2010

A low-cost, connection aware, load-balancing solution for distributing Gigabit Ethernet traffic between two intrusion detection systems

Adam Christopher Jackson
Iowa State University

A low-cost, connection aware, load-balancing solution for distributing Gigabit Ethernet traffic between two intrusion detection systems

by

Adam Christopher Jackson

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Co-majors: Computer Engineering; Information Assurance

Program of Study Committee:
Doug Jacobson, Major Professor
Phillip H. Jones III
Yong Guan

Iowa State University

Ames, Iowa

2010

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I want to take this opportunity to thank the many people that helped me both with this project and throughout my pursuit of a degree. First, thank you to all of my committee members for their work to educate students and their willingness to serve on my committee. I specifically want to thank Dr. Doug Jacobson for providing advice and direction throughout this project and over the course of my graduate studies. Additionally, Dr. Phillip H. Jones III helped greatly in discussing implementation approaches and debugging implementation issues. I also want to acknowledge the many professors over the course of my graduate studies who challenged me and taught me much about the fields of Information Assurance and Computer Engineering. Finally, I thank my family and friends for their continued support and constant willingness to listen when I simply needed to discuss the challenges I encountered.

ABSTRACT

In today's world of computer networking, Gigabit Ethernet is quickly becoming the norm for connectivity in computer networks. The ease of access to information on these networks leads to new information being made available daily. Rises in both malicious users and malicious network traffic increase the need for intrusion detection systems to monitor network traffic. However, intrusion detection systems capable of processing network traffic at the rate necessary for Gigabit Ethernet are typically expensive. An alternative to purchasing one of these systems is to use multiple, cheaper intrusion detection systems and run them in parallel. This requires that traffic be distributed to these intrusion detection systems such that their traffic monitoring activity is unaffected. For typical intrusion detection systems this means that all traffic belonging to a single connection cannot be separated. This thesis presents the design and implementation of a low-cost, connection aware, load balancing solution capable of distributing traffic to two intrusion detection systems while ensuring that all traffic for a given connection is not separated.

CHAPTER 1. INTRODUCTION

In today's world, Gigabit Ethernet is rapidly becoming the norm for network connectivity. As Gigabit Ethernet increases the speed of network connectivity, the amount of information available for access over the Internet also grows. This increase in data availability and the ease with which new data may be made available via the Internet creates a definite need for systems that secure this data. Recent breaches in computer and network security at high profile corporations have brought increased attention to information security. One powerful tool for preventing such network attacks is an intrusion detection system. At a high level, intrusion detection systems are responsible for monitoring network traffic and raising alerts upon identification of malicious traffic indicating a possible intrusion attempt. These systems typically rely on a set of rules or search patterns to identify possible intrusion attempts. Since the protection needs of the networked systems secured by intrusion detection systems are constantly changing, new rules must be added to intrusion detection system rule sets as new system vulnerabilities are identified. Network attack becomes more sophisticated as attackers learn and develop new methods for exploiting vulnerable systems. These more complex attacks drive the need for frequent additions to intrusion detection system rule sets. As rule sets grow, the amount of time required to apply all of these rule checks to network traffic increases. However, each new generation of network connectivity solutions allows traffic to flow faster. Many intrusion detection systems simply cannot properly process network traffic at the speed offered by Gigabit Ethernet networks. As a result, there exists a need for novel solutions to alleviate the monitoring load on a single intrusion detection system.

This problem can be solved by replacing a single intrusion detection with multiple intrusion detection systems operating in parallel. This solution reveals another problem because

network traffic must now be distributed between the multiple intrusion detection systems in a way that still allows them to properly monitor the traffic. For intrusion detection systems to properly identify possible intrusion attempts they typically need to monitor all traffic between two communicating systems. (For simplicity, the remainder of this thesis refers to this two-way communication as a connection.) As a result, the system distributing traffic between multiple intrusion detection systems must be connection aware to prevent dividing the traffic for a single connection between multiple intrusion detection systems. Current solutions on the market offering similar functionality cost as much as $20,000 [1] [2], and are thus prohibitively expensive for many businesses. With this in mind, this thesis presents the design and implementation of a low-cost, connection aware, load balancing solution capable of distributing traffic to two intrusion detection systems while ensuring that all traffic for a given connection is not separated.

The remainder of this thesis is organized as follows. Chapter 2 provides some background information related to the design presented in this thesis. Chapter 3 presents the design and implementation of this solution. Chapter 4 discusses testing and results. Chapter 5 identifies future work related to this project, and Chapter 6 offers conclusions of this project.

CHAPTER 2. BACKGROUND

This chapter provides background information related to this thesis. First, Section 2.1 describes some related work in the areas of Gigabit Ethernet network traffic processing and load balancing. Then, Section 2.2 provides an introduction to the NetFPGA platform used to implement the design presented in this thesis.

2.1 Related Work

This section discusses related solutions in the area of intrusion detection systems in high speed networks and intelligent traffic distribution and monitoring systems. Research into related market solutions revealed three vendors that offer products providing similar functionality to the design in this thesis. First, IP Fabrics' DeepSweep is discussed in Section 2.1.1. Then, Section 2.1.2 briefly examines Top Layer Security's DCFD 3500. Finally, CoyotePoint's Equalizer line of products is discussed in Section 2.1.3.

2.1.1 IP Fabrics' DeepSweep-1

IP Fabrics' DeepSweep-1 is a well-featured network surveillance solution designed for law enforcement surveillance applications. This product operates on 1 Gigabit Ethernet (1000-baseT connections), and supports many protocols running on IP. It operates as a standalone network connected system, and includes a web interface for controlling surveillance tasks [3].

Overall, this product differs from the design presented in this thesis in that it offers far more functionality than is necessary for the application of this thesis. Its surveillance capabilities also create potential wiretapping issues for non-law enforcement applications. Additionally, all functionality is implemented on the device itself rather than distributing functionality to

backend systems. As a result, this product does not qualify as a relevant solution to the problem addressed by this thesis.

2.1.2 Top Layer Security's DCFD 3500

Top Layer Security markets its DCFD 3500 as a data collection and filtering device intended for law enforcement surveillance applications. The device operates non-intrusively on Gigabit Ethernet (1000 base-T) connections extracting traffic based on identifying information found in layers up to layer 7. As with the IP Fabrics solution, a web administration interface is provided for configuring the device [4].

This solution does not address the problem identified by this thesis for a few reasons. First, its surveillance and collection nature raises similar wiretapping issues to the IP Fabrics solution. As with that solution, this product offers more functionality than the simple connection aware, load-balancer needed to address the problem. Current pricing information for this product available online is generally unreliable, but [1] and [2] suggest prices of at least $20,000, approximately 8 times the cost of the solution presented in this thesis.

2.1.3 Coyote Point's Equalizer Series

Coyote Point's Equalizer series offers four load balancing solutions that provide similar functionality to the design in this thesis. The Equalizer product line is comprised of the E250GX, E350GX, E450GX, and E650GX. All four products offer load balancing at layers 4 and above, and offer support for more protocols than the TCP/IP configuration supported by the design in this thesis. They also all offer a web interface for configuring and managing the device. The E250GX supports 2 Gigabit Ethernet connections, while the E350GX and E450GX each support 12, and the E650GX supports 20 1000 base-T connections and 2 fiber based GigE SFP ports. The E450GX and E650GX are the only solutions of those examined for this thesis that indicate hardware acceleration of some system functionality [5].

Although these solutions offer functionality closer to that of the design in this thesis, they still do not adequately address the problem. The E250GX is marketed as a low-cost solution,

and is listed at a price of $1,995 [5], but its throughput (650 Mbps) does not meet Gigabit per second line rates. The E350GX's throughput (850 Mbps) is also lower than 1 Gigabit per second, and is thus not a reasonable solution. The other products in this line are certain to be higher priced than this solution, so they do not qualify as low-cost solutions. Also, since these are load balancing solutions, they likely modify network traffic changing destination addressing information. These modifications will prohibit proper functioning of the intrusion detection systems to which the solution needs to deliver traffic.

2.2 NetFPGA

This section discusses relevant details of NetFPGA, the platform used to implement the solution presented in this thesis. First, Section 2.2.1 provides a brief introduction to the NetFPGA platform. Section 2.2.2 explains the specific NetFPGA system used for the implementation. Finally, Section 2.2.3 explains the NetFPGA Reference Pipeline extended for the implementation in this thesis.

2.2.1 NetFPGA Overview

The design presented in this thesis was implemented using a NetFPGA card. NetFPGA is a PCI card with four Gigabit Ethernet ports, an FPGA, and three types of memory that allows for implementing network systems such as switches, routers, or simple network interface cards. The implementation presented in this thesis uses a NetFPGA Version 2.1 PCI card. This card contains a Xilinx Virtex-II Pro 50 FPGA, and has 4.5 MB of SRAM and 64 MB of DDR2 DRAM available on the PCI card [6]. Figure 2.1 on page 6 found in [7] provides a graphical representation of how a NetFPGA card is structured. The implementation of the design presented in this thesis required only the configuration of the FPGA for interaction with the Gigabit Ethernet MACs and associated FIFOs, so a discussion of the memory system present on the NetFPGA is not necessary.

Overall, the cost of a deploying a NetFPGA system functionally equivalent to the one used for this thesis fits well within the category of low-cost solutions. The NetFPGA card is

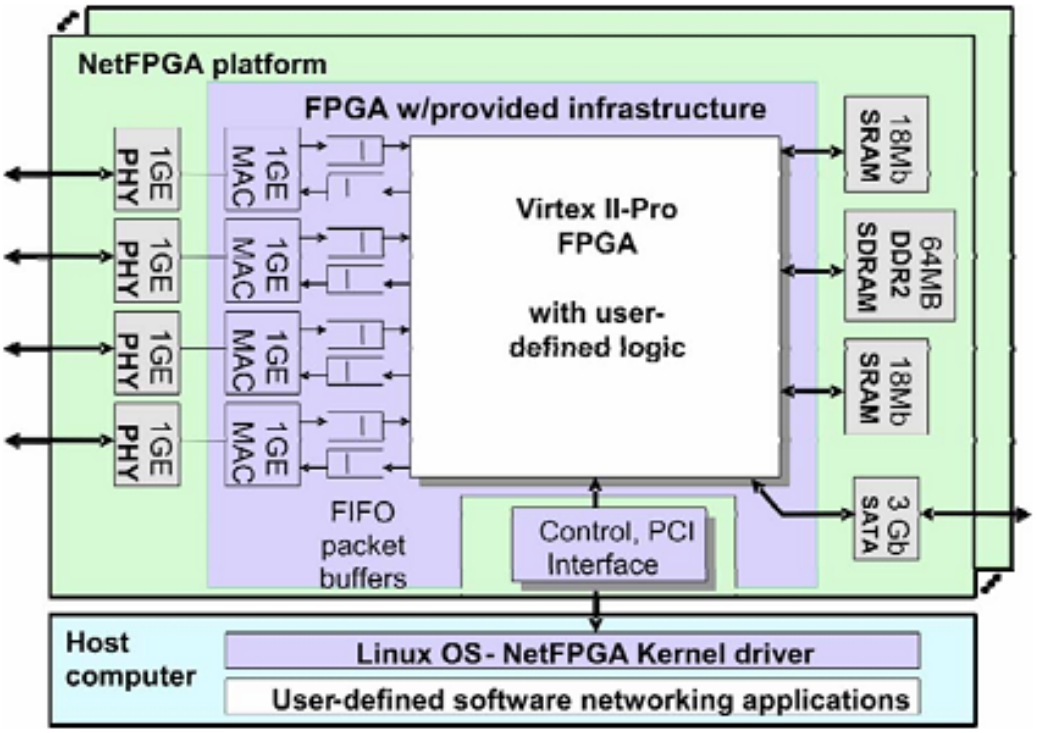

Figure 2.1 System block diagram of the NetFPGA platform

available for $1,199 ($499 for academia) according to [8], so it fits into the category of low-cost solutions. Accent Technology produces two types of pre-built systems with NetFPGA cards: the workstation class NetFPGA Cube and a rack-mountable NetFPGA 1U design. These systems are available in dual-core and quad-core designs with the quad-core NetFPGA 1U listed as the most expensive system at $1,849 [9]. This is the academic pricing, so the price is likely at least $2,549 to cover the additional cost of the NetFPGA for industry applications.

2.2.2 Development System

Development and testing of this design was performed on a quad-core NetFPGA Cube system as distributed by Accent Technology Inc. NetFPGA Package 1.2 was used for the NetFPGA development environment. This environment includes an integrated Makefile environment that is capable of many development tasks ranging from running simulations to generating a bitfile to configure the hardware. The build environment used by these Makefiles

relies on the Xilinx ISE 9.2i design tool suite. All of the hardware modules of this design are written in the Verilog hardware description language, and simulation test cases are written in Perl.

2.2.3 NetFPGA Reference Design

The design presented in this thesis is implemented as an extension of the NetFPGA Reference Pipeline described in [10]. The datapath for packet traffic is shown in Figure 2.2 taken from [10]. In this design, the Input Arbiter module determines the order in which to process packets from each of the different input ports, reads packets from the eight input queues (4 Ethernet MAC inputs, 4 CPU direct memory access (DMA) ports), and outputs a stream of packets to the Output Port Lookup module. The Output Port Lookup module processes the packet determining the output Ethernet MAC ports on which the packet should be sent, and passes this information along with the packet to the Output Queues module. The Output Queues module accesses the output port information to determine the Ethernet ports on which to output the packet, and adds the packet to the queues for these ports.

