Dynamic Silicon Firewall

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Electrical Engineering
University of Saskatchewan
Saskatoon

By
Darrell Laturnas

Permission to Use

Head of the Department of Electrical Engineering
57 Campus Drive
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Computers are networked together in order to share the information they store and process. The internet connects many of these networks together, offering a multitude of options for communication, productivity and entertainment. It also offers the opportunity for unscrupulous individuals to contact these networked computers and attempt to appropriate or destroy the data on them, the computing resources they provide, and the identity or reputation of the computer user. Measures to secure networks need to be implemented by network administrators and users to protect their computing assets.

Firewalls filter information as it flows through a network. This filter can be implemented in hardware or software and can be used to protect computers from unwanted access. While software firewalls are considered easier to set up and use, hardware firewalls are often considered faster and more secure. Absent from the marketplace is an embedded hardware solution applicable to desktop systems.

Traditional software firewalls use the processor of the computer to filter packets; this is disadvantageous because the computer can become unusable during a network attack when the processor is swamped by the firewall process. Traditional hardware firewalls are usually implemented in a single location, between a private network and the internet. Depending on the size of the private network, a hardware firewall may be responsible for filtering the network traffic of hundreds of clients. This not only makes the required hardware firewall quite expensive, but dedicates those financial resources to a single point that may fail.

The dynamic silicon firewall project implements a hardware firewall using a soft-core processor with a custom peripheral designed using a hardware description language. Embedding this hardware firewall on each network interface card in a network would offer many benefits. It would avoid the aforementioned denial of service problem that software firewalls are susceptible to since the custom peripheral handles the filtering of packets. It could also reduce the complexity required to secure a large private network, and eliminate the problem of a single point of failure. Also, the dynamic silicon firewall requires little to no administration since the filtering rules change with the users network activity. The design of the dynamic silicon firewall incorporates the best features from traditional hardware and software firewalls, while minimizing or avoiding the negative aspects of each.

Acknowledgements

I would like to acknowledge Dr. Ron Bolton for his guidance and encouragement throughout my post-graduate program. His professionalism, humor, and patience were greatly appreciated.

I also need to thank the management and staff of Telecommunications Research Laboratories (TRLabs) for providing financial assistance and the use of their facilities throughout my research. The knowledge I gained and the equipment I used throughout my time with TRLabs were indispensable.

Finally, I would like to express gratitude to my family for believing in me throughout my education. I have never felt that anything was too hard to learn and never doubted my ability to succeed, due largely in part to their faith in me.

This thesis is dedicated to the greatest teachers I have known: Alan Melenchuk in primary school, Donald Nakonechny in secondary school, Jerry Huff, Safa Kasap and Ron Bolton in university. To teach will always remain a noble pursuit, as it improves not only the individual, but future generations of mankind.

Contents

List of Figures

List of Abbreviations

ARDY	Asynchronous Ready
ARP	Address Resolution Protocol
ASIC	Application-Specific Integrated Circuit
CAM	Content-Addressable Memory
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
EPH	Ethernet Protocol Handler
FPGA	Field-Programmable Gate Array
FTP	File Transfer Protocol
HDL	Hardware Description Language
HTTP	HyperText Transport Protocol
IP	Internet Protocol
LAN	Local Area Network
MAC	Media Access Control
OS	Operating System
OSI	Open Systems Interconnection
PCI	Peripheral Component Interconnect
PDA	Personal Digital Assistant
PIO	Parallel Input/Output
PLD	Programmable Logic Device
PPP	Point-to-Point Protocol
RAM	Random Access Memory
SOPC	System On a Programmable Chip
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Chapter 1
Introduction

This chapter begins by outlining the computer security problems motivating this research. The research goals are then presented, followed by an outline of the entire thesis.

1.1 Motivation

This research is motivated by many factors in the realm of home computer security. It is becoming quite common in developed nations for people to have high-bandwidth internet connections in their homes, often connected to powerful personal computers. This combination of computing power and communication capability has become an inviting target for malicious hackers and programmers. By corrupting or gaining access to a computer, a malicious programmer may destroy or steal the information stored on that computer. Furthermore, they may appropriate the computer and network resources and use them for their own purposes, denying access to the legitimate user. Some malicious programmers then use these co-opted computers to spread worms, viruses and other forms of malicious software, often called malware. Email spam is sent out from these computers as well, in an effort to make money through fraudulent means and continue the spread of malware.

Gaining access to one computer system using another computer on a remote network is not trivial. To formulate a plan of attack requires knowledge of the victim computer, which operating system (OS) it is running, the services that the victim is providing to the network, and/or the software installed on that computer. Also, the underlying technology being used to provide network access to the victim computer provides information on how to find and communicate with the victim. The operating system, software, and network services being used on a computer are choices left to the computer user. These choices are often made without security in mind. Users may choose to use an operating system because it comes pre-installed when they purchase a computer, or because it is the only one compatible with their favorite software. Similarly, the networking technology being used is a choice the user must make, though this choice is often determined by the choice of network service provider, since the home network technology must be compatible with the service provider's network technology. This most often results in choices which are not consciously made by the user, leading to the majority of home computer users running a Windows® operating system on their computer, with Ethernet networking technology connecting their computer to a home network or the internet. The popularity of Windows® operating systems and the Ethernet networking protocol has resulted in these technologies being the primary targets of malicious programmers. By designing malware for a Windows® operating system, the programmer knows that a large number of computers will be vulnerable to it, simply due to the popularity of the OS. Similarly, software for monitoring Ethernet networks has been developed which allows malicious hackers to get information from network traffic regarding which computers are on the network and which services they are using or providing.

The responsibility to maintain secure computers and networks for the average home user falls to three entities. First, Microsoft®, the maker of Windows® operating systems, releases security patches to help secure the operating system. Second, the user's internet service provider must monitor their networks to mitigate the spread of malicious software and to look for unusual network traffic that may be an indicator of a network attack. Finally, a large portion of computer security is left to the home user. Many commercial software products for home users have been developed to address the threats presented by the various malicious programs, such as viruses. In addition to software, there have been consumer hardware security products developed for computer users at home. The problem which has arisen is that many of these consumer products require knowledge, about the underlying technology being used, to be used effectively. This is knowledge which the average home user does not possess. The malicious programmers developing security attack strategies do have this knowledge, and they are using it.

1.2 Research Objective and Thesis Outline

This research explores how an embedded firewall, designed using a hardware description language, may improve upon the positive features and mitigate the negative aspects of traditional firewall implementations. The main focus is on the method of implementing a simple yet useful firewall for network users who are naive to the underlying technology and communication protocols involved in computer networking. This project will build upon and seek to improve upon the limited research done in the area of embedded hardware firewalls [1, 2].

The fundamentals of network functionality and security are presented in chapter 2. In chapter 3, the components and tools used in the design of embedded systems are discussed. The hardware design of the dynamic silicon firewall is presented in chapter 4, while the software components of the design are discussed in chapter 5. In chapter 6, the testing process and results of the conducted tests are explained. In chapter 7, the conclusions drawn from this project are presented and the direction for future work is suggested.

Chapter 2

Computer Networking

This chapter focusses on the functionality and security vulnerabilities of computer networks. In order to better understand the way computer networks are attacked and the ways in which data is vulnerable, the fundamentals of computer networking are presented. Approaches to keep computer networks secure are also discussed, with an emphasis on network packet filters.

2.1 Network Models

Computers are networked together so that information can be shared. Networks, such as the internet, are very complex and made up of multiple communication protocols. Often, engineers and computer scientists organize the protocols into layers for easier understanding. Two main models of these layers exist [3], the most widely used are the Open Systems Interconnection (OSI) and the Transmission Control Protocol / Internet Protocol (TCP/IP) models. The TCP/IP model focusses on the hardware involved in computer networking, while the OSI model extends the TCP/IP model to include the software involved in computer networking. For the purposes of discussion, the TCP/IP model is more than adequate.

