example to assign the value in the user defined variable named "variable1" to the user defined variable named "variable2" would have a op attribute of "ASSIGN" and a params attribute of "variable1, variable2". The table is partitioned into the categories: general operations, channel specific operations, thread-synchronization operations, and memory specific operations.

conditional

Expl: The tag that represents a conditional. A conditional is an if – else if – else mechanism.

condition

Expl: A single condition in a conditional.

Attr: [cond] – the type of comparison that will determine if the condition gets executed or not. If it is not specified it "else".

Attr: [params] – the parameters to be passed to the condition instruction. It is specific to the condition specified with the cond attribute. If it is not specified an empty parameter set is passed.

(see table A.2 f or a full description of valid cond values along with the associated params)

Instruction (cond)	Parameters (params)	Explanation
GENERAL CONDITIONS		
EQUAL	a, b	The result is true if a equals b
NOT_EQUAL	a, b	The result is true if a does not equal b
BETWEEN_INCLUSIVE	a, b, c	The result is true if a is greater than or equal to b and less than or equal to c ($b \leq a \leq c$).
LESS_THAN	a, b	The result is true if a is less than b.
GREATER_THAN	a, b	The result is true if a is greater than b.
MEMORY SPECIFIC CONDITIONS		
GOT_LOCK	memory	Returns true if the lock granted signal for memory is high, signifying that a lock was acquired.
DATA_AVAIL	memory	Returns true if the nonEmpty signal for memory is high, signifying that the buffer has data.
THREAD-SYNCHRONIZATION CONDITIONS		
IS_FINISHED	thread_name	Returns true if the thread thread_name is not running (i.e. it is finished).

Table A.2. Details the attribute values for the condition element type. The first column ("Instruction") is the string used as the value for the cond attribute. The second column (Parameters) is the string used as the value for the params attribute. Note that the given string (e.g. "a,b") needs to be replaced with one that is specific to the design. For example to compare a user defined variable named "variable1" with the constant "0x0FFF" would have a cond attribute of "EQUAL" and a params attribute of "variable1, 0x0FFF". The table is partitioned into the categories: general conditions, channel specific conditions, and thread-synchronization operations.

A.3 Implementation with XML

XML is a markup language that presents a hierarchy of nodes through the use of tags and attributes. The extensibility of XML comes from the fact that the tags and attributes can be defined to fit a specific use. This is unlike a markup language such as HTML where the tags and

structure are predefined and geared toward web documents. To describe a node in XML involves the syntax as seen in Figure A.2:

```
1: <element-type1  attribute1-name="attribute1-value"
2:                 attribute2-name="attribute2-value">
3:    <element-type2 attribute1-name="attribute1-value"
4:                   attribute2-name="attribute2-value"/>
5: </element-type1>
```

Figure A.2 Example XML Syntax.

The first thing to note is the opening tag (element-type1). This tag is the element type and in the language presented in this thesis includes all of the nodes shown in Figure A.3 (e.g. "System", "FSM", "operation"). Following the opening tag is a list of attribute names (attribute1-name) with associated values (attribute1-value). As described in Section A.2, the element type defines the possible attribute names. The second thing to notice is the difference between the node with the element-type1 tag and the node with the element-type2 tag. The one node has a child and the other does not. In XML, a node with no children can be closed with the '/' character immediately following the attribute list. For nodes that have children, a tag marking the end of the list of children must be given. In this case it is "</element-type1>."

XML tags and attributes can be geared to specific use. In this case that use is to define network processing applications targeted to FPGAs. Shown in Figure A.2 is sample XML code that makes use of the tags and attributes as discussed in this appendix.

```
1: <system  name="example1">
2:    <FSM name="thread1">
3:       ...
4:    </FSM>
5:    <FSM name="thread2">
6:       ...
7:    </FSM>
8: </system>
```

Figure A.3. Sample XML code making use of specific tags.

BIBLIOGRAPHY

[1] G. Brebner, "Single-chip Gigabit Mixed-version IP Router on Virtex-II Pro" The 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. Napa Valley, VA. April, 2002

[2] G. Brebner, "Multi-Threading for Logic-Centric Systems" 12th International Conference on Field-Programmable Logic and Applications, Montpellier, France. September 2002.

[3] C. Kulkarni, G. Brebner, G. Shelle, Mapping a Domain Specific Language to a Platform FPGA. Submitted to the 41st Design Automation Conference, San Diego, CA, June 2004.

[4] C. Patterson, "High Performance DES Encryption in Virtex FPGAs using JBits" The 8th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, CA, April 2000.

[5] IEEE Standard VHDL Language Reference Manual 1076, 2002.

[6] IEEE Standard Description Language Based on the Verilog Hardware Description Language 1364, 2001

[7] P. Bellows, B. Hutchings, "JHDL An HDL for Reconfigurable Systems" The 6th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. Napa Valley, CA, April 1998.

[8] S. Guccione, D Levi. XBI: A Java-Based Interface to FPGA hardware. In John Schewel, editor, Configurable Computing: Technology and Applications, Proc. SPIE 3526, pages 97-102, Bellingham, WA, November 1998.

[9] S. Guccione, D. Levi. Run-Time Parameterizable Cores, Field-Programmable Logic and Applications, pages 215-222. Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications. Lecture Notes in Computer Science 1673. August/September 1999.

[10] Celoxica, Inc. "Handel-C Language Reference Manual". 2003.