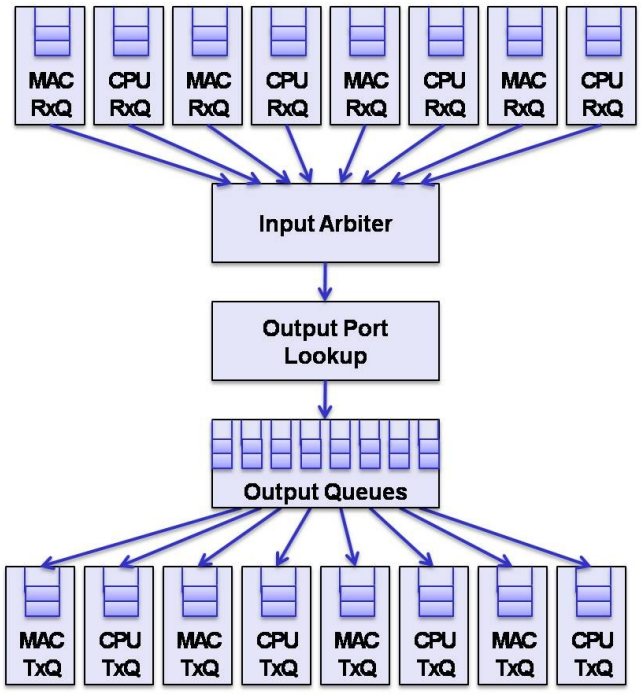

Figure 2.2 NetFPGA reference pipeline

Table 2.1 shows the internal datapath interface between modules from the perspective of the receiving module. Packet data and control information are sent via the `in_data` and `in_ctrl` buses respectively when the `in_wr` signal is asserted. It is the responsibility of the sending module to send only when the `in_rdy` signal is asserted, and new data and control information are sent during each cycle that it is asserted.

Table 2.1 NetFPGA Reference Design datapath signals

Signal Name	**Direction**	**Description**
in_data [DATA_WIDTH:0]	Input	**Input Data**: Used by writing module to pass the 64-bit headers and data packets
in_ctrl [CTRL_WIDTH:0]	Input	**Input Control**: Used by writing module to provide control information about the value on the `in_data` bus
in_wr	Input	**Input Write**: Used by writing module to indicate that the data on the `in_data` and `in_ctrl` buses is valid
in_rdy	Output	**Input Ready**: Used by reading module to indicate that it is ready to receive new data

At this time, a discussion of how packet data is handled internally in the Reference Pipeline is necessary. In the reference pipeline, and thus this thesis' design, each Ethernet packet is split into 64-bit packets and transported throughout the system on the `in_data` bus. These packets are processed by each subsequent module in the datapath before being output as a full Ethernet packet at the Ethernet MAC output ports. In addition to dividing the output into 64-bit packets internally, the datapath also supports using internal headers created by different datapath modules. These headers are identified by non-zero values on the `in_ctrl` control bus. The Input Arbiter appends the first header with an associated value of `0xff` on the `in_ctrl` bus. This header specifies the source port of this packet, the size in bytes and words of the Ethernet packet, and provides a field to later specify the output ports for this packet. Subsequent headers are added to the packet between this header and the actual data packets which are identified by a value of `0x00` on the `in_ctrl` bus. The control value for the last internal packet of the Ethernet packet is set to the location of the last valid byte in the internal packet.

CHAPTER 3. DESIGN & IMPLEMENTATION

This chapter presents the details of the design and implementation of the solution developed for this thesis. Section 3.1 explains the overall architecture of the design, and how it solves the problem of distributing network traffic with stream-awareness. Subsequently, Section 3.2 provides details of the implementation of this design.

3.1 Design

This section explains the design architecture used to distribute network traffic between two monitoring systems in this solution. At a high level, this solution acts as a duplicating Ethernet tap. It is designed to be located at the boundary between a secured internal network, and an unsecured external network - typically the Internet. Packets traveling between the networks are examined to determine the monitoring system to which they will be forwarded. This design uses a lookup table to determine the monitoring system output for each packet. Each time a packet enters the system, its TCP connection is identified based on IP address and TCP port. The active connection table is then queried to determine if the connection is already associated with one of the two monitoring systems. If the connection is found in the table, the packet is forwarded to the corresponding monitoring system's output. If this connection is not found, it is added to the table. New connections are assigned to the monitoring system to which a connection was least recently assigned. Connections remain in the table until the communication on this connection is complete. For the purposes of this design, this occurs when the design encounters a packet with the TCP FIN control flag set.

3.2 Implementation

This section details the implementation of the design described in Section 3.1. First, Section 3.2.1 offers an overview of the implementation and some considerations and their effect on the final solution. Section 3.2.2 describes the components of this system in detail. Finally, Section 3.2.3 discusses some simple future implementation steps.

3.2.1 Overview

This section provides an overview of how the design presented in Section 3.1 was implemented using the NetFPGA platform. The implementation presented here is created by replacing the Output Port Lookup module from the NetFPGA Reference Pipeline with a module that performs the expected stream-aware traffic distribution. The remainder of this section discusses some implementation challenges, their solutions, and their effect on the overall design.

The first implementation challenge was determining exactly what constitutes a connection for the purposes of distributing traffic based on streams. Current intrusion detection systems typically need to analyze all traffic for a given TCP connection. These TCP connections are established between TCP ports on two systems starting with a TCP handshake and concluding with a packet with the FIN flag set in the TCP header. Both systems in a TCP connection have specific IP addresses and TCP ports on which the connection occurs. Thus, for the purposes of this design a connection is defined to be all packets sent between two IP address-TCP port pairs until a packet with the TCP FIN flag set is encountered. With this definition in mind, there are 2^{96} possible connections because each IP address is 32 bits and each TCP port is 16 bits. Maintaining a table with 96-bit wide addresses proved impossible using the NetFPGA platform. (For details on why this is impossible please refer to the discussion of the Connection CAM LUT module found in Section 3.2.2.3.) With a little knowledge of the environment in which this traffic distribution system is likely to be placed, the number of bits needed to identify a connection may be reduced. Intrusion detection systems are typically located at the boundary of a local area network. In other words, traffic that the intrusion detection system monitors is either sent from or destined to a system on the Internet. IP addresses on the

public Internet are defined to be globally unique, and it is impossible to open two connections on the same TCP port of a single system. As a result, the IP address-TCP port pair for the connecting system on the Internet is sufficient to identify a connection. This means that only the IP address and TCP port of the Internet side of a connection must be stored to uniquely identify a connection for this design. As a result, this design uses the 48 bits of this IP address and TCP port as a connection identification tag.

Another implementation issue to consider was how many active TCP connections to handle simultaneously. In order to forward traffic by stream, the system must maintain a table of connection identification tags and the monitoring system used for all active connections. The design presented here is capable of handling 32 simultaneous connections. The FPGA used in this design is large enough to implement a larger connection table, but timing requirements prevented using larger tables. A more detailed discussion of this design compromise is included in the details of the Connection CAM LUT module in Section 3.2.2.3.

The next implementation issue was creating the connection table. This implementation actually uses two Content Addressable Memory (CAM) modules, each associated with one of the connections to a monitoring system. The CAM associated with each monitoring system stores connection identification tags for all active connections forwarded to that system. Consequently, the connection identification tag for all incoming packets is looked up in both tables, but written to only one when the system encounters a new connection.

Figure 3.1 on page 12 shows how to connect this implementation to a network. The design's intended placement is at the boundary of a local network such that all traffic from the Internet passes through the design to the internal network. With this in mind, Port 1 is connected to the external wide area network, and Port 2 is connected to the internal local area network. Ports 3 and 4 are connected to the monitoring systems or intrusion detection systems.

3.2.2 Details

This section provides details of exactly what functionality is present in each component of the implementation. Since this implementation only alters the Output Port Lookup of the

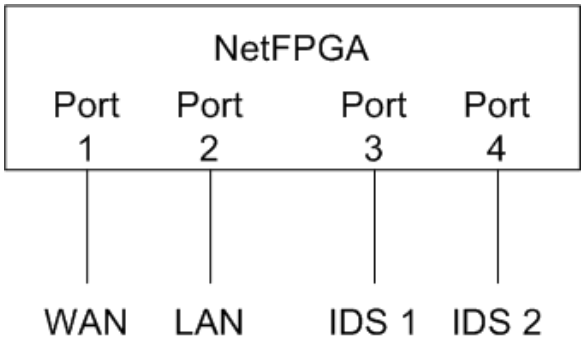

Figure 3.1 Port connections to the NetFPGA

Reference Pipeline, only it and its sub-modules are discussed here. For more information on the functionality of the Input Arbiter and Output Queues modules used as part of the NetFPGA Reference Pipeline refer to [10]. This section is organized according to individual components of the design. For each component, the details of its functionality are discussed, the input/output interface is defined, and any finite state machines used to control the module are explained. First, Section 3.2.2.1 describes the top level Output Port Lookup module. Section 3.2.2.2 explains the Header Parser module. Finally, Section 3.2.2.3 details the Connection CAM LUT module.

3.2.2.1 Output Port Lookup Module

The Output Port Lookup module fills a supervisory role in the process of determining the output ports for each packet. It also receives packets from the Input Arbiter and sends them the packet onto the Output Queues module after processing completes. Figure 3.2 on page 13 shows the hierarchy of the components within this module. The Header Parser module parses the Ethernet, IP, and TCP headers, and the Connection CAM LUT uses the parsed header information to determine on which ports to output this packet. The two FIFOs in the design allow for buffering packet data while the Connection CAM LUT module determines the output ports.

Although this module is primarily a supervisory module, it also handles receiving and sending packet data. As each packet enters this module, its `in_data` and `in_ctrl` values

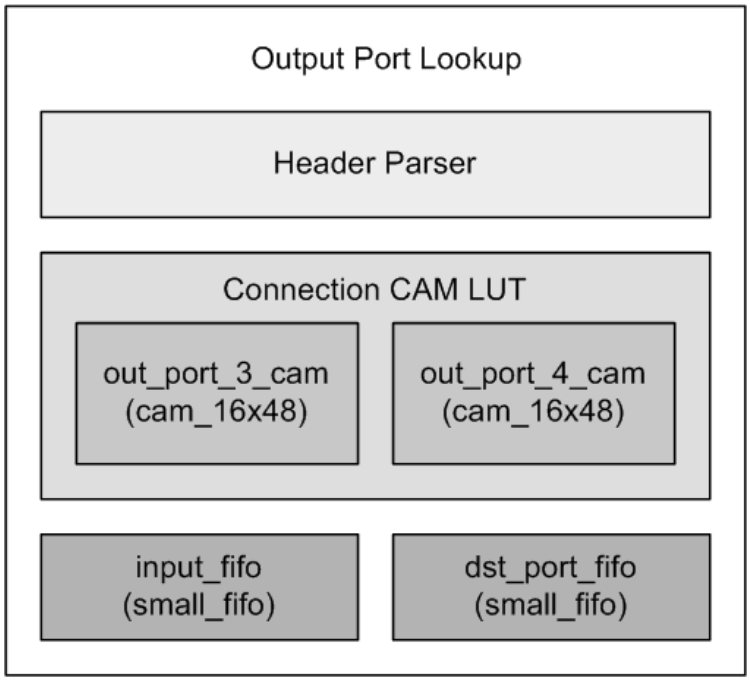

Figure 3.2 Sub-components of the Output Port Lookup module

are buffered in the `input_fifo` until its output ports are determined. Once the Connection CAM LUT module determines the output ports for this packet, the ports are written to the `dst_port_fifo`. As soon as this FIFO indicates it is not empty, the NetFPGA header packet is read from the `input_fifo`, the output ports are read from the `dst_port_fifo` and added to it, and finally it is output from the module. Subsequently, all data packets (the 64-bit internal packets that represent the actual Ethernet packet) are output from this module.

Figure 3.3 on page 14 shows the system diagram for the Output Port Lookup module. As stated before, this module is primarily responsible for configuring the connections between its submodules. However, it also forwards all register requests onto the next module, and is responsible for adding the specified output ports to the packet output. The finite state machine control logic handles the process of modifying packets with output port information before outputting them. Aside from these steps, all major packet processing functionality is performed by the Header Parser and Connection CAM LUT modules.

Table 3.1 on page 15 describes the interface for the Output Port Lookup module. Signals prefixed `in_` and `out_` support the input and output datapaths respectively. The `data` and

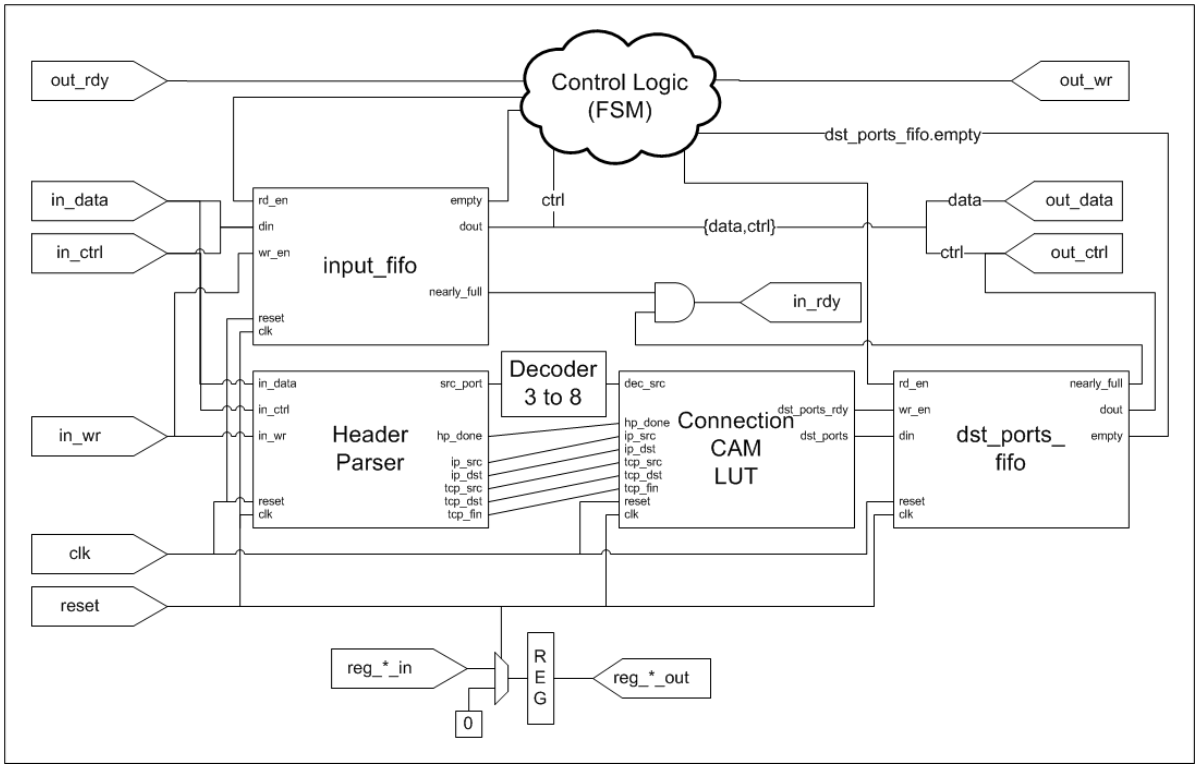

Figure 3.3 System diagram of the Output Port Lookup module

`ctrl` buses provide the raw packet data and control information related to each packet. The `wr` signals indicate when data is valid on the `data` and `ctrl` buses, and `rdy` signals communicate that a receiver is ready to receive input. All signals prefixed `reg_` comprise the register interface with input and output postfixed `_in` and `_out` respectively. The design detailed in this thesis does not provide any registers, so register signals are passed from input to output each clock cycle.