The TCP/IP model consists of five layers: Application, Transport, Network, Data Link and Physical. The application layer concerns the network software the end user is using. In the TCP/IP model, examples of the application layer would include web browsers and instant messaging software. The transport layer involves the protocols that applications use to communicate with each other. The transport layer is also the layer where any desired communication reliability is implemented, since the network, data link, and physical layers provide no guarantees for successful transmission. Furthermore, flow control is implemented at the transport layer. Examples of transport layer protocols are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). The network layer is responsible for routing information from one computer to another. It is also responsible for connecting multiple networks together. This is accomplished in the internet with the aptly-named Internet Protocol (IP). The data link layer is where data is encoded into bits for transmission onto the network medium. The most popular data link layer protocol used in home networking is the Ethernet protocol. Another example of a data link layer protocol is the Point-to-Point Protocol (PPP). The lowest layer in the TCP/IP network model is the physical layer. The physical layer is the actual medium used for communication. Examples of networking media are twisted-pair copper wires, coaxial cable, and optical fiber. Figure 2.1 illustrates a network using the TCP/IP network model. Data being sent from the PC on the left of the diagram travels down through the layers of the network model and onto the network. As the data is routed to its destination it may encounter a number of routers, requiring inspection of the data contained at the network layer. When the message reaches its final destination, the PC must go back up through the various layers in the network model to get to the application data.

2.1.1 Application Layer

Many protocols operate at the application layer since applications often provide a variety of services. For example, a web browser application could implement not only the HyperText Transport Protocol (HTTP), used to access web sites, but also the File Transfer Protocol (FTP) if a user directs the web browser to an FTP server. Application layer protocols implement the functionality needed for a specific application to function, but they abstract the underlying network and rely only on the transport layer for transmission of data across the network.

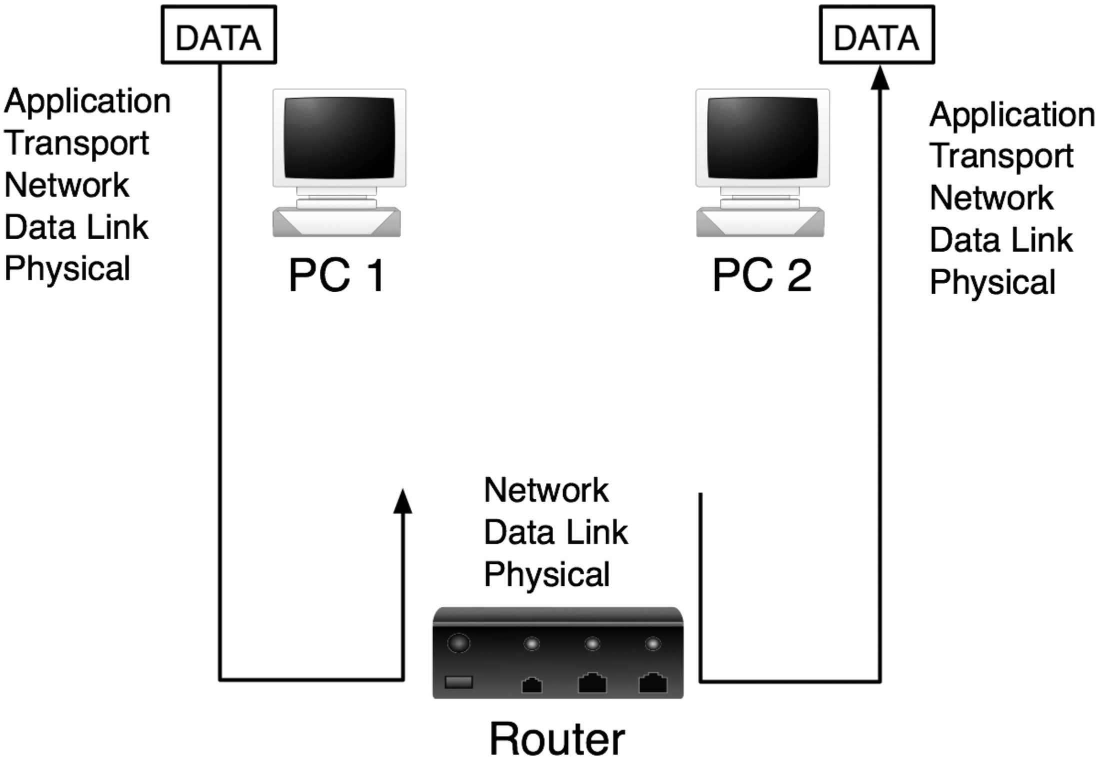


Figure 2.1: Example TCP/IP Modeled Network

2.1.2 Transport Layer

The transport layer takes data from the application layer and passes it to the network layer. The main protocols that operate at the transport layer are the Transmission Control Protocol [4] (TCP) and the User Datagram Protocol [5] (UDP). The message passed from the application layer has data prepended to it, this data forms a header used to implement the transport layer protocols. Essential to all transport layer protocols are the source and destination port numbers. Port numbers represent a service running on a host, and they provide a way to differentiate data from multiple applications running on a single host.

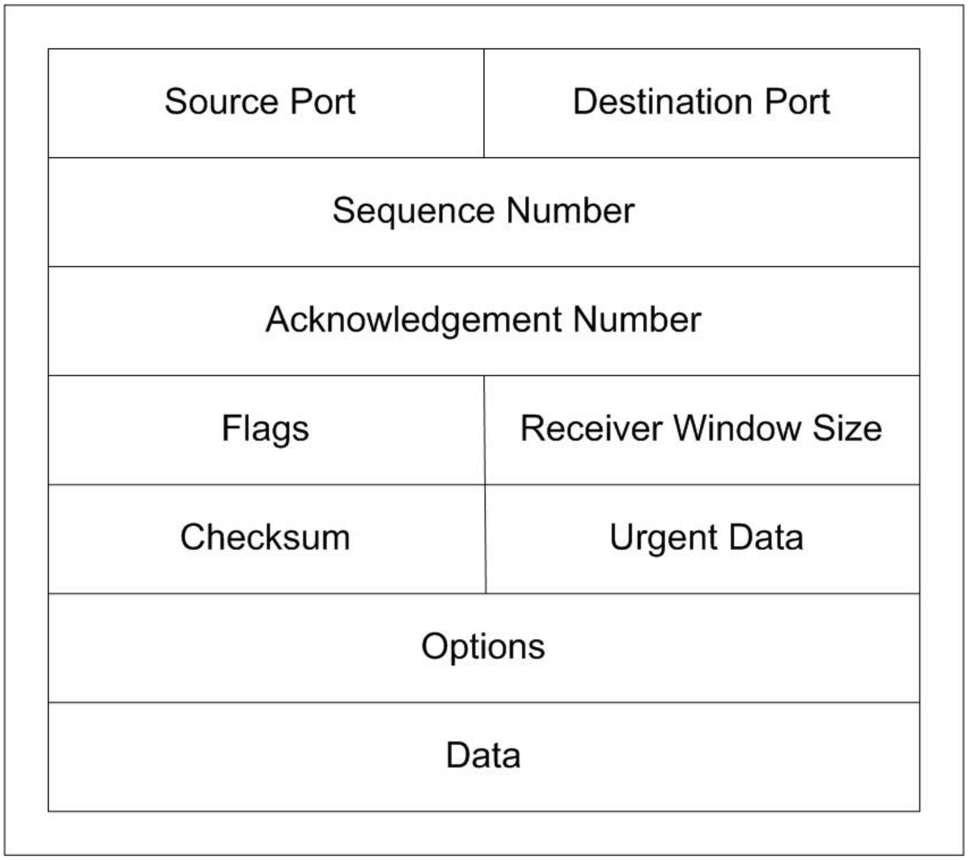
UDP is a simple transport layer protocol. It is made up of only four pieces of data, each consisting of two bytes, that are prepended to the application layer message. This UDP header consists of the source and destination ports, as well as the length and checksum fields. The length field contains the size of the UDP segment, in bytes. The checksum field allows the destination host to check if any errors were introduced during transmission. This does not ensure successful transmission, but it does allow corrupted data to be discarded. As mentioned, the port numbers allow the application layer data to be delivered to the proper application running on the destination host.