[11] S. Young, P. Alfke, C. Fewer, S. McMillan, B. Blodget, D. Levi. "A High I/O Reconfigurable Crossbar Switch," In The 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, CA, April 2003.

[12] D. Davis, "Forge: High Performance Hardware from High-Level Software", White Paper, September 2002.

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, "The Click modular router" In ACM Transactions on Computer Systems 18(3), August 2000, pages 263-297.

[14] J.W. Lockwood, J. Moscola, D. Reddick, M. Kulig, and T. Brooks. "Application of Hardware Accelerated Extensible Network Nodes for Internet Worm and Virus Protection," International Working Conference on Active Networks (IWAN), Kyoto, Japan, Dec. 2003.

[15] D.E. Taylor, J.W. Lockwood, T.S. Sproull, J.S. Turner, D.B. Parlour. "Scalable IP Lookup for Programmable Routers" IEEE Infocom 2002, New York NY, June 23-27, 2002.

[16] I. Hadzic, J.M. Smith, "P4: A platform for FPGA implementation of Protocol Boosters", 7th International Workshop on Field Programmable Logic and Applications, Sept. 1997

[17] J. McHenry, P. Dowd, T. Carrozzi, F. Pellegrino, W. Cocks. "An FPGA-Based Coprocessor for ATM Firewalls" The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. Napa Valley, CA, April 1997.

[18] Symantec Corp. http://www.symantec.com, 2004

[19] McAfee. http://www.mcafee.com, 2004

[20] H. Fallside, M. Smith, "Internet Connected FPL" 10th International Conference on Field Programmable Logic and Applications, Villach, Austria, Sept 2000.

[21] D. Lyons, "Chipping Away", In Forbes Magazine, April 2003.

[22] Sun Microsystems, Inc. "RFC 1057 - RPC: Remote Procedure Call Protocol Specification version 2" June, 1988.

[23] Sun Microsystems, Inc. "RFC 1094 - NFS: Network File System Protocol specification" March 1989.

[24] A.D. Birrel and B.J. Nelson. Implementing remote procedure calls. ACM Transactions on Computer Systems, 2(1):3959, February 1984.

[25] IEEE 802.3z Standard, July 1998.

[26] Xilinx, Inc. Aurora_201 Reference Design version 1.2 User Guide. October 2003.

[27] Xilinx, Inc. Virtex-II Pro Data Sheet, 2003.

[28] P. James-Roxby, “Gigabit Ethernet to Aurora Bridge” to appear, Xilinx Application Note XAPP777.

[29] J. Ditmar, K. Torkelsson, A. Jantsch, “A Dynamically Reconfigurable {FPGA}-Based Content Addressable Memory for Internet Protocol Characterization”, 10th International Conference on Field Programmable Logic and Applications, Villach, Austria, September 2000.

[30] CloudShield Technologies, Inc., http://www.cloudshield.com. 2004.

[31] Novilit, Inc., http://www.novilit.com, 2004.

[32] Teja Technologies, Inc., http://www.teja.com, 2004.

[33] W. Huang, N. Saxena, E. J. McCluskey, "A Reliable LZ Data Compressor on Reconfigurable Coprocessors", The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. Napa Valley, CA, April 2001.

[34] P. Bellows, J. Flidr, L. Gharai, C. Perkins, P. Chodowiec, K. Gaj, "IPsec-Protected Transport of HDTV over IP", 13th International Conference on Field Programmable Logic, Lisbon, Portugal, 2003.

[35] N. Damianou, N. Dulay, E. Lupu, M Sloman, "The Ponder Specification Language", Workshop on Policies for Distributed Systems and Networks (Policy2001), HP Labs Bristol, Jan 2001.

[36] T.K.Lee, S. Yusuf, W. Luk, M. Sloman, E. Lupu, N. Fulay, "Compiling Policy Descriptions into Reconfigurable Firewall Processors", Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, CA, 2003.

[37] S. Bradner, "RFC 1242 - Benchmarking terminology for network interconnection devices", 1991.

[38] United Online, Inc. http://www.unitedonline.net, 2004.

[39] America Online, Inc. http://www.aol.com, 2004.

[40] J. Liang, R. Tessier, and D. Goeckel**, "**A Dynamically-Reconfigurable, Power-Efficient Turbo Decoder", in the Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*,* Napa, California, April 2004

[41] Xilinx, Inc. "3GPP Turbo Decoder Data Sheer", 2001.

[42] BL Hutchings, R. Franklin, and D. Carver. "Assisting Network Intrusion Detection with Reconfigurable Hardware", in the Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*,* Napa, California, 2002.

[43] Z.K. Baker, V.K. Prasanna, "A Methodology for the Synthesis of Efficient Intrusion Detection Systems on FPGAs", in the Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*,* Napa, California, 2004.

[44] J. Moscola, J. Lockwood, R. Loui, M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall", in the Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*,* Napa, California, 2003.

[45] M. Roesch, "Snort Users Manual", 2004.

[46] E. Koehler, R. Morris, B. Chen, "Programming Language Optimizations for Modular Router Configurations", Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, 2002.

[47] N. Shah, W. Plishker, K. Keutzer, "NP-Click: A Programming Model for the Intel IXP 1200", 2nd Workshop on Network Processors, Anaheim, CA, 2003.

[48] Xilinx, Inc. "ML300 User Guide (v1.3.1)", 2004.