Figure 3.4 on page 16 shows the finite state machine that controls the Output Port Lookup module. The Reference Pipeline based `learning_cam_switch` design uses the same state machine, so only a brief discussion of it is necessary here. After a reset, the state machine starts in the `WAIT_TILL_DONE_DECODE` state and waits in this state until the output ports for the next packet in the `input_fifo` are determined. Since these ports are written to the `dst_ports_fifo` the state machine waits until this FIFO indicates that it is no longer empty. The state machine then transitions to the `WRITE_HDR` state in which the internal NetFPGA header for this

Table 3.1 Interface description for the Output Port Lookup module

Signal Name	Direction	Description
in_data [DATA_WIDTH-1:0]	input	**Input Data**: Input internal header and data packets
in_ctrl [CTRL_WIDTH-1:0]	input	**Input Control**: Control information associated with each packet on the `in_data`: bus
in_wr	input	**Input Write**: Asserted when `in_data`: and `in_ctrl`: buses are valid
in_rdy	output	**Input Ready**: Asserted when this module is ready to receive input data
out_data [DATA_WIDTH-1:0]	input	**Output Data**: Output internal header and data packets
out_ctrl [CTRL_WIDTH-1:0]	input	**Output Control**: Control information associated with each packet on the `out_data`: bus
out_wr	input	**Output Write**: Asserted when `out_data`: and `out_ctrl`: buses are valid
out_rdy	output	**Output Ready**: Asserted when this module is ready to receive input data
reg_req_in	input	**Register Request In**: Indicates a register request
reg_ack_in	input	**Register Ack In**: Acknowledges a register request
reg_rd_wr_L_in	input	**Register Read/Write In**: High to indicate a read request, low to indicate a write request
reg_addr_in [REG_ADDR_WIDTH-1:0]	input	**Register Address In**: Address of the requested register
reg_data_in [REG_DATA_WIDTH-1:0]	input	**Register Data In**: Data to write to the requested register
reg_src_in [REG_SRC_WIDTH-1:0]	input	**Register Source In**: Source identification of the requestor
reg_req_out	output	**Register Request Out**: Indicates a register request
reg_ack_out	output	**Register Ack Out**: Acknowledges a register request
reg_rd_wr_L_out	output	**Register Read/Write Out**: High to indicate a read request, low to indicate a write request
reg_addr_out [REG_ADDR_WIDTH-1:0]	output	**Register Address Out**: Address of the requested register
reg_data_out [REG_DATA_WIDTH-1:0]	output	**Register Data Out**: Data read from the requested register
reg_src_out [REG_SRC_WIDTH-1:0]	output	**Register Source Out**: Source identification of the requestor
clk	input	**Clock**: Module clock
reset	input	**Reset**: Module reset

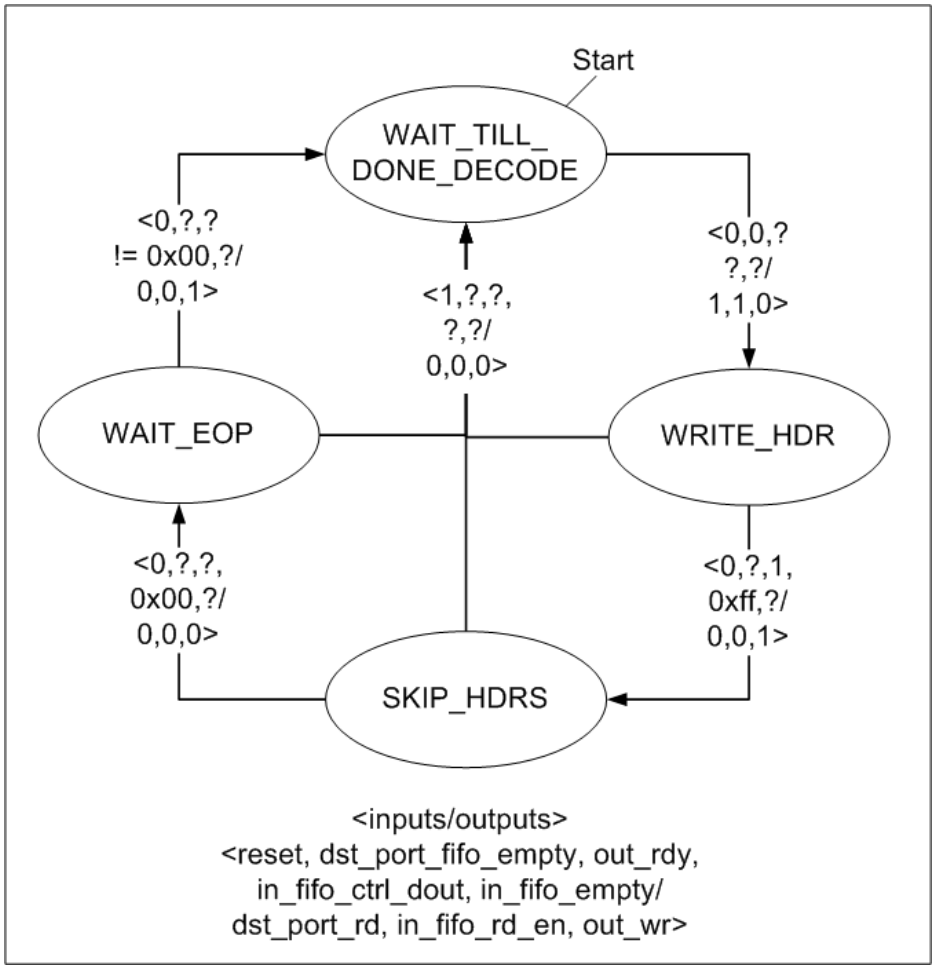

Figure 3.4 Finite State Machine for the Output Port Lookup module

Ethernet packet is read from the `input_fifo`. The destination ports are then added to this header and it is written to the output as the state transitions to `SKIP_HDRS`. This state simply outputs all of the other internal headers for this Ethernet packet by setting the `out_wr` wire to 1 and the `in_fifo_rd_en` wire to 1 to cause reading and output of new data each clock cycle. These headers are identified by non-zero values on the `in_fifo_ctrl_dout` bus - the buffered `in_ctrl` signal. Once the first zero value on this bus is observed, the state transitions to the `WAIT_EOP` state where buffered packets are output until the last packet is encountered. This packet is identified by another non-zero value on the `in_fifo_ctrl_dout` bus. After outputting the last packet of this Ethernet packet, the state returns to the `WAIT_TILL_DONE_DECODE` state and the process repeats.

3.2.2.2 Header Parser Module

This module parses and returns the values of necessary fields from Ethernet, IP, and TCP headers. At a high level, this module reads the first six 64-bit internal NetFPGA and stores the value of each field to a register connected to the outputs. Once a header field is stored into its register, it is not changed until it is overwritten by the data from the same field in the next packet.

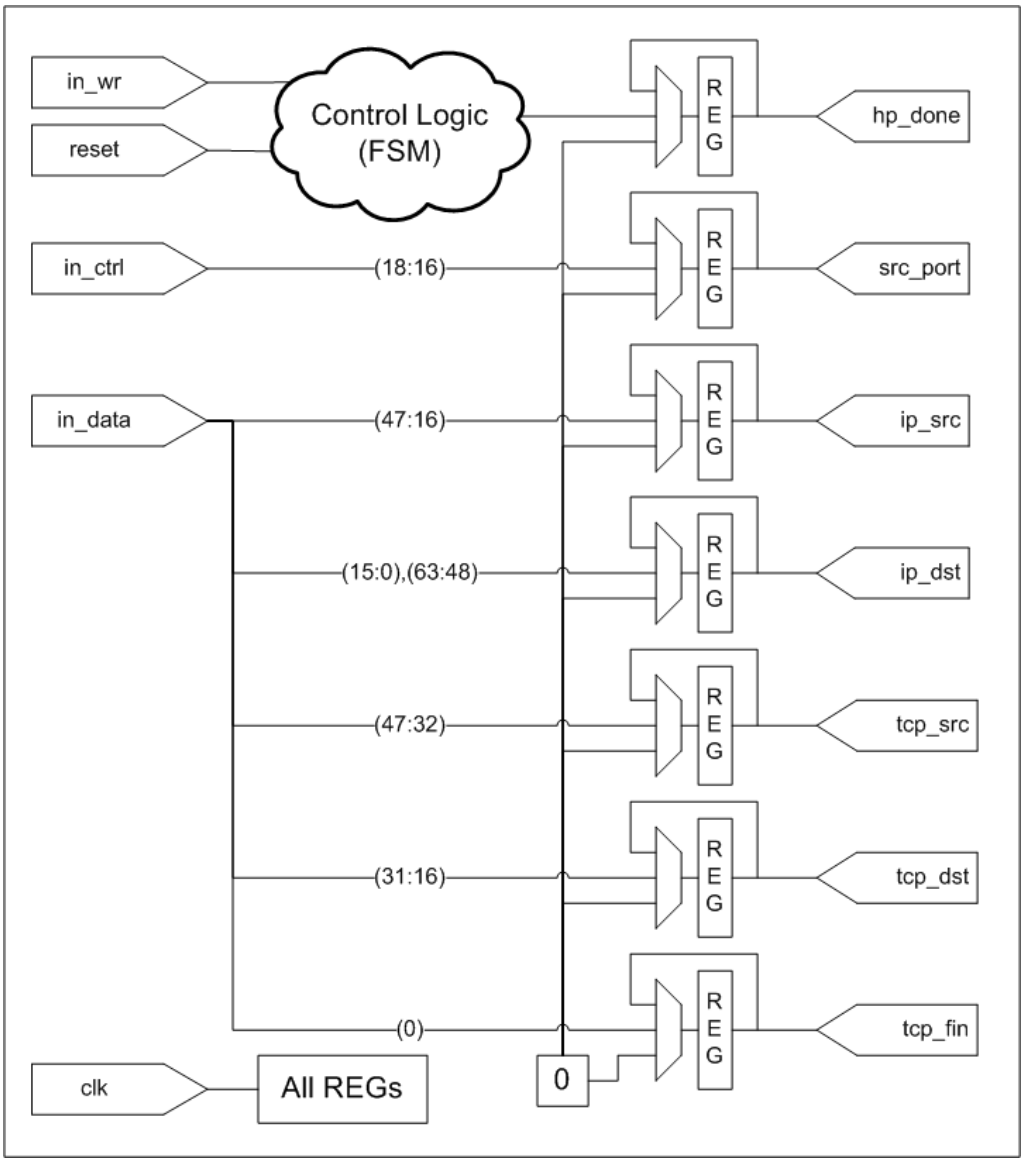

Figure 3.5 System diagram of the Header Parser module

Figure 3.5 depicts the overall system diagram for the Header Parser module. This module is effectively a set of registers with multiplexers controlling their next input value. The next value for all registers is the previous value except when a reset occurs or when the header field for the register is updated. Although some registers share input bits, these correspond to the same bits in different input packets. As a result, registers that share input bits will never register new data during the same packet. The finite state machine control logic for this module controls all of the multiplexers and loads the registers when the correct header fields are encountered in the packet stream.

Table 3.2 Interface description for the Header Parser module

Signal Name	Direction	Description
in_data [DATA_WIDTH-1:0]	input	**Input Data**: Input internal header and data packets
in_ctrl [CTRL_WIDTH-1:0]	input	**Input Control**: Control information associated with each packet on the `in_data`: bus
in_wr	input	**Input Write**: Asserted when data `in_data`: and `in_ctrl`: buses are valid
hp_done	output	**Header Parser Done**: Asserted when Ethernet, IP, and TCP header parsing is complete
src_port [INPUT_QUEUES-1:0]	output	**Internal Source Port**: The NetFPGA source port from which this packet originated
ip_src [31:0]	output	**IP Source Address**: Source address from the IP header
ip_dst [31:0]	output	**IP Destination Address**: Destination address from the IP header
tcp_src [15:0]	output	**TCP Source Address**: Source port from the TCP header
tcp_dst [15:0]	output	**TCP Destination Address**: Destination port from the TCP header
tcp_fin	output	**TCP FIN**: Asserted when the parsed packet had the TCP FIN flag set
clk	input	**Clock**: Module clock
reset	input	**Reset**: Module reset

Table 3.2 shows the input and output signals for this module. The signals prefixed with `in_` are the datapath input signals. The `hp_done` signal indicates when header parsing is complete, and is raised to 1 when parsing of the Ethernet, IP, and TCP headers completes. The `src_port`

signal is a one-hot encoded representation of the source port from which this packet came. For packets from the first Ethernet MAC port it has a binary value of "00000001," "00000100" for the second, "00010000" for the third, and "01000000" for the fourth. Output signals prefixed by `ip_` and `tcp_` are fields in the IP and TCP headers. The `clk` and `reset` signals are the system wide clock and reset signals. In this module, all Ethernet, IP, TCP header fields are parsed in this module, but most are not output in order to minimize the amount of wiring in this design. The source code is present to output all fields, but most of it is commented out because this design only needs the IP addresses and ports parsed. Although only one IP address-TCP port pair is used to identify a connection in this design, both source and destination pairs are output by this module. This occurs because the Connection CAM LUT module uses either the source or destination pair for the tag based on the input port from which this packet originated.