TCP is a more complex transport layer protocol, providing reliability and congestion control to the network. The TCP segment structure is shown in figure 2.2. The source and destination ports remain, as in UDP, but many other header fields are also required. The sequence and acknowledgement numbers are used to ensure reliable delivery of data. When a large piece of data is being transferred across a network, it is broken into pieces. The total data bytes being transferred are counted and each group of bytes in each TCP segment are given sequence numbers so that the data can be reconstructed after being delivered out of order. The acknowledgement number of an outgoing packet is the sequence number for the next byte expected from an incoming packet. Sending an acknowledgement number means that all data prior to the specified byte has been received successfully. The other feature that TCP provides to the network is congestion control. This is a service that benefits the network as a whole. When a network is very busy and saturated with packets, data can be lost due to packet buffer overflows. TCP provides mechanisms for recognizing network congestion, and reducing packet flow until the congestion is reduced.

2.1.3 Network Layer

The network layer takes a UDP or TCP segment from the transport layer and sends it to the data link layer for transmission. As with the transport layer, network layer protocols are implemented by adding data to the transport layer segment, forming what is called a network layer datagram. Internet Protocol datagram is an appropriate term when the network layer protocol being used is IP. The Internet Protocol [6] operates at the network layer and is responsible for the routing of datagrams from one computer to another across a network.

Figure 2.3 illustrates the IP datagram format. At the heart of the Internet



Source Port	Destination Port
Sequence Number	
Acknowledgement Number	
Flags	Receiver Window Size
Checksum	Urgent Data
Options	
Data	

Figure 2.2: TCP Segment Structure

Protocol is the Internet Protocol address, or IP address. The IP address is made up of four bytes that define a specific computer on a network. These four bytes can be divided into the network and host portions of the IP address. The network portion is used to identify a group of hosts that all belong to the same network, while the host portion of the IP address specifies an individual host on a network. The network and host portions of the IP address are essential in routing, as packets must first reach the proper destination network before they can reach the proper destination host.

The Address Resolution Protocol [7] (ARP) also operates at the network layer and is responsible for connecting hardware media access control addresses and IP addresses. Media access control addresses will be discussed further in the data link layer section. ARP is an essential service provided by the network layer as it connects the network IP address with the addresses used at the data link layer.

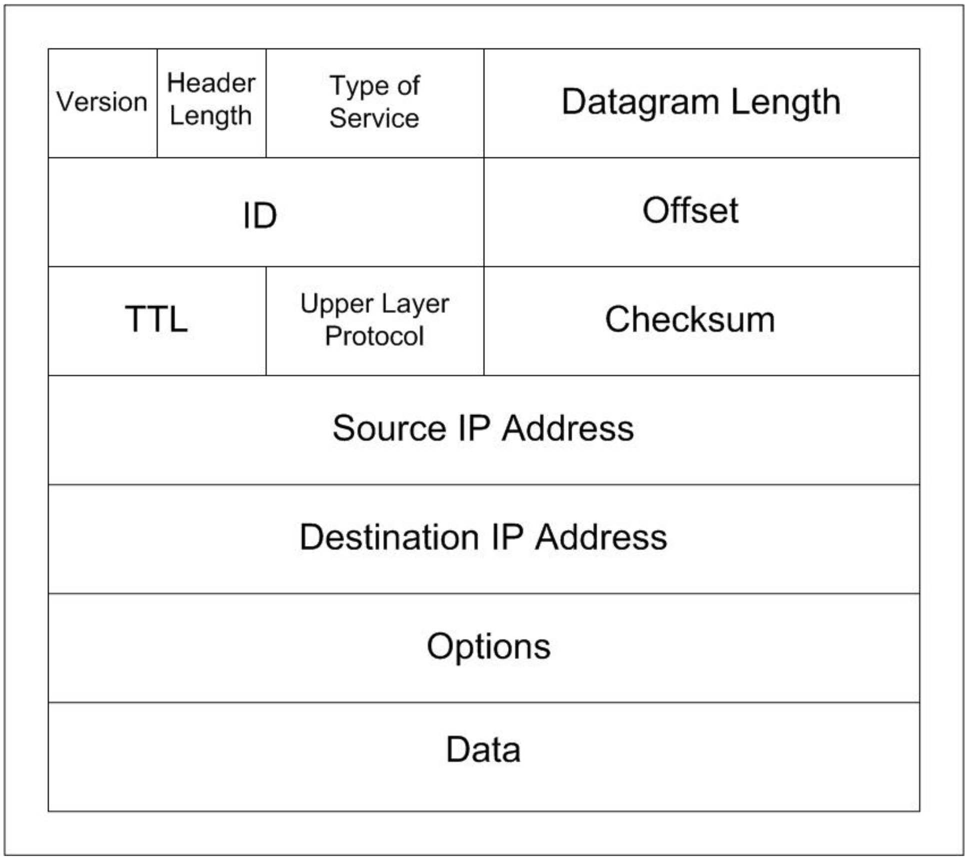


Figure 2.3: IP Datagram Structure

2.1.4 Data Link Layer

The Ethernet Protocol [8] operates at the data link layer and defines what signals to send on the physical layer to allow two computers to communicate. Since its invention in the 1970s, it has become the dominant networking protocol for local area networks. An Ethernet network can run over twisted-pair copper wire, coaxial cable, or optical fiber. It can also run at a variety of speeds: 10Mbps, 100Mbps or 1Gbps. Independent of transmission medium and speed, all Ethernet flavors share a few characteristics.

All Ethernet variants are unreliable connectionless services. Ethernet is connectionless in that an adapter does not directly contact the adapter with which it wants to communicate. An Ethernet adapter encapsulates information into a frame of data, often called a packet, and transmits that onto the transmission medium. All adapters connected to the medium then receive that frame of information. Ethernet is considered unreliable because there is no acknowledgement from the intended recipient of the information as to whether the frame was received without error. This raises the question of how the intended recipient knows whether a frame is meant for it, and how it knows the frame is error-free.

Ethernet relies on hardware addresses that are unique to each hardware adapter. These addresses are often called Media Access Control (MAC) addresses, and they consist of a unique six-byte identifier. The MAC address of the intended recipient of an Ethernet frame is included in the frame itself (as is the MAC address of the transmitting adapter), so adapters can check each packet of information to see if it is for them. Once an adapter has determined that it is the intended recipient of a packet, it must check if any errors were introduced during transmission. This is done with a Cyclic Redundancy Check (CRC). Before sending a frame out onto the transmission medium, an adapter appends four bytes to the end of the frame which are obtained from a mapping of the other bits in the frame. This allows receiving adapters to check for errors by performing the same mapping of the data and comparing the results. If the CRC code calculated by the transmitting adapter and sent with the frame does not match the CRC code calculated at the receiving adapter, then it can be concluded that an error occurred during transmission.

Now that the functionality of the Ethernet protocol has been discussed, the Ethernet packet structure can be examined. Figure 2.4 illustrates the fields of an Ethernet packet. The preamble consists of 8 bytes where the first seven bytes consist of the repeated binary pattern 10101010. The eighth byte is 10101011. Alternating bit values allow a receiving adapter to synchronize to the clock of the transmitting adapter, allowing bit positions to be determined. The final byte ends with two high bits. These bits let the adapter know that the preamble is over. The destination and source MAC addresses, along with the CRC field allow adapters to recognize packets that are destined for them and to ensure they have been received without error. Encapsulated between the MAC addresses and the CRC are the type and data fields, which make up the information from the network layer. The type field is a two-byte value that defines the network layer protocol being used, while the data

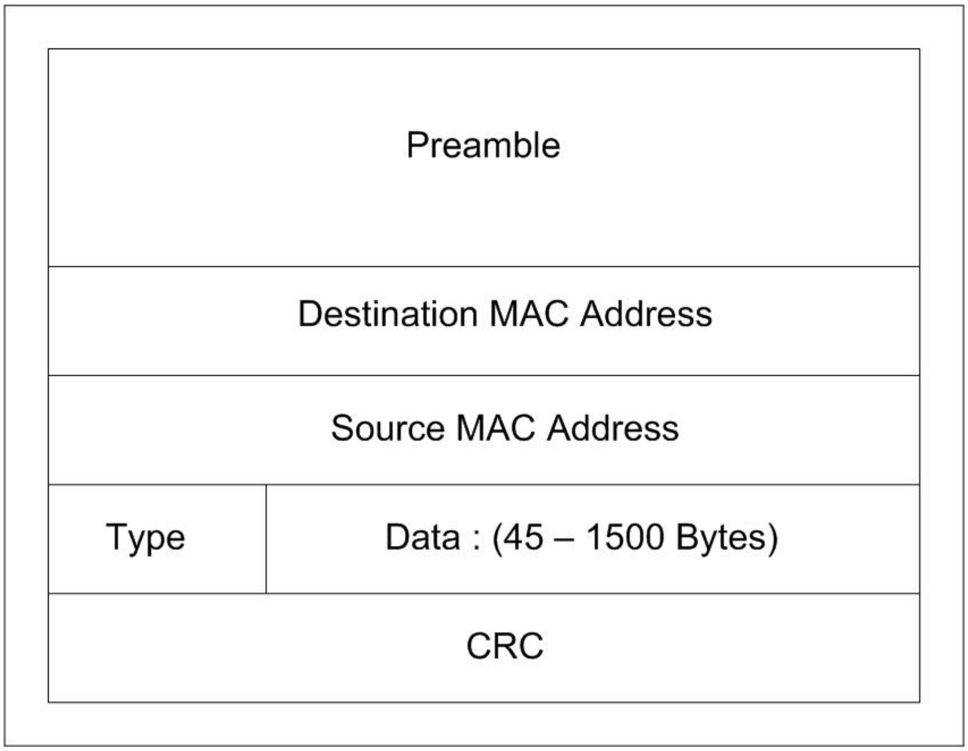


Figure 2.4: Ethernet Packet Structure

field contains the information being sent by the network layer.

2.1.5 Physical Layer

As previously mentioned, the physical layer is the lowest layer in the TCP/IP network model. It represents the physical medium used in communication. This could be optical fiber, coaxial cable, or twisted-pair copper wire. Twisted-pair copper wire is widely used in Ethernet networks [3].

2.2 Network Security

There are many reasons for people to connect computers together into a network. Some examples of internet applications include: banking, where someone may pay bills or set up investments; entertainment, where people may enjoy multiplayer online games or streaming media; and sharing information, by sending email or surfing the world-wide web. To accomplish these tasks, personal information often needs to be shared and transmitted over a network. Since personal information can be used for financial gain, some people try to gather this information and either sell it or use it to make money. The internet, by connecting different networks together, can provide a means for unscrupulous people to search for and steal valuable personal information. Network security measures must be taken to protect this data, ensuring only authorized individuals can access it.

2.2.1 Common Network Attacks

Computers execute instructions. These instructions can be pre-defined and organized into computer programs, or software, that tells the computer what to do in certain circumstances. Unfortunately, the people writing computer software can not consider all possible circumstances under which a computer may be executing instructions. These overlooked conditions can leave vulnerabilities which someone may use to gain unauthorized access to a computer [9]. Software containing overlooked or unintentional errors can lead to someone gaining remote access to a computer over the internet, especially if the erroneous software is a networked application. Before software can be exploited, a remote attacker must first find a target for his attack and then gain information about the system which the software is running on.

Many attempts at computer intrusion begin with port scanning [10]. Port scanning is the process of sending out packets to either a specific IP address or an IP address range and attempting to connect to specific UDP or TCP ports. Since most networked programs accept connections only on specific port numbers, any attempts to connect that succeed identify both that a host is connected to the network and that it is running a specific application. Even connection attempts that fail can identify a host as being connected to the network if the host explicitly denies the connection. Once an attacker knows a host IP address and some services that are running on that host, they may begin trying to gain access to the system with usernames and passwords, or by using any known vulnerabilities in the code of the running network application.

Another way to gain information from a network is by network sniffing [9]. In

Ethernet networks, recall that packets are sent to all hosts on the local network and each Ethernet adapter is trusted to only look at packets destined for themselves. Unfortunately, it is easy to get an adapter to look at all packets being sent to it, whether the packets are destined for that adapter or not. This can allow people to not only gain network information such as IP addresses, but even usernames and passwords used to access network services. For example, when someone signs into an email server, the packets that leave their computer with the private username and password in them are first sent to all hosts on the user's local network. Anyone sniffing packets on that local network could then recover the private username and password. For this reason, some email servers encrypt all data including usernames and passwords.

Some network attacks are designed to gain information by attempting to trick a network host into communicating with a different host than was intended. This type of attack can be implemented in a number of ways, but they all rely on manipulating the networking protocols. Spoofing network packets is the act of sending packets out from a host, but changing the IP and/or MAC address to make it appear that the packets were sent from a host other than the actual sender [9]. This type of attack usually needs to be used in conjunction with an attack against a specific service or networked application at one of the higher layers in the network stack. An example of this type of attack is ARP cache poisoning.

As mentioned earlier, ARP is used to associate IP and MAC addresses. When an Ethernet adapter wants to transmit packets to a specific host, it needs to know the MAC address of the destination host. The source host sends out an ARP request, which is a request for the MAC address associated with the destination IP address. The ARP reply contains the association of a MAC address with an IP address. Once the source host has the destination IP and MAC addresses, it can send packets out onto the network to the destination host. However, when ARP was created, it was assumed that ARP replies would only come as responses to ARP requests. Unfortunately, a spoofed ARP reply packet will be accepted by hosts as a response to a request. A packet could be sent out from a host that associates its own MAC address with the IP address of a different host. In this case, all traffic destined for the second host would be delivered to the first host.

Not all network attacks seek to steal data or services. Some are designed to deny use of network resources to legitimate users. These types of attack are aptly named denial of service attacks [9]. There are two main types of denial of service attacks, those that seek to flood services and those that seek to crash services. Flooding a service refers to sending so much traffic to a specific host on a specific port that the host is unable to answer service requests. If enough traffic is sent to a victim host, it will be overloaded and unable to respond to even authorized service requests. Crashing a service refers to using vulnerabilities in the way the service software was written to halt execution of the software. For networking services, this is usually accomplished by sending packets that are malformed or have invalid data in them to the victim host [9]. If the service software doesn't know how to deal with the malformed packets, it may cause the program to halt execution, which also stops the service from being available.