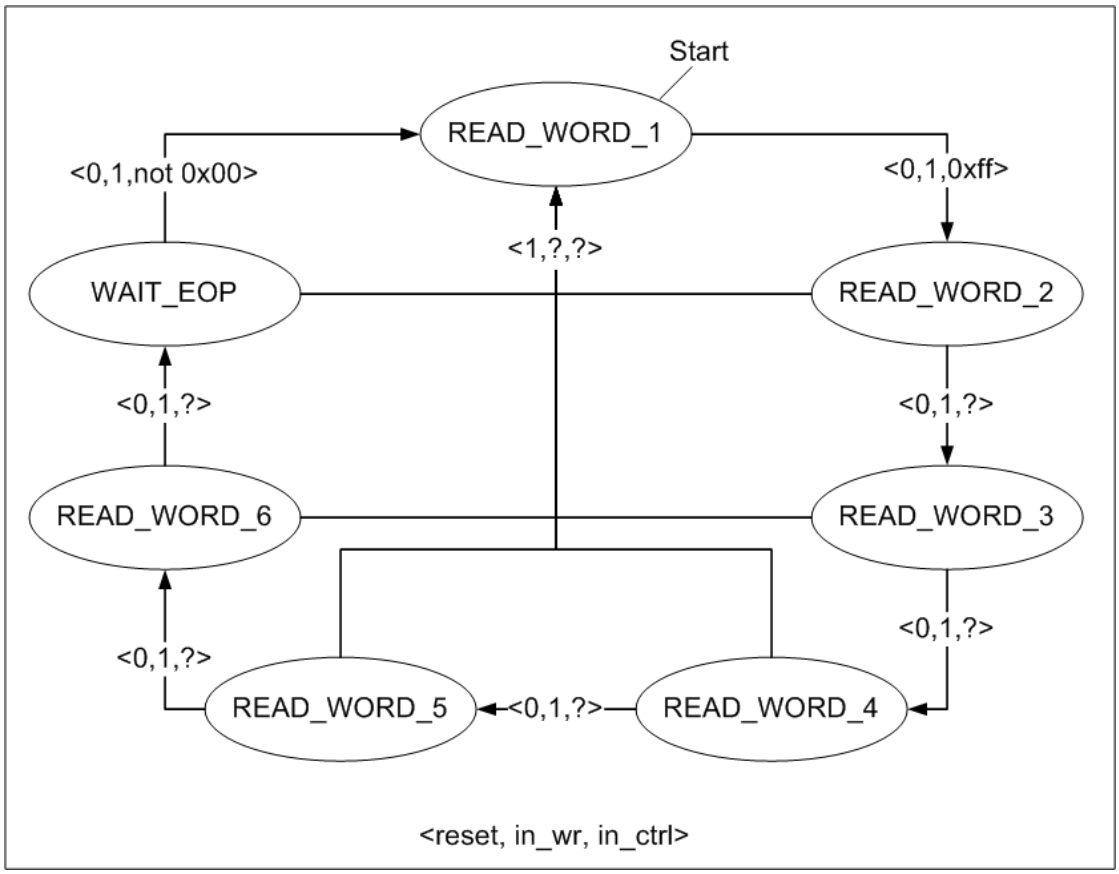

Figure 3.6 Finite State Machine for the Header Parser module

Figure 3.6 shows the finite state machine used to control this module. The READ_WORD_1 state reads data from the in_data bus when the in_wr wire is set to 1. In this state, the internal control packet is first read and the source port for this packet is decoded and stored in a register connected to the src_port output signal. After reading the internal control packet, the first data packet is also read while in this state. Since each internal packet is 64 bits long in this design, the first data packet contains only the Ethernet destination field and the upper half of the Ethernet source field. These values are stored to registers and remain unchanged until they are overwritten again by the next packet. After storing these values the state machine transitions to the READ_WORD_2 state. In this state, the next packet is read when the design receives a 1 on the in_wr wire. This packet contains the next 64 bits of the headers, so the lower half of the Ethernet source address and the Ethertype fields are stored to registers as with the previous state. This packet also contains the first three fields of the IP header: version, Internet header length, and the differentiated services. These values are also stored to registers for possible output. As stated before, these values are not output by this module but are still parsed. The READ_WORD_3, READ_WORD_4, READ_WORD_5, and READ_WORD_6 states parse the remaining fields in the IP and TCP headers similar to the READ_WORD_2 state. After READ_WORD_6 all of the headers are parsed, so the hp_done signal is set to 1 to indicate successful completion of the header parsing. At this point the state machine waits in the WAIT_EOP state until it receives a non-zero value on the in_ctrl bus indicating the end of the Ethernet packet. At this point, the state machine returns to the READ_WORD_1 state and idles until the start of the next packet.

3.2.2.3 Connection CAM LUT Module

This module manages the connection table and uses it to determine and return the destination ports for each packet. Each time a lookup is requested every packet inbound on port 1 is output on port 2, and every packet inbound on port 2 is output on port 1 to make this design act as a tap on the Ethernet connection. In addition, this module uses either the source IP address-TCP port pair (for packets input on wide area network facing port 1) or

the destination IP address-TCP port pair (for packets coming from local area network facing port 2). This module uses two Content Addressable Memory (CAM) modules to implement the connection table. This design relies on these CAMs for determining to which monitoring system a given stream should be output. The address-port pair is used as a tag to lookup in the CAMs. If it is found in either CAM, the output port corresponding to the CAM in which it was found is added to the output ports. If the connection is not found in either CAM it is added to the CAM written to least recently, and the corresponding port is added to the output ports. Connections are removed from the CAMs whenever the design encounters a TCP FIN packet. With this configuration, it is impossible for a packet to be output on more than two ports and at no time will a packet be output on both ports 1 and 2 or on both ports 3 and 4.

Figure 3.7 on page 22 shows the overall system diagram of the Connection CAM LUT module. For this diagram, all multiplexers are controlled by the finite state machine control logic unless another wire is directly connected as the select signal. There are two major aspects of this system: the CAM management logic and the output port control logic. The CAM management logic must configure the CAMs for reads to lookup an input connection and for writes when a new connection is added to the CAM or a closed connection is removed. The output port control logic determines the output port to use for each packet. All four Ethernet MAC ports are possible for a generic input packet, so this logic must determine from where the packet originated and on which monitoring port it should output. As a result, this logic must also communicate with the finite state machine control to determine which monitoring ports to use.

Table 3.3 on page 23 shows the input and output signals for the Connection CAM LUT module. The IP and TCP source and destination signals should be clear from the signal naming convention. The `tcp_fin` signal is set to 1 by the Header Parser module whenever it encounters a TCP packet with the FIN control set. The `lookup_req` input is set to high when the IP and TCP addresses are valid to begin a lookup. The `decoded_src` signal is a one-hot encoded bus indicating the input port from which a packet originated. The `dst_ports` output is a one-hot encoded bus indicating on which output port to send a packet and the `dst_ports_rdy` output

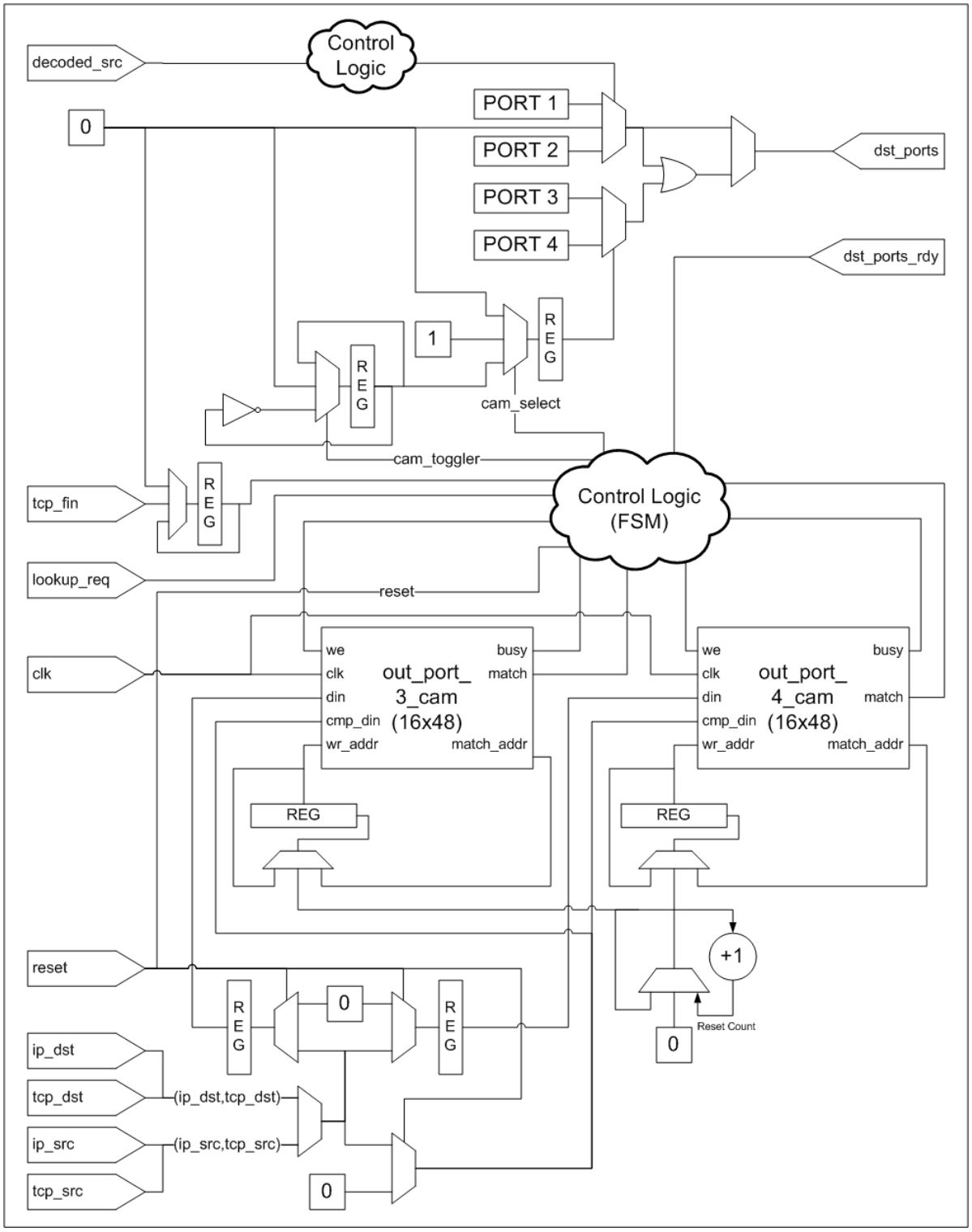

Figure 3.7 System diagram of the Connection CAM LUT module

Table 3.3 Interface description for the Connection CAM LUT module

Signal Name	Direction	Description
src_port [INPUT_QUEUES-1:0]	output	**Internal Source Port**: The NetFPGA source port from which this packet originated
ip_src [31:0]	output	**IP Source Address**: Source address from the IP header
ip_dst [31:0]	output	**IP Destination Address**: Destination address from the IP header
tcp_src [15:0]	output	**TCP Source Address**: Source port from the TCP header
tcp_dst [15:0]	output	**TCP Destination Address**: Destination port from the TCP header
tcp_fin	output	**TCP FIN**: Asserted when the packet being processed had the TCP FIN flag set
lookup_req	input	**Lookup Request**: Lookup process starts when this signal is asserted
decoded_src [i:0]	input	**Decoded Internal Source Port**: One-hot encoded bus indicating the source of this packet
dst_ports [OUTPUT_QUEUES-1:0]	output	**Internal Destination Port**: One-hot encoded bus indicating the destination of this packet
dst_ports_rdy	output	**Destination Port Ready**: Asserted when the `dst_ports` bus is valid
clk	input	**Clock**: Module clock
reset	input	**Reset**: Module reset

is set to 1 when its output is valid. The `clk` and `reset` signals are the system clock and reset signals.

Figure 3.8 on page 24 shows the finite state machine that controls the Connection CAM LUT module. Upon a system reset, the state machine starts in the `RESET` state. While in this state, each entry in both CAMs is overwritten with a value of 0 to indicate that this location is empty. Once all of these writes are completed (as indicated by values of 0 on the busy and write enable signals for both CAMs) the state machine transitions to the `IDLE` state. During this state, the compare input for the CAM is set to the proper IP address-TCP port pair for a CAM lookup. This lookup then occurs during the cycle that the state transitions to the `LOOKUP_DONE` state. As the state transitions to the `LOOKUP_DONE` state, the input IP addresses,

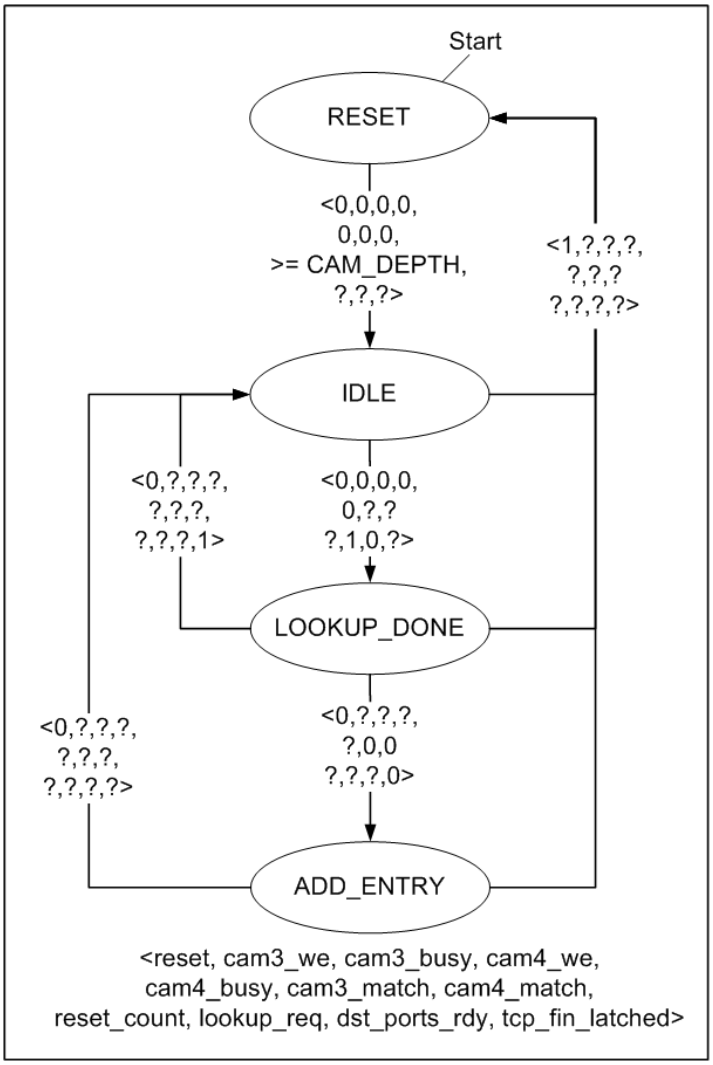

Figure 3.8 Finite State Machine for the Connection CAM LUT module

TCP ports, and the `tcp_fin` signal are stored to registers for the remainder of this packet's processing. In the `LOOKUP_DONE` state, the result of the CAM lookup is available, so the match signals for each CAM are checked to determine if either CAM contained the queried connection. If a match is found the design adds the port associated with the matching CAM to `dst_ports` and asserts the `dst_ports_rdy` signal. Also at this time, the design configures the matching CAM for a write of 0 to the match location to remove the entry from the CAM if the TCP FIN control for this packet was set. Regardless of whether the entry is removed from the CAM, the state machine returns to the `IDLE` state whenever one of the CAM lookups matched. If neither