2.2.2 Security Strategies

There are many strategies that have been developed to try to prevent the above attacks from succeeding [11]. The simplest, and least effective, is often called "security through obscurity" and doesn't implement any security measures at all. Security through obscurity relies on believing that a host would not be attacked because no one knows about it or because it is assumed no information of interest is on the host. Since it is incredibly easy to gain information about hosts on a network, obscurity doesn't last long if someone is probing a network for weak points. The idea that a host is of little interest no longer holds true either, as many attackers will gain access to unprotected systems and use them to attack more interesting and valuable hosts.

A better, and more widely adopted, security strategy is host security [11]. Host security means that each individual host on a network has security measures in place to protect them from attacks. This could include additional hardware or software added to the host machine that prevents or counters the attacker's methods. Host security can work well on individual systems, however it can become quite complex when trying to secure a large network with many connected hosts. Each host requires individual attention because different hosts will have different software and services running on them. As discussed, different pieces of software will have different inherent vulnerabilities in them. Even if all machines on the network were configured identically, it could still be very difficult and time-consuming to implement host security if the number of hosts is large.

A more efficient and effective method is to secure the underlying network [11], instead of securing individual hosts on the network. This network security strategy is composed of using: authentication to limit who is allowed to use network resources, encryption to prevent unauthorized individuals from gaining information by sniffing network packets, and filters that look at network traffic and remove packets that are malformed or come from untrusted hosts or networks. Filtering network packets as they travel through the network is considered key to securing the network infrastructure. Network packet filters are relatively easy to set up and maintain, they scale to support large networks, and they remain transparent to network users [12]. The term “firewall” is a common name for a network packet filter being used to restrict communication between two or more networks.

2.2.3 Packet Filtering

Packet filtering is the process of filtering network traffic based on various rules. By default, the packet filter can either allow or deny all traffic, meaning the rules specify conditions under which specific packets are allowed onto the network or are dropped, respectively. The most secure method is to deny all packets that try to come onto the network and only allow specific packets to pass when they match certain filter rules. The filter compares the data in individual packets with a set of rules that defines what to do when there is a match between a rule and a packet. The data used to define filtering rules is primarily based on the information used to implement the various network protocols, such as TCP port numbers, IP addresses or MAC addresses. The consequences of a rule match are often to drop the matching packet, though it will depend on the default filter conditions. For example, a filter could allow all traffic by default and have rules looking for packets containing matching source MAC or IP addresses with specific transport layer port numbers. By dropping packets that match this rule, a network could prevent specific service requests coming from a specific host. Packet filtering requires a tradeoff between the number of memory accesses needed to implement the filter and the maximum achievable network transfer speed [13]. There is some overhead as the filter rules need to be compared to the network traffic and each memory access takes time to complete. This overhead needs to be minimized in order to achieve a desired network transmission rate and keep the network filter transparent to the end users. Several algorithms exist to minimize this overhead for software-based packet filters, but great speed gains can be made by selecting fast memory, or by using memory that has been adapted for pattern matching, to reduce filtering time.

2.3 Content-Addressable Memory

Content-addressable memory (CAM) is similar to random access memory (RAM) except logic comparison circuitry is associated with each bit of storage. This logic comparison circuitry serves to compare all data storage locations in memory at the same time when a search is being conducted. Random access memory associates a piece of data with an address. When an address is provided to RAM, the data at that location is returned. To search for a specific piece of data in RAM requires one to check the data at each address sequentially. This means that the time it takes to search through RAM is dependent upon the number of addressable locations. Content-addressable memory, on the other hand, accepts a piece of data as input and returns the address where that data is located in memory. It searches all data locations at the same time and can identify the location of the data in one clock cycle. This makes CAM an excellent tool for pattern matching applications such as the comparison between internet packet data and pre-defined packet filter rules [14]. Content-addressable memory has become popular with hardware firewall manufacturers because it offers massive parallelism.

Chapter 3

Embedded Systems

This chapter focusses on the components and design of embedded systems. After discussing the general components that make up an embedded system and the tools used to create them, the specific tools and components used in designing the dynamic silicon firewall embedded system will be presented.

3.1 Embedded System Overview

An embedded system is a computer system that has been designed to perform specific tasks, in contrast to a personal computer which is designed to perform general tasks. Also, this specific computer system has been incorporated into a larger system or product [15]. Many consumer electronics products are embedded systems, such as digital music players, personal digital assistants, or cellular phones. Since embedded systems perform specific tasks, the required hardware is customized to the specific application. By selecting only the hardware needed to perform the required task, embedded systems efficiently use hardware and are therefore economical. Any software that may be running in an embedded system is called firmware [16]. Firmware would also be customized to the specific task being performed, and to the limited hardware chosen for the system. The hardware and software in combination are often referred to as an embedded device.

3.1.1 Embedded System Components

The hardware components that make up an embedded system are chosen depending on the purpose of the embedded device, but they are drawn from the same types of hardware that general computers use [15, 17]. Embedded systems may have a variety of hardware devices, including: processors, volatile and non-volatile storage, displays, and other input/output peripherals.

The hardware controlling the entire system may consist of a simple logic circuit, or it may be more complex, such as a microcontroller or microprocessor. If a microprocessor is required, then it will likely be executing some sort of firmware. Volatile or non-volatile memory may be used to store data in the system. This may be flash-memory for storage of firmware or other non-volatile information, or simple random access memory used for temporary storage. If user input is required, simple buttons may suffice, however, some embedded systems use more traditional input devices such as a keyboard and mouse. In short, embedded computer systems can draw on hardware available for general computing systems, but are tailored for the specific task they need to accomplish.

For embedded devices that contain a microprocessor and software, the software needs to be compiled to run on the specific hardware in the system. This may be done using cross compilers [16], where the firmware is compiled on a general purpose computer, then transferred into the embedded system. Some embedded devices run an embedded operating system that bridges the gap between hardware and software. These embedded operating systems are also called real-time operating systems [15, 16], as they can change program execution in real-time, depending on external inputs. For embedded systems without an operating system, the firmware needs to include the routines used to access each peripheral being used in the system.

3.2 System on a Programmable Chip

The components used in an embedded system draw from the same types of hardware as a general computing system, but they are chosen so that the system is efficient and economical to produce. While many embedded systems will use individual, discretely packaged components that are combined together on a printed circuit board, there is another alternative. Many embedded systems are defined in software and implemented on a programmable logic device (PLD), such as a field-programmable gate array (FPGA). This type of embedded system is called a system on a programmable chip (SOPC) [15].

3.2.1 SOPC Design

Systems on a programmable chip contain components that are defined using a hardware description language (HDL) such as VHDL [18] or Verilog [19]. Defining a system using an HDL offers many benefits. First, components can be designed once and re-used in different systems. Second, components can be purchased as intellectual property and used in system design. Third, the entire system can be extensively tested through simulations before a hardware version is put to market. Finally, the entire system can be tested in hardware on an FPGA and then transferred into an application-specific integrated circuit (ASIC) if desired. Also, since the system is defined with software modules which are then translated into logic on a PLD, any custom logic that is needed in addition to standard components can be easily added in a custom software module.

3.2.2 Altera® SOPC Design Software

The Altera® corporation specializes in SOPC solutions, including PLDs, associated software tools and intellectual property software blocks. The primary software package from Altera® is the Quartus® II design software [20]. It contains tools for all aspects of the design cycle. Designs can be created in block diagram form, with blocks representing the individual HDL modules. The design as a whole is then synthesized as each HDL module is compiled. The compiled project is then fitted to the individual PLD and the result can then be used to program the logic device being used. Tools for design analysis and verification are also included in Quartus® II. Timing requirements and logic functionality can be analyzed through software simulations or they can be investigated on a PLD as the design operates.