CAM matches the connection lookup, there are two possible scenarios. First, if the looked up packet has the TCP FIN control set, then this is the last packet in the connection and there is no need to add an entry to the CAM for it. The output port to which a new stream was least recently assigned is added to the `dst_ports` bus, the `dst_ports_rdy` signal is asserted, and the state transitions to `IDLE`. However, if the TCP FIN control is not set, the design needs to add this connection to the CAM. At this point, a value of 0 is loaded onto the compare input bus for the CAM to find an empty location in the CAM, and the state transitions to the `ADD_ENTRY` state. After entering the `ADD_ENTRY` state, the design checks for a match in the CAM for the monitoring port that was least recently assigned a new connection. A match indicates an available entry in the CAM for this port (since the design looked for the empty value 0 during the transition into this state). When it detects a match, the design configures the CAM in question to write the IP address-TCP port tag for this new stream to the address of the unused entry. In addition to configuring this write, the output port associated with the CAM is added to `dst_ports` and the `dst_ports_rdy` signal is asserted. If no available entries are found in the CAM, the design asserts the `dst_ports_rdy` signal without adding either of the monitoring ports to the set of output ports. Thus, this packet is only forwarded between the internal and external networks. Regardless of whether the entry is added, the state machine transitions to the `IDLE` state on the next clock cycle to continue processing packets.

At this point, it is possible to return to the implementation consideration regarding the number of active connections to support as described in Section 3.2.1. At that time it was noted that this design only supports 32 active connections. This occurs because the design cannot handle more active streams than available entries in the CAMs. In the implementation used for this thesis, each CAM has 16 entries, so a maximum of 32 active streams may be monitored. Additional connections are effectively dropped for monitoring purposes, but are still forwarded between the internal and external networks. The number of active connections is limited by the fact that a CAM with more entries will not meet clock timing requirements for the FPGA used in this design. However, this limit is platform dependent, therefore using an FPGA newer than the Virtex-II Pro will likely allow for monitoring more active streams.

3.2.3 Future Steps

Since this is a proof of concept design, some implementation compromises were made and need to be discussed here. First, the Internet Header Length field of the IP header is ignored, and the design assumes all packets have a 20 byte IP header. For a production design, this value must be handled properly to ensure that the correct location for the TCP ports is identified. Since this field rarely indicates a header length other than 20 bytes, a possible alternative solution is to use only an IP address and a hard coded TCP port for the connection identification tag when this field does not indicate a 20 byte IP header.

In addition to the IP header issue, this design currently assumes all traffic is comprised of IP packets, but in practice this is not true. In the case of non-IP traffic the design needs only to forward the traffic between ports 1 and 2. It is possible to use the `tcp_fin` signal as a "Do Not Write" signal to prevent adding these packets to the connection tables.

Another issue relates to the possibility that the NetFPGA datapath is not 64 bits wide. The NetFPGA code base defines the datapath width using a parameter for each module. As a result it is possible for the data width to not be 64 bits. Some reference designs even process a 32-bit wide datapath, so this thesis' design needs to eventually handle different datapath widths. Currently, the Output Port Lookup module is capable of handling other datapath widths, but it needs to receive an output port for each packet from the Connection CAM LUT. The Connection CAM LUT module relies on the Header Parser to provide these output ports, but the Header Parser does not support datapath widths other than 64 bits. Thus the implementation as is does not support other datapath widths. Given the current design, the simplest solution is to configure the Header Parser module to provide default values for other datapath widths that would indicate to the Connection CAM LUT to only forward the packets.

CHAPTER 4. TESTING

This chapter explains the process of testing the implementation found in Section 3.2. It starts with a discussion of how NetFPGA designs are typically simulated in Section 4.1. Section 4.2 describes details of the simulations performed on this design. Finally, Section 4.3 explains the status of hardware testing for this design.

4.1 NetFPGA Verification Test Environment

Before discussing simulation testing of the implementation described in Section 3.2 it is necessary to explain the simulation infrastructure associated with the NetFPGA Package. All NetFPGA documentation refers to this simulation testing as verification testing, so the same convention is used here. At a high level, each verification test first generates a set of input packets and a set of output packets. Then a simulation of the system with the specified input packets is run. At the completion of the simulation, the actual output packets are compared to the expected output packets. Any differences in individual packets, the number of packets on each port, or the order of the output packets between the expected and actual packet output causes a verification test to fail and output simple debugging information.

Table 4.1 on page 28 shows the output of a failed verification test. This failure was created by switching the expected monitoring port output for a single packet. Notice that this change caused the number of packets observed on each output port to differ the expected number as indicated by the error messages for ports 3 and 4. The output for a passing test is shown in Table 4.2 on page 29. This is effectively the same output as for a failed test with the exclusion of the error indications.

Table 4.1 Example output of failing NetFPGA verification test

```
--- Simulation is complete. Validating the output.
        Comparing simulation output for port 1 ...
        Port 1 matches [5 packets]
        Comparing simulation output for port 2 ...
        Port 2 matches [7 packets]
        Comparing simulation output for port 3 ...
        ERROR: Number of packets mismatch: expected 5 but saw 6
        Port 3 saw 1 errors.

        Comparing simulation output for port 4 ...
        ERROR: Number of packets mismatch: expected 7 but saw 6
        Port 4 saw 1 errors.

        Comparing simulation output for DMA queue 1 ...
        DMA queue 1 matches [0 packets]
        Comparing simulation output for DMA queue 2 ...
        DMA queue 2 matches [0 packets]
        Comparing simulation output for DMA queue 3 ...
        DMA queue 3 matches [0 packets]
        Comparing simulation output for DMA queue 4 ...
        DMA queue 4 matches [0 packets]
--- Test failed (test_thesis_short) - expected and seen data differs.
Error: test test_thesis_short failed!
------------SUMMARY---------------
PASSING TESTS:
FAILING TESTS:
                test_thesis_short
TOTAL: 1 PASS: 0  FAIL: 1
```

Table 4.2 Example output of passing NetFPGA verification test

```
--- Simulation is complete. Validating the output.
        Comparing simulation output for port 1 ...
        Port 1 matches [5 packets]
        Comparing simulation output for port 2 ...
        Port 2 matches [7 packets]
        Comparing simulation output for port 3 ...
        Port 3 matches [6 packets]
        Comparing simulation output for port 4 ...
        Port 4 matches [6 packets]
        Comparing simulation output for DMA queue 1 ...
        DMA queue 1 matches [0 packets]
        Comparing simulation output for DMA queue 2 ...
        DMA queue 2 matches [0 packets]
        Comparing simulation output for DMA queue 3 ...
        DMA queue 3 matches [0 packets]
        Comparing simulation output for DMA queue 4 ...
        DMA queue 4 matches [0 packets]
--- Test PASSED (test_thesis_short)
Test test_thesis_short passed!
------------SUMMARY--------------
PASSING TESTS:
                test_thesis_short
FAILING TESTS:
TOTAL: 1 PASS: 1  FAIL: 0
```

The NetFPGA verification test environment relies on a simple directory structure for test configuration and three Perl scripts to test a design. The two files necessary to configure and run a simulation using the NetFPGA verification test environment reside inside the `verif/` directory within a NetFPGA project directory. Each verification test's directory uses the naming convention `test_majorName_minorName`. The `config.txt` file provides a short test description and specifies how long the simulation will run. The second file is the `make_pkts.pl` script explained below. The first of the Perl scripts used for simulations is `nf2_run_test.pl`. This script runs a simulation test for the current project using the system's simulator (ModelSim 6.3 SE for this thesis). The `--major` and `--minor` options specify which test(s) to run, where `--major` and `--minor` are the same as the `majorName` and `minorName` portions of the name of the verification test's directory. The second script, `nf2_compare.pl`, compares the expected and actual outputs after a simulation and indicates whether they match. The third script, `make_pkts.pl` is project specific and is responsible for specifying the packets to input to the simulation and the expected output packets. In addition to these main files, many Perl library files are included with the NetFPGA Package that allow for creating, sending, and expecting Ethernet and IP packets.

4.2 Verification Testing

This section provides details of exactly what was done to simulate the implementation for this thesis. Section 4.2.1 discusses the Perl library functions written to enable simulation of TCP packets and TCP streams. Section 4.2.2 describes a short test that verifies basic functionality of the design, and Section 4.2.4 describes a longer simulation that tests the system more fully.

4.2.1 Perl Library Additions

This section explains Perl packages and functions written to allow simulating TCP packets in the NetFPGA simulation environment. Each subsection describes either a package or a function written to enable testing of TCP traffic via the NetFPGA simulation environment.

The NetFPGA Package provides no support for creating a TCP packet, so this functionality was added with one Perl package and two functions.

4.2.1.1 The `TCP_hdr` Package

This package provides the ability to create a TCP header. It is modeled after the Ethernet and IP header creation packages provided with the NetFPGA Package. A constructor provides the ability to specify each field in the TCP header. The TCP headers created by this package are simply an array of bytes, and the package contains all the functionality necessary to read individual fields of the TCP header. Table 4.3 lists the functions available in this package and briefly describes their functionality.

Table 4.3 Functions available in the `TCP_hdr()` package

Function Name	**Description**
new	Constructor to create a new TCP_hdr. All values default to 0.
SrcPort	Get TCP source port
DstPort	Get TCP destination port
SeqNum	Get the Sequence Number
AckNum	Get the ACK Number
DataOffset	Get the value of the Data Offset field
ECN	Get the value of the ECN field
Control	Get the value of the Control field
Data	Get the packet's data
Window	Get the value of the Window field.
Checksum	Get the checksum
UrgentPointer	Get the value of the Urgent Pointer field
IP_hdr	Get the IP_hdr structure associated with this TCP_hdr
length_in_bytes	Get the header length in bytes (always returns 20)
calc_checksum	Calculate and set the checksum

4.2.1.2 The `make_TCP_pkt()` Function

This function belongs to the `SimTCP` package and creates a TCP packet that may be sent to the NetFPGA. This function creates the packet by first generating the Ethernet and IP headers using library packages provided with the NetFPGA Package. The TCP_hdr package previously described in this thesis generates the TCP header. These headers are concatenated

Table 4.4 Example `make_TCP_pkt()` function call

```
make_TCP_pkt(length, ETH_DA, ETH_SA, TTL, IP_DST, IP_SRC, TCP_DST,
    TCP_SRC, TCP_CTRL)
```

as a string of bytes and the remainder of the packet, up to the specified packet size, is filled with the hex value of each byte's position in the packet.

The `make_TCP_pkt()` function is called with the arguments shown in Table 4.4. Table 4.5 explains each argument. All arguments are required and the length parameter specifies the length of the packet including the Ethernet, IP, and TCP headers. As a result, a warning is displayed in the shell, and a packet with a length of 54 bytes (the minimum length of the 3 headers) is returned if a length shorter than 54 is provided. The `TCP_CTRL` parameter is placed directly into the TCP Control section of the header providing direct access to `SYN`, `ACK`, `FIN`, and other flags.

Table 4.5 Arguments for the `make_TCP_pkt()` function

Argument Name	**Description**
length	Length of packet to create
ETH_DST	Ethernet Destination MAC Address
ETH_SRC	Ethernet Source MAC Address
TTL	TTL value for this packet
IP_DST	IP Destination Address
SRC_IP	IP Source Address
TCP_DST	TCP Destination Port
TCP_SRC	TCP Source Port
TCP_SEQ_NUM	TCP Sequence Number of this packet
TCP_CTRL	TCP Control value for this packet. Note that the following bit locations are the reverse of what is typically documented for TCP headers. Bit 0 is FIN, bit 1 is SYN, bit 2 is RST, bit 3 is PSH, bit 4 is ACK, and bit 5 is URG. For example, a value of 0x01 sets the FIN flag and a value of 0x13 set FIN, SYN, and URG.

Table 4.6 Example `make_TCP_stream()` function call

```
make_TCP_stream(stream_length, pkt_length, ETH_DST, ETH_SRC,
    TTL, IP_DST, SRC_IP, TCP_DST, TCP_SRC)
```

4.2.1.3 The `make_TCP_stream()` Function

This function creates a unidirectional stream of TCP packets, and is also part of the `SimTCP` package. Before discussing the details of this function, it is important to note that this function does not create a true TCP stream as the initial three-way TCP handshake is not performed since traffic flows in only one direction. However, the stream is ended by a packet with the TCP FIN flag set.

The `make_TCP_stream()` function is called with the arguments shown in Table 4.6. Table 4.7 explains each argument. Every argument is required, and a stream of two packets is returned if the `stream_length` argument is less than 2.