For embedded systems, Quartus® II also comes with a tool called SOPC Builder

[20]. SOPC Builder integrates the various components required for an embedded system. Altera® provides many HDL modules, as commercially available intellectual property, that work with SOPC Builder. This includes memories, communication interfaces, and even a configurable soft-core processor. SOPC Builder integrates the processor, busses and peripherals into a memory-mapped system. Firmware can then be written for the processor to access and manipulate the peripherals as needed. Various software libraries and application examples are also provided by Altera® for firmware development.

3.3 Dynamic Silicon Firewall Embedded System

The dynamic silicon firewall is an embedded system on a programmable chip. The decision to implement the design on a PLD allowed the use of pre-written intellectual property components, the ability to test the system using simulation, and the capability to easily transition the design to an ASIC, if desired. The network filter was to be designed in hardware as a module of custom logic, which lends itself well to PLDs and the use of an HDL. As an HDL module, the filter was relatively easy to test and modify as needed throughout the design cycle.

3.3.1 Dynamic Silicon Firewall Testbed

The dynamic silicon firewall was developed and tested on an FPGA development kit from Altera® [21]. This kit includes a number of features and peripherals that work in conjunction with SOPC Builder and Quartus® II. The FPGA on the board is the Stratix® EP1S25F1020C5 device, a general purpose FPGA featuring 25,660 logic elements, 1,994,576 bits of RAM, 10 digital signal processing blocks and 6 phase locked loops. The board comes with 33MHz and 100MHz oscillators, though other clock speeds can be synthesized using HDL modules. The onboard memory includes 256MB of dual data rate RAM and 8MB of flash memory. The hardware board comes in the form of a Peripheral Component Interconnect (PCI) board supporting both the 32 and 64 bit PCI bus configurations. There are expansion ports on the

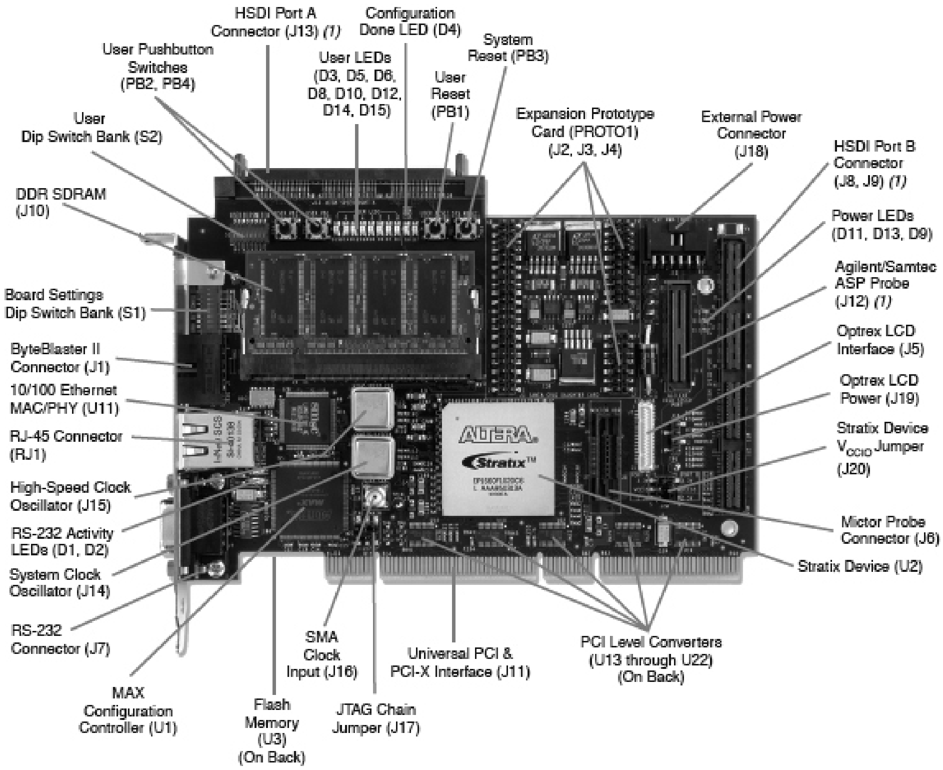

HSDI Port A Connector (J13) (1)
Configuration Done LED (D4)
System Reset (PB3)
User Pushbutton Switches (PB2, PB4)
User LEDs (D3, D5, D6, D8, D10, D12, D14, D15)
User Reset (PB1)
User Dip Switch Bank (S2)
Expansion Prototype Card (PROTO1) (J2, J3, J4)
External Power Connector (J18)
HSDI Port B Connector (J8, J9) (1)
DDR SDRAM (J10)
Power LEDs (D11, D13, D9)
Agilent/Samtec ASP Probe (J12) (1)
Board Settings Dip Switch Bank (S1)
ByteBlaster II Connector (J1)
Optrex LCD Interface (J5)
10/100 Ethernet MAC/PHY (U11)
Optrex LCD Power (J19)
RJ-45 Connector (RJ1)
Stratix Device V_{CCIO} Jumper (J20)
High-Speed Clock Oscillator (J15)
RS-232 Activity LEDs (D1, D2)
Mictor Probe Connector (J6)
System Clock Oscillator (J14)
Stratix Device (U2)
RS-232 Connector (J7)
SMA Clock Input (J16)
Universal PCI & PCI-X Interface (J11)
PCI Level Converters (U13 through U22) (On Back)
MAX Configuration Controller (U1)
Flash Memory (U3) (On Back)
JTAG Chain Jumper (J17)
ALTERA
Stratix™

Figure 3.1: Altera® Stratix® PCI Development Kit

board that support daughter-cards which can be custom made, or purchased from Altera®. Various communication interfaces are also included, such as serial, parallel and Ethernet ports. The hardware board is shown in figure 3.1.

In addition to the hardware, various software components are also used in the testbed. The firmware being used to test the system is based on an example web server program from Altera®. The web server accepts HTTP requests and serves simple web pages. The dynamic silicon firewall was tested by monitoring which specific hosts were allowed or denied access to the web server. The web server relies on another firmware component from Altera®. The Plugs Ethernet Library [22] is a set of routines, provided by Altera®, used to configure Ethernet devices, and to transmit and receive data packets. Plugs provides the use of the raw Ethernet protocol at the link layer, ARP and IP at the network layer and both TCP and UDP at the transport layer. For the example web server application, Ethernet, ARP, IP and TCP are used from the Plugs library.

Chapter 4

Silicon Firewall Hardware Components

In this chapter, the hardware design of the dynamic silicon firewall is presented. The core hardware components of the design: the Ethernet chip, the CAM, the soft-core processor, and the firewall HDL module are discussed in detail.

4.1 Ethernet Device

The Ethernet chip being used in this design is the LAN91C111 from Standard Microsystems corporation. It is a mixed-signal analog/digital device that implements the Ethernet protocol at transmission rates of 10 or 100 Mbps in either half or full duplex mode. It has 8kB of buffer memory, which is used for both transmission and reception. The registers used to configure and utilize the MAC implementation on the chip are mapped into four banks, each bank holding eight of the 16-bit registers.

4.1.1 Configuration

The LAN91C111 is configured by accessing various registers in the device. In this application, auto-negotiation is enabled so that the Ethernet device can achieve the highest performance when it is connected to various networks with differing speeds. Auto-negotiation on the LAN91C111 can result in 10 or 100 Mbps speeds, in either full or half duplex mode. Three interrupt sources are also enabled during configuration. Reception and transmission completions are denoted by an interrupt, as are Ethernet protocol handler (EPH) errors. EPH errors can occur if the physical medium is disconnected or if a fatal transmission error occurs, such as if collisions occur or a transmit buffer under-run occurs.