Table 4.7 Arguments for the `make_TCP_stream()` function

Argument Name	**Description**
stream_length	Length of stream to create
pkt_length	Length of all packets in the stream
ETH_DST	Ethernet Destination MAC Address for all packets in the stream in the output stream
ETH_SRC	Ethernet Source MAC Address for all packets in the stream in the output stream
TTL	TTL value for all packets in the stream in the output stream
IP_DST	IP Destination Address for all packets in the output stream
SRC_IP	IP Source Address for all packets in the output stream
TCP_DST	TCP Destination Port for all packets in the output stream
TCP_SRC	TCP Source Port for all packets in the output stream

4.2.1.4 The `make_TCP_duplex_stream()` Function

This function creates a bi-directional TCP packet stream. Unlike the `make_TCP_stream()` function, this function uses standard TCP 3-way handshake to begin a connection. The first three packets perform the 3-way handshake with the first packet source IP address and TCP

Table 4.8 Example `make_TCP_duplex_stream()` function call

```
make_TCP_duplex_stream(stream_length, pkt_length, ETH_DST, ETH_SRC,
    TTL, IP_DST, SRC_IP, TCP_DST, TCP_SRC)
```

port to the destination IP address and TCP port. The last packet is also sent from the source IP address-TCP port pair to the destination, but with the TCP FIN flag set. If a stream length of more than four is specified, the intermediate packets are randomly sent either from the source to destination used in the first and last packets or from the destination to the source specified in these packets.

The `make_TCP_duplex_stream()` function is called with the arguments shown in Table 4.8. Table 4.9 details what each argument specifies. All arguments are required, and the stream length is set to 2 if a length less than 2 is provided.

Table 4.9 Arguments for the `make_TCP_duplex_stream()` function

Argument Name	**Description**
stream_length	Length of stream to create
pkt_length	Length of all packets in the stream
ETH_DST	Ethernet Destination MAC Address for all packets in the stream in the output stream
ETH_SRC	Ethernet Source MAC Address for all packets in the stream in the output stream
TTL	TTL value for all packets in the stream in the output stream
IP_DST	IP Destination Address for all packets in the output stream
SRC_IP	IP Source Address for all packets in the output stream
TCP_DST	TCP Destination Port for all packets in the output stream
TCP_SRC	TCP Source Port for all packets in the output stream

4.2.2 The `test_thesis_short` Verification Test

The `test_thesis_short` simulation is a simple test designed to confirm that basic functionality of the design performs properly. Table 4.10 on page 35 demonstrates how to run this test using the verification testing scheme explained in 4.1. All packets input to this verification test are separated by 1 microsecond so the Input Arbiter will process them in order causing the

Table 4.10 Commands to start the `test_thesis_short` verification test.

```
[user@host]# NF2_DESIGN_DIR=/path/to/project/
[user@host]# nf2_run_test.pl --major thesis --minor short
```

Table 4.11 Packets sent for the `test_thesis_short` simulation

(1) Send 2 packets in on ports 1 and 2
(2) Send same packets as in (1), in same order, switching the source and destination
(3) Send same packets as (1), with TCP FIN set, in on port 1
(4) Send 1 packet with TCP FIN set
(5) Send original 2 packets with TCP FIN set
(6) Send same packet as in (4)
(7) Send same packets as in (5) in reversed order
(8) Send packets in on MAC ports 3 and 4, and on CPU DMA ports

outputs to be in the expected order. This test first creates two streams by sending two unique packets. Next, two packets are sent in the opposite direction (from destination to source of the original packets). Then packets are sent in the original direction with the TCP FIN flag set to close the connection. The two streams are reopened with expectation of being sent to the opposite monitoring output port as before, but the TCP FIN flag is again set to prevent adding the connections to the table. After this, the streams are again reopened with the TCP FIN flag set with the expectation that they will be forwarded to the opposite ports as they were the second time the connections were opened (since they should not have been added to the connection table). Finally, packets are sent in on the monitoring ports and the CPU DMA ports because the design must drop all packets input on these ports.

Table 4.11 explains the exact packet input procedure of this test. Unless specified otherwise, all numbered ports refer to the Ethernet MAC ports of the device. For all packets input on ports 1 and 2, the exact same packet is expected to be output on ports 2 and 1, respectively. In step 1, both packets should be seen as new streams (referred to here as "stream1" and "stream2" for simplicity), and thus output to ports 3 and 4, respectively. In step 2, stream1 should still be output to port 3, and likewise stream2 to port 4. In step 3, both streams should still be output on the same ports, but setting the TCP FIN should indicate the end of a

connection and cause the removal of the connections from the connection tables. Step 4 sends a single packet with the TCP FIN set (so the connection is not added to the table) that is seen as a new connection, output on port 3, and causes the next new connection to be output on port 4. Step 5 resends the two streams with TCP FIN set so they will not be added to the connection table. Since the packet sent in step 4 caused the next new connection to output on port 4, and the streams were closed in step 3, stream1 and stream2 are now new connections and thus should output to ports 4 and 3 respectively. Step 6 performs the exact same function as step 4 in switching the output port for the next new connection to port 3. Step 7 is the same as step 5, and since the TCP FIN flag was set on the packets in step 5, both stream1 and stream2 should again be seen as new connections, and thus output on output ports 3 and 4 respectively. Finally in step 8, input packets are sent on the monitoring ports 3 and 4 and on all four of the CPU DMA ports with the expectation that all of these packets will be dropped (not output on any port).

4.2.3 Simulation Waveforms for `test_thesis_short`

This section contains simulation waveforms taken from the `test_thesis_short` verificaton test. Each waveform shows the major functionality of a module and an associated explanation of the waveform draws attention to the major points of interest in the waveform.

Figure 4.1 on page 37 shows the processing of a single packet from the view of the Output Port Lookup module. Near the top left of the waveform, the packet is input on the `in_` buses and passed directly to the Header Parser module. This module takes 7 cycles to process the 64-bit packets needed to parse the Ethernet, IP, and TCP headers. Once parsing is complete, the module raises its `hp_done` output signal asserting the `lookup_req` input for the Connection CAM LUT. After 3 more cycles, the Connection CAM LUT module completes its processing, sets `dst_ports` to the proper value, and asserts the `dst_ports_rdy` signal. This triggers the Output Port Lookup module to read the input packet from the FIFO, update the internal header with the correct output ports, and begin outputting it during the next 2 cycles. Overall, this design requires 12 cycles of the 125 MHz clock to process a packet from a new connection.

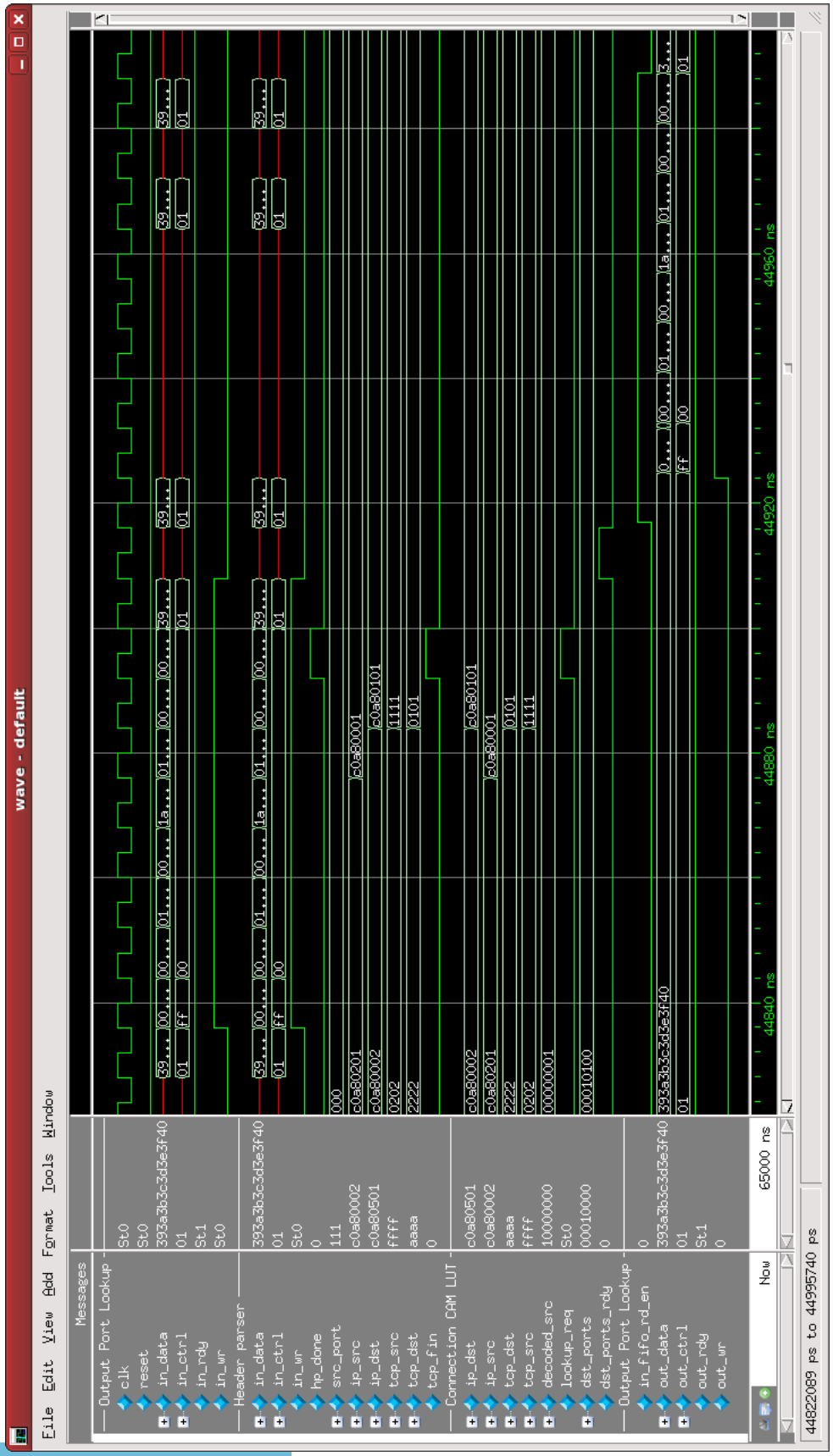

Figure 4.1 Simulation waveform showing the Output Port Lookup module's processing of a packet

A packet from an existing packet requires only 11 cycles of processing because it does not need to be written to the connection table. Although removing entries from the table requires a 2 cycle write, packet processing is only delayed for packets less than 56 bytes in length. Packets this short are completely buffered by the time the Header Parser completes processing, and thus the next packet enters immediately and will later encounter a single cycle delay if the packet before it was removed from the connection table. These processing cycle counts are constant, regardless of packet length. As a result, performance will be better with longer packets because there is a higher packet data to header ratio, and thus less processing per packet.

Figure 4.2 on page 39 depicts the Header Parser module's processing of a single packet. The `in_` signals shown at the top of the figure show a packet being input to this module in the 64-bit internal NetFPGA format. As the packet is input, the IP and TCP header fields for source and destination are output as the input packet is processed. Once all Ethernet, IP, and TCP fields are processed, the `hp_done` signal is asserted during the next to the last cycle shown in the figure.

Figure 4.3 on page 40 details the Connection CAM LUT module handling a packet input on port 1 without the TCP FIN flag set. During the first cycle shown, `lookup_request` is asserted, so the module begins processing the input IP and TCP ports. Also, `latch_inputs` is asserted to store the values of the inputs in case the connection later needs added to the connection table. Since this packet came in on port 1 (external network to internal network traffic), the IP source address and TCP source port comprise the connection identification tag. During the first cycle, `lookup_request` is asserted and both CAMs are queried for this tag. In the second cycle, neither CAM indicates a match, so the system configures a write to port 3's CAM during the third cycle. The design knew to use this CAM because a 0 on the `cam_toggler` wire indicated that a new connection was least recently assigned to this CAM. The value of this signal is toggled during cycle 4 to specify that the next new connection should use port 4's CAM. The connection tag is written to the CAM during the fourth cycle, and the CAM indicates that it is busy for the fifth cycle. During the seventh cycle, the `cam3_match`

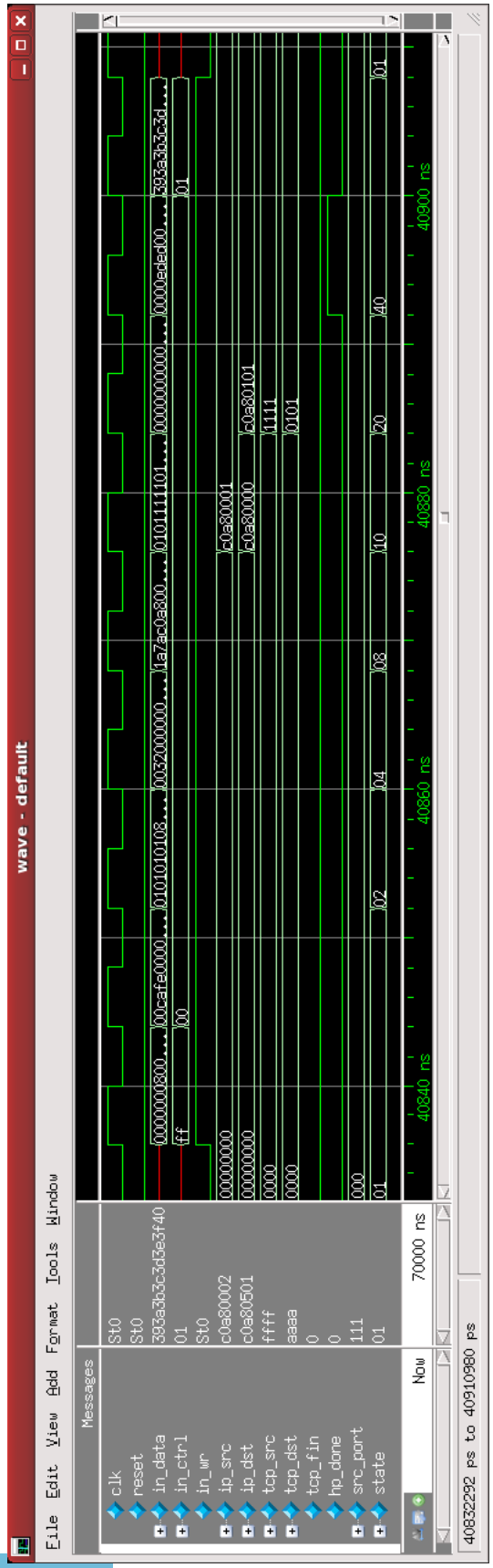

Figure 4.2 Simulation waveform showing the Header Parser module processing a packet

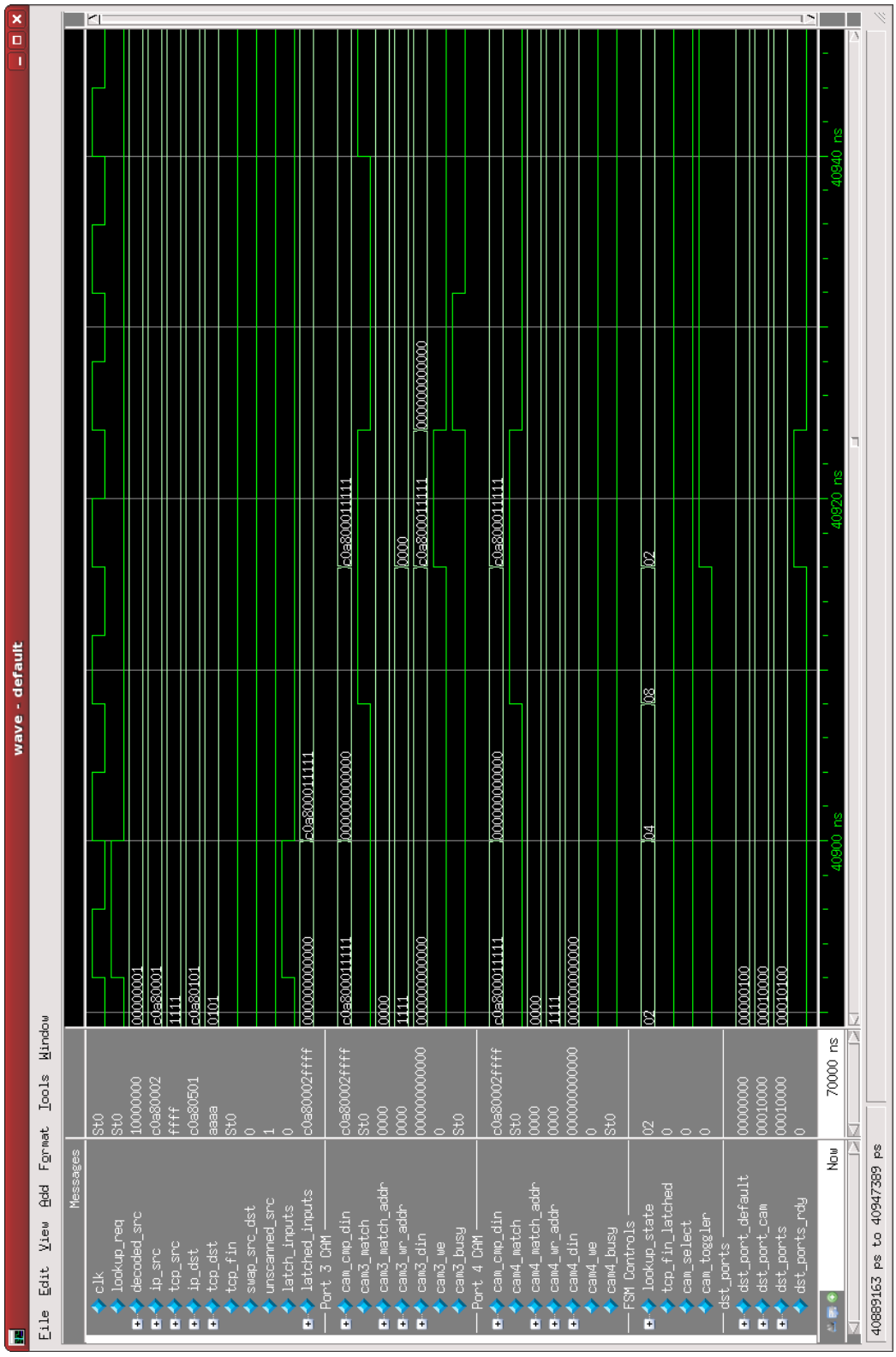

Figure 4.3 Simulation waveform showing the Connection CAM LUT module processing a packet input on port 1

signal is asserted while the connection tag is still on the compare input bus, so the value was successfully written to the CAM.

Figure 4.4 on page 42 depicts the Connection CAM LUT module processing a packet input on port 2. Since this packet is traveling from the internal network to the external network the IP address-TCP port pair used for the connection identification tag must be switched. The `swap_src_dst` wire is asserted to indicate this switch, and the `latched_inputs` bus stores the destination IP address and TCP port. Since this packet is a new connection and immediately follows the packet shown in Figure 4.3, it is added to the CAM for and output on port 4. The `cam_select` wire is set to 1 when `dst_ports_rdy` is asserted indicating that port 4 is added to `dst_ports`. At this point, adding the entry to the CAM proceeds as it did in Figure 4.3 but uses port 4's CAM rather than port 3's CAM.

Figure 4.5 on page 43 details the Connection CAM LUT processing a packet input on port 3. Throughout the processing of this packet, the `unscanned_src` wire is asserted indicating that this packet came from an unmonitored port (any port besides MAC ports 1 and 2). As a result, neither CAM is configured for a write during cycle 4 as was the case for packets input on ports 1 and 2. The `cam_toggler` and `cam_select` signals are unaltered, and no ports are indicated for output by `dst_ports` since it is set to "00000000" when `dst_ports_rdy` is asserted.

Figure 4.6 on page 44 shows the Connection CAM LUT module handling a packet input on port 1 with TCP FIN set. Processing starts similar to the previous two examples as the connection identification tag is looked up in both CAMs during the first cycle. During the second cycle, `cam3_match` is asserted indicating that the connection is assigned to port 3. As a result, port 3 is added to `dst_ports` and `dst_ports_rdy` is asserted during cycle 3. However, during cycle 2, the `tcp_fin_latched` signal is asserted indicating that this packet had TCP FIN set. Accordingly, the CAM is configured for a write of 0 to the address where this connection identification was found during cycle 3. The CAM asserts the `cam3_busy` signal during cycle 4 to indicate that it is processing the write. Notice that the connection identification tag no longer matches after the write completes.

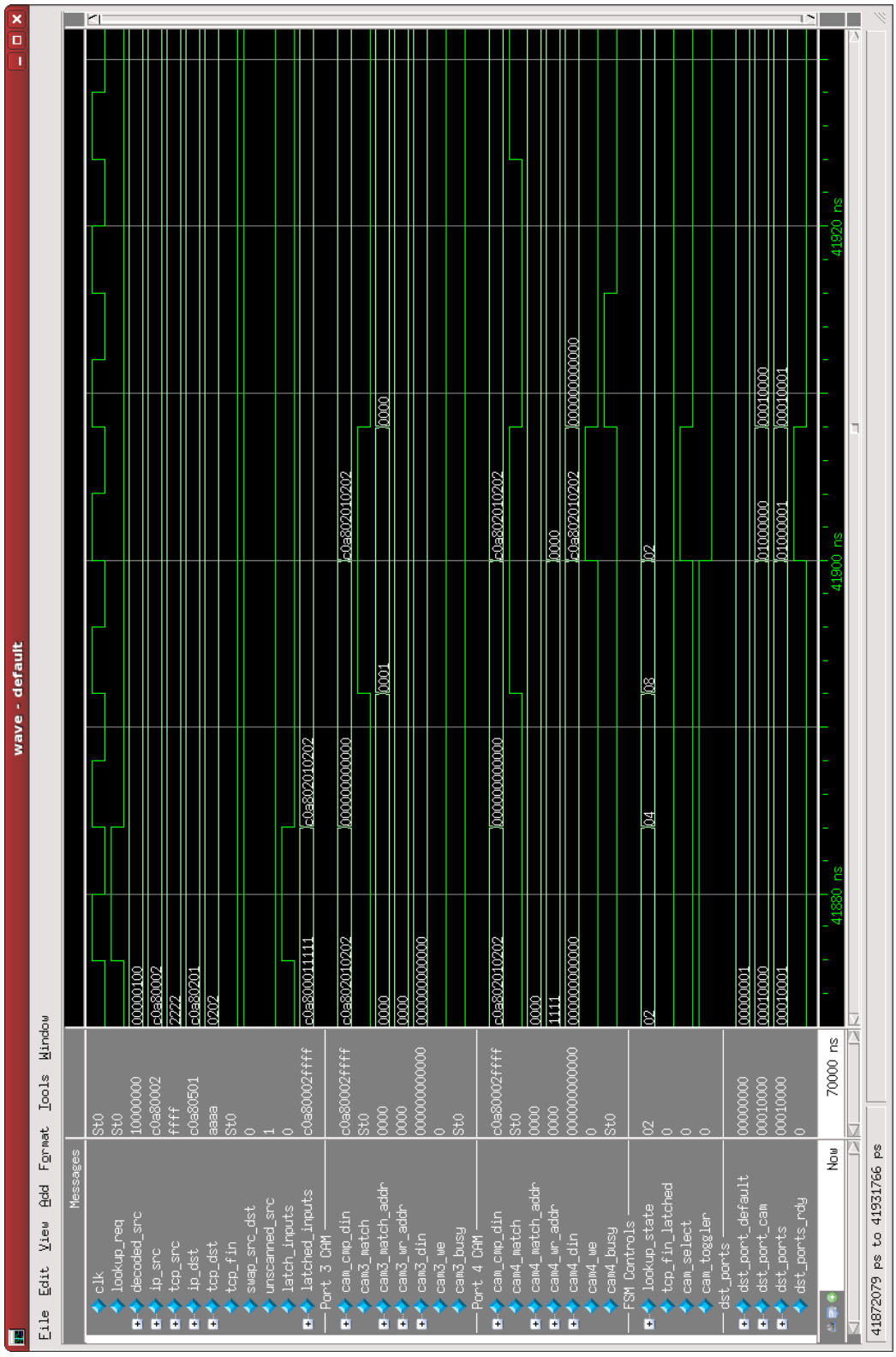

Figure 4.4 Simulation waveform showing the Connection CAM LUT module processing a packet input on port 2

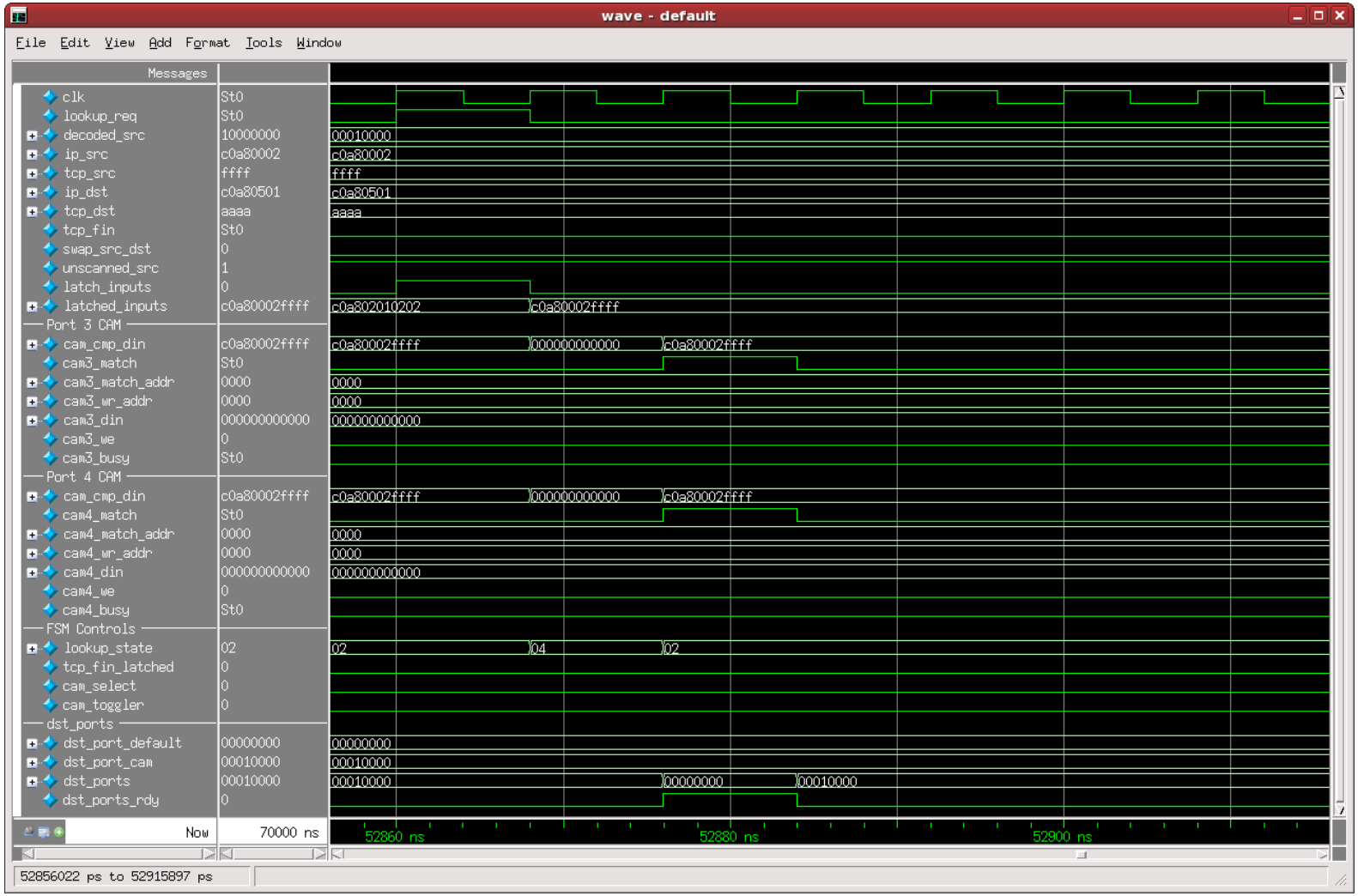

Figure 4.5 Simulation waveform showing the Connection CAM LUT module processing a packet input on the unforwarded port 3