4.1.2 Transmission

The first step in Ethernet transmission for the LAN91C111 is to allocate a portion of the onboard 8kB buffer memory to hold the packet being transmitted. The packet is then loaded into the allocated memory and assigned a packet number, which is used to enqueue the packet for transmission. Once the packet number has been enqueued, the packet will be keyed onto the transmission medium, assuming that transmissions have been enabled. When the transmission is completed, the first word in the allocated buffer memory is written with a status word. This status word contains important information on the most recently transmitted Ethernet frame, noting if an error occurred or if the transmission succeeded. An interrupt occurs in either case, however the transmission sequence stops in the case of a failure. In addition, the packet number is reported if a failure occurred, whereas the number is moved to a completed transmission queue upon success. When handling the inevitable transmission interrupt, the status word needs to be examined to determine success or failure. If the transmission succeeded, then acknowledging the interrupt is all that is needed to remove the completed packet number from the completed transmission queue, and to continue further transmissions. If the transmission failed, the type of error can be determined from the status register. To retransmit a failed packet, the packet number can be re-enqueued for transmission and it will be sent out again once transmissions have been re-enabled.

4.1.3 Reception

Assuming the LAN91C111 has been enabled to receive packets, the device will first request a portion of buffer memory to hold an incoming packet. When a packet arrives on the interface, it is assigned a packet number and the incoming data is written to the allocated segment of memory. If an overrun of memory occurs, the packet is dropped and the allocated memory is released. When reception has finished, the first word in the allocated buffer memory is written with a status word. This status word contains information on any errors that may have occurred during reception, and denotes if the packet was longer or shorter than a standard Ethernet frame size. A cyclic redundancy check (CRC) is performed on the received frame, and if the result shows that no errors are present in the received data, then the associated packet number is written to a reception queue and an interrupt occurs. If the CRC shows that errors exist in the received data, the packet is dropped, the memory is freed, and no interrupt occurs. When a reception has successfully occurred, the data can be retrieved from the reception area of buffer memory by using the packet numbers in the reception queue. As each packet in the reception queue is processed, commands to remove the packet number from the reception queue and to release the allocated buffer memory must be issued.

4.2 Content-Addressable Memory

The content-addressable memory (CAM) used in the dynamic silicon firewall holds 32 32-bit words and was created using dual-port random access memory blocks [23]. Since the system is synthesized on an Altera® Stratix® FPGA, the Altera® altsyncram megafunction was used for the dual-port memory blocks. The altsyncram megafunction supports single and dual-port configurations for both random-access and read-only memory [24]. Each port of the dual-port RAM is configured independently to provide the separate write and read functionalities of the CAM.

4.2.1 CAM Writes

Write operations are performed by storing a one-hot encoded value. For example, take a 16-byte sized CAM. Each byte would be translated into a 256-bit one-hot word. With 16 of these one-hot encoded bytes, the total storage needed is 4,096 bits. Each write operation toggles one of these bits. The write port of the dual-port RAM has a single bit as the data input and 12 bits for the address input. These 12 bits are used to access the 4,096 individual bits. The 12-bit address going to the RAM is made up of the data and address inputs to the CAM. The CAM data byte and the four bits used to address the 16 CAM locations are concatenated together

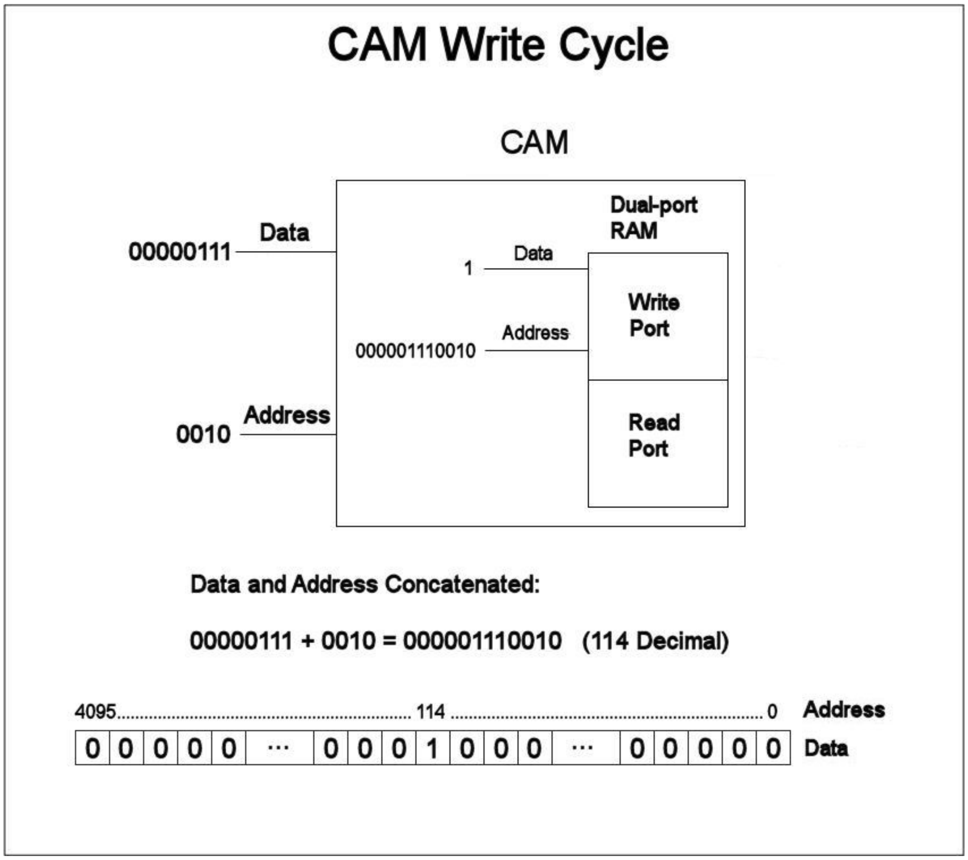


Figure 4.1: CAM made from dual-port RAM: write functionality

to provide the 12-bit address going to the RAM, with the CAM data byte in the most significant position. Figure 4.1 illustrates an example write operation for the 16-byte sized CAM. In this example, it is desired that a decimal data value of seven is written to address number two in the CAM. The concatenation shows that the address presented to the RAM is 114, meaning a one is written to the 114th bit of the 4,096-bit storage area. This example will be furthered by looking at the CAM read functionality.

4.2.2 CAM Reads

The port of the dual-port RAM being used for CAM reads, considering our example 16-byte CAM, has a byte-wide address as input and a 16-bit data output. This equates the 4,096 bits of storage into 256 words, each 16 bits in length. The 16 bits of output data represent the 16 storage locations in the CAM. The byte-wide RAM

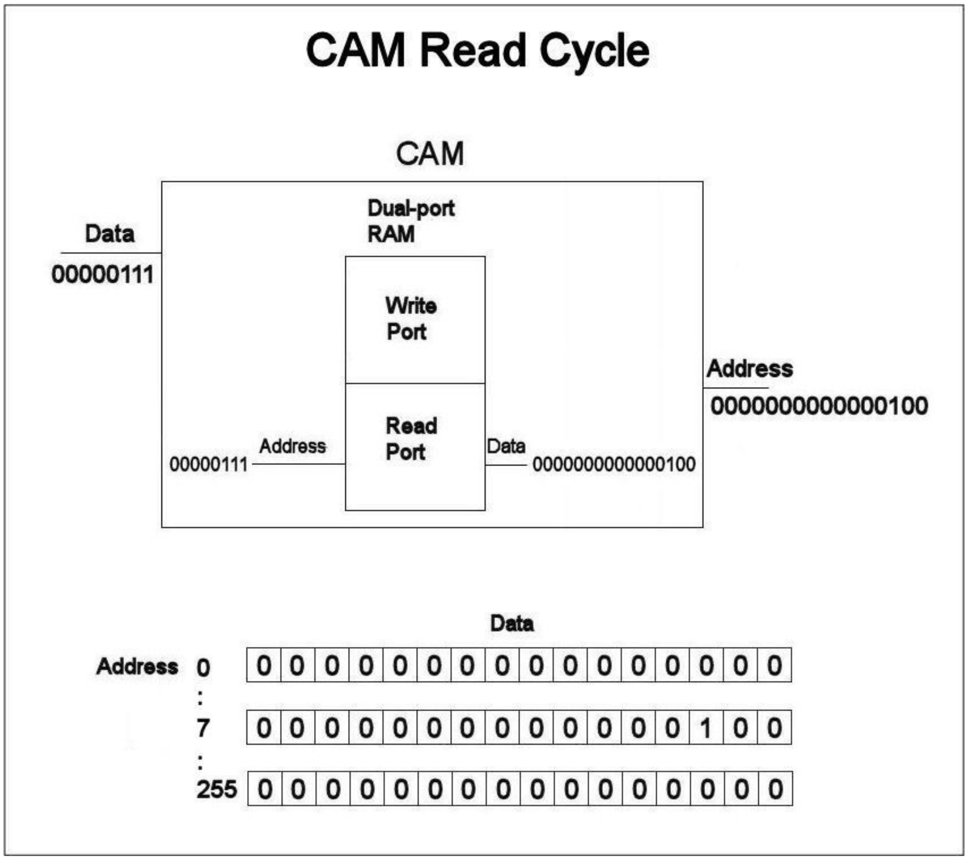


Figure 4.2: CAM made from dual-port RAM: read functionality

address is connected to the CAM input data byte. When a CAM search is performed, the data byte used as input results in a 16-bit one-hot output, effectively searching all 16 storage locations in the CAM. A value of one at any bit in the output denotes that the input data byte was previously written to one of the 16 storage locations in the CAM. Figure 4.2 shows an example read operation, assuming that a data value of seven was previously written to address number two in the CAM. The input byte has the value of seven, and the 16-bit output has the value one at the third bit location. This represents address number two, counting up from zero. Looking back at the write example, we can see this result comes about because a one was written to the 114th bit of the 4,096-bit storage area. When this storage area is configured as 256 16-bit words, the 114th bit is the third bit of the seventh word, representing a value of seven written to address number two.

4.3 Nios® Soft-core Processor

The Nios® central processing unit (CPU) is a soft-core, pipelined, general purpose, reduced instruction set microprocessor provided by Altera® for use in PLDs in either a 16 or 32-bit architecture form [25]. Both variants use a 16-bit instruction set, which reduces instruction-memory bandwidth requirements. Since the Nios® core is configurable, it is possible to add custom logic directly into the arithmetic logic unit and to add custom instructions to the Nios® instruction set. The Nios® CPU has a five-stage pipeline with separate data and instruction busses in accordance with the Harvard computer architecture. Each bus follows the Avalon® bus specification, which is an Altera® bus specified for use with the Quartus® II and SOPC Builder design tools. The instruction bus master only reads instructions from memory and provides them to the CPU for execution, it never writes any information to memory. It can use an optional cache memory if the instruction memory being used has slow access speeds. The data bus master is a 16-bit wide bus when the 16-bit Nios® architecture is used, and is 32 bits wide for the 32-bit architecture. The data master is used for three purposes. It reads data from memory when the CPU is executing a load instruction, it writes data to memory when the CPU is executing a store instruction, and it reads interrupt vectors from the interrupt vector table when an interrupt occurs.

Altera® provides a number of soft-core peripherals for use with the Nios® CPU. These peripherals exist as HDL modules which integrate with the Altera® design tools and allow rapid development of complete systems, since the peripherals operate on the Avalon® bus and come with software drivers for use with Nios® software programs. Two Nios® peripherals are used in the dynamic silicon firewall project.

4.3.1 Nios® Peripherals

The timer peripheral is essential, as it provides the dynamic aspect of the project. A timer is used to determine when the IP addresses, used as filtering rules, should be purged so that packets received from those addresses are no longer trusted. The Nios® timer peripheral is a simple interval timer that is configured and controlled through six 16-bit registers [26]. Two of these registers are used to hold the value of the timer period. Another two of these registers can be used to determine the current value of the timer while it is running. This leaves two registers for status information and timer control. The status register simply denotes whether the timer is currently running or if it has reached the period value stored in the timer period registers. The control register contains four bits of interest. Two bits are used to start or stop the timer. The start and stop bits are event driven, not value driven, meaning the start or stop function occurs when written with a one. Writing a zero to either bit has no effect. The other two bits in the control register are used to enable timer interrupts and to determine how the timer reacts at the end of each period. When the interrupt bit is set to one, the timer will generate an interrupt at the end of every period. Setting this bit to zero ensures no timer interrupts occur. The continuous bit, when set to one, causes the timer to reload and restart at the end of every period. When the continuous bit is zero, the timer reloads, but does not restart at the end of the period. Use of the start bit is then required to restart the timer. For the dynamic silicon firewall project, the timer generates an interrupt every two minutes. It is in the interrupt service routine that the purging of IP addresses occurs. The timer operates in continuous mode, meaning the timer will simply continue to generate interrupts every two minutes for the periodic removal of filtering rules.

The parallel input/output (PIO) peripheral is used to provide a memory-mapped interface between software running on the Nios® CPU and user-created logic that is operating external to the system created using SOPC Builder [26]. The PIO port can be configured for input-only, output-only, or bi-directional accesses between the internal Avalon® bus and external logic. Data reads or writes occur through register accesses, as does configuration. The PIO can be configured to capture edge transitions and to generate edge or level-triggered interrupts, if desired. For the dynamic silicon firewall project, PIO peripherals are used to connect the custom

Verilog packet filter to the Nios® CPU.

4.4 Verilog Silicon Firewall Module

The system component that is performing the filtering operation on Ethernet packets is a finite state machine designed as a custom Verilog module. A state diagram for this finite state machine is shown in figure 4.3. From this diagram, it can be seen that there are eight states in total, with a number of labeled transitions. A more detailed state diagram can be seen in figure 4.4. This diagram includes some state information regarding the last and next states, as well as what functionality is performed in each state.

The finite state machine is implemented using nested case statements. In the outer case statement, each case represents one of the finite states, while the inner case statement has a case select parameter that is incremented on each system clock cycle to step through command execution and to determine the next state. This allows timing requirements to be met for any Ethernet chip instructions, since the system clock speed will always be known and any instructions being executed in the inner case statement can be ordered to execute in any multiple of clock cycles so that the LAN91C111 timing requirements are met.

4.4.1 State Machine Entry

The entry point to the state machine occurs when an interrupt is generated by the LAN91C111 Ethernet chip. The first state (Wait for Interrupt) simply waits for a new interrupt to occur, and then transitions to the second state (Read Interrupt Type). A diagram illustrating the flow of the second state is shown in figure 4.5. The second state begins by storing specific LAN91C111 registers, so that the Ethernet chip can be restored to pre-interrupt status after the interrupt has been processed. The next transition is then determined by finding out the cause of the interrupt, which may be due to reception, transmission, or errors. Ethernet is an unreliable transmission protocol, meaning that ensuring successful communication is relegated