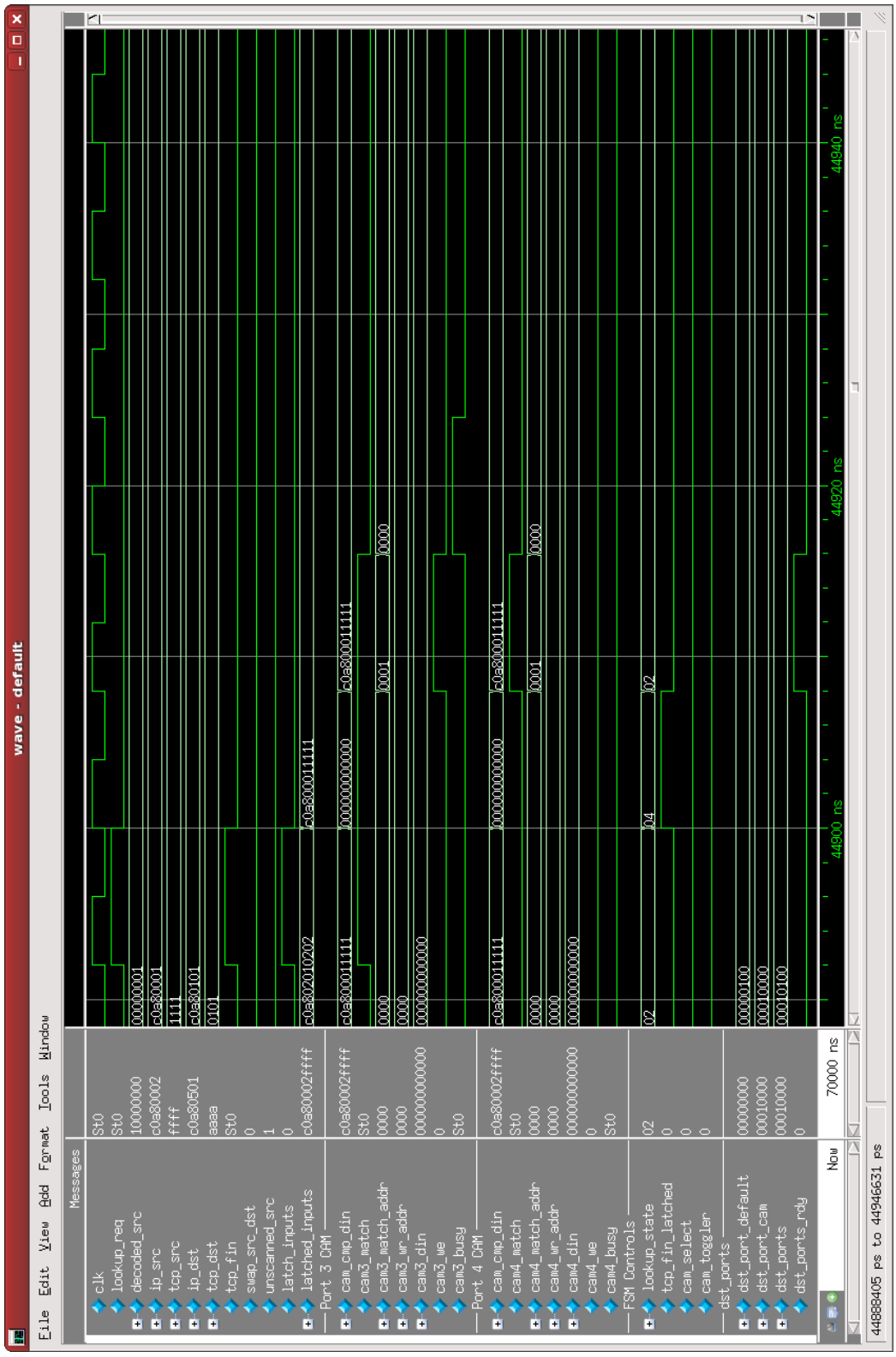

Figure 4.6 Simulation waveform showing the Connection CAM LUT module processing a packet input on port 1 with TCP FIN set

Table 4.12 Commands to start the `test_thesis_long` verification test.

```
[user@host]# NF2_DESIGN_DIR=/path/to/project/
[user@host]# nf2_run_test.pl --major thesis --minor long
```

4.2.4 The `test_thesis_long` Verification Test

The `test_thesis_long` simulation attempts to test all functionality of the design for this thesis. Using the conventions and environment described in section 4.1, this verification test is started using the commands shown in Table 4.12. Packets for this test are again input in 1 microsecond increments so the Input Arbiter will process them in order to preserve the expected output packet order. At a high level, the test first sends four bi-directional TCP streams including a TCP FIN packet to close the connection at the end of the stream. Then all four connections are reopened using the last packet of the stream (with the TCP FIN flag set, so they will not be added to the connection table). The simulation is then configured to reopen these connections such that they will each be output on the opposite monitoring port as before. These same four last packets are repeated again to confirm that the packets with TCP FIN set were not added to the connection table. Next, 33 unique packets are sent, creating 33 unique connections. This will overflow the connection table by one packet, and this packet is expected to be dropped. The first 32 of these packets are resent expecting that the connections were added to the table, and thus all packets will output on the same monitoring port as before. Finally, packets are input on the monitoring MAC ports and the four CPU DMA ports with all six packets expected to be dropped.

Table 4.13 on page 46 shows the exact procedure for this test. In step 1, the four bi-directional TCP streams are created. Each stream is between 2 and 10 packets in length, and each packet is 64 bytes long. All but the last packet of each stream are sent in step 2, and the last packet of each stream is sent in step 3. In step 4, since the number of connections is even, the test sends a single packet as an offset to force the next new connection to use the opposite monitoring port of when the simulation began. In step 5, all four connections are reopened in the same order with the last packet of the original stream (thus the TCP FIN flag is set to

Table 4.13 Packets sent for the `test_thesis_long` simulation

(1) Create 4 bi-directional TCP streams of 64 byte packets
(2) Send all but the last packet of each stream
(3) Send the last packet of each stream
(4) Send 1 packet if number of streams is even
(5) Resend the last packet of each stream
(6) Send 1 packet if number of streams is even
(7) Resend the last packet of each stream
(8) Send 33 unique packets
(9) Send first 32 packets of (8)
(10) Send packets in on MAC ports 3 and 4, and on CPU DMA ports

avoid adding the connections to the table). As a result of the offset packet, each packet should output on the opposite monitoring port as before. This indicates that the design successfully removed the connections from the table when it encountered the TCP FIN packet. Steps 6 and 7 perform the exact function as steps 4 and 5 to ensure that the connections created in step 5 were not added to the connection table when the first packet received had the TCP FIN flag set. Thus, each packet should output on the opposite port as it did in step 5. Step 8 sends 33 packets, or one more than the connection table can store, and the last packet is expected to be dropped. Step 9 resends the first 32 packets of step 8 expecting output for each packet on the same port to ensure that all entries in the connection table are accessible and function properly. Step 10 sends packets in on all ports that are not forwarded, and thus expects the design to drop all packets sent in this step.

4.3 Hardware Testing

The implementation of this design is not tested in the actual NetFPGA hardware. Using the NetFPGA Makefile environment for the project, a bitfile was successfully generated and downloaded to hardware. However, this hardware implementation is untested because proper testing requires sending and receiving network traffic on 4 physical Ethernet ports connected to the NetFPGA. Although hardware testing was not performed, successfully generating a bitfile means that device utilization information is available. Table 4.14 on page 47 shows the Device

Table 4.14 Summarized results of Place and Route for the design

```
Device Utilization Summary:

   Number of BUFGMUXs                        8 out of 16     50%
      Number of LOCed BUFGMUXs               8 out of 8     100%

   Number of DCMs                            6 out of 8      75%
   Number of External IOBs                 356 out of 692    51%
      Number of LOCed IOBs                 356 out of 356   100%

   Number of RAMB16s                       112 out of 232    48%
   Number of SLICEs                      12997 out of 23616  55%
```

Utilization Summary provided by the Xilinx ISE 9.2i Place and Route tool. These results indicate that the design uses less than half of the memory (RAMB16s) and just over half of the logic resources (SLICEs) available on-chip. In addition, timing analysis (not shown here) reveals that for a clock period requirement of 8 ns, the best case clock period for the design is 7.963 ns.

CHAPTER 5. FUTURE WORK

This chapter discusses future work related to this project. It is not intended to be all-inclusive. Instead it identifies directions for future work, and possible projects in these directions. Included in this chapter are discussions of future work in testing the design, extending it, and expanding upon it.

Although a bitfile was successfully generated for this project, the design's performance in hardware is untested. Thus, possible future work is to design and implement a hardware test bench for this design. The NetFPGA Cube used for this design has four physical Ethernet ports in addition to those on the NetFPGA, so this system could possibly be used to test the hardware design. Regardless of the system chosen, a hardware test bench will allow for verifying the hardware implementation and may offer some performance benchmarking.

One possible extension to this work is to create a similar solution that is not only stream-aware, but also stream type aware. For example, all web traffic could be forwarded to a certain monitoring system. This allows more specialized monitoring systems to process larger rule sets for specific connection types. Since the monitoring system would only be responsible for specific traffic types, rules for other traffic types could be excluded in favor of more specific rules for these types.

Another extension is to parallelize the processing and distribution of incoming traffic. The current solution processes traffic inbound and outbound on two different ports with the same logic. With duplicate processing logic, the design could process traffic inbound on ports 1 and 2 in parallel. Implementing this extension would require duplicating the Output Port Lookup logic, and replacing the input arbiter from the design presented in this thesis with a module that retrieves incoming packets from multiple ports simultaneously.

A third direction for future work is to develop an algorithm to reduce the number of bits used to store a connection identification tag. The current design uses the concatenation of the 32-bit IP address and 16-bit TCP port from the wide area network side of the system to form a 48-bit connection identification tag. Using such a large tag in the connection lookup table reduces the number of active connections supported by this design. It is worth noting at this point that conflicts in connection identification tags are not a major issue because all packets of a given stream will still be forwarded to the same monitoring system. The issue with conflicts is that they disproportionately increase the workload on the monitoring system whose table encountered the conflict. In other words, the conflicting connection identification entry in the table now represents two connections rather than one. This is only a major issue when many conflicts occur, so selection of a reduction algorithm should consider the types of conflicts that may be encountered.

Another direction for future work is to expand the design to share the intrusion detection load between itself and the backend intrusion detection systems to create a hierarchical detection system. One possible example is to process a set of simple intrusion detection rules in hardware and only forward suspicious connections to the backend, software based intrusion detection systems. Another possibility is to implement basic intrusion prevention system functionality in hardware and to forward traffic to backend intrusion detection systems for more in-depth processing. (Intrusion prevention systems may be thought of intrusion detection systems that take action to prevent intrusions.)

Beyond design modifications, research into the number of active connections to maintain in the connection table would prove beneficial. Currently the design supports 32 active connections as a proof of concept, and all connections beyond this number are unmonitored. Due to space and timing limitations of most FPGAs, the size of this table will likely be limited regardless of the FPGA used. As a result, it is important to determine the typical size of the set of active connections in small business networks in which this solution is most likely to be deployed. Knowing the size of this set will allow minimization of the size of the connection table.

CHAPTER 6. CONCLUSION

This thesis presented a low-cost, connection aware, load balancing solution for distributing network traffic to multiple intrusion detection systems. It included a discussion of some related solutions, and a short introduction to the NetFPGA platform used to implement the design. It presented the design architecture used for the solution, and the details of its implementation. The testing performed on this design and the related support functionality added for testing was discussed. Finally, it identified some possible directions for future work related to the solution.

Overall, this project successfully accomplishes its goal of providing a low-cost, connection aware, load balancing solution for distributing network traffic to multiple intrusion detection systems. The design proved to be fully capable of addressing this problem. It provides a proof of concept system that could be implemented on newer hardware to support monitoring more active connections. Regardless of the hardware used, this design provides a solid foundation for expansion to implementing more intelligent in network traffic distribution systems. In addition to the solution itself, this project caused the development of valuable simulation and testing components. The `TCP_hdr` package provides the functionality for the NetFPGA simulation libraries to support creating TCP packets. The `make_TCP_pkt()`, `make_TCP_stream()` and `make_TCP_duplex_stream()` Perl functions create packets to test designs with TCP traffic. The combination of all of these additions to the Perl test environment provided by the NetFPGA Package provides a foundation for expansion into creating and sending more complicated TCP traffic patterns. Overall, this project proved successful, offering not only a working solution to distribute traffic to multiple intrusion detection systems, but also a foundation for creating more intelligent traffic distribution solutions.

BIBLIOGRAPHY

[1] immixGroup, "GSA Schedule 70 | Top Layer Networks," [Online] Available: http://var.immixgroup.com/contracts/gsa70_pricing.cfm?client_id=25&contract=GS-35F-0330J [Accessed: April 5, 2010].

[2] PEPPM, "Pricelist Template Form," [Online] Available: http://www.peppm.org/Products/TopLayer/ca/price.pdf [Accessed: April 5, 2010].

[3] IP Fabrics, "IP Fabrics' DeepSweep-1," [Online] Available: http://www.ipfabrics.com/products/deepsweep1.php [Accessed: April 5, 2010].

[4] Top Layer Security, "Network Intrusion Prevention System IPS for Computer Network Security Management," Available: http://www.toplayer.com/ [Accessed: April 5, 2010].

[5] Coyote Point Systems Inc., "Server Load Balancing Hardware — Application Acceleration Solutions — Coyote Point Systems," [Online] Available: http://www.coyotepoint.com/products/ [Accessed: April 5, 2010].

[6] NetFPGA, "NetFPGA," [Online] Available: http://www.netfpga.org/php/specs.php [Accessed: April 2, 2010].

[7] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown, "NetFPGAAn Open Platform for Teaching How to Build Gigabit-Rate Network Switches and Routers," *IEEE Transactions on Education*, vol. 51, 364-369, Aug. 2008, Available: http://netfpga.org/documents/NetFPGA-Transactions_on_Education-Aug_2008.pdf [Accessed: April 2, 2010].

[8] Digilent Inc., "Digilent Inc. - Digital Design Engineer's Source," [Online] Available: http://www.digilentinc.com/Products/Detail.cfm?Prod=NETFPGA [Accessed: April 5, 2010].

[9] Accent Technology, "Accent Technology - NetFPGA Pre-Built Solutions," [Online] Available: http://www.accenttechnologyinc.com/netfpga.php?category_id=0&item_id=1 [Accessed: April 5, 2010].

[10] NetFPGA, "Guide < NetFPGA/OneGig < Foswiki," [Online] Available: http://netfpga.org/foswiki/pub/NetFPGA/OneGig/Guide/ReferencePipeline.jpg [Accessed: April 2, 2010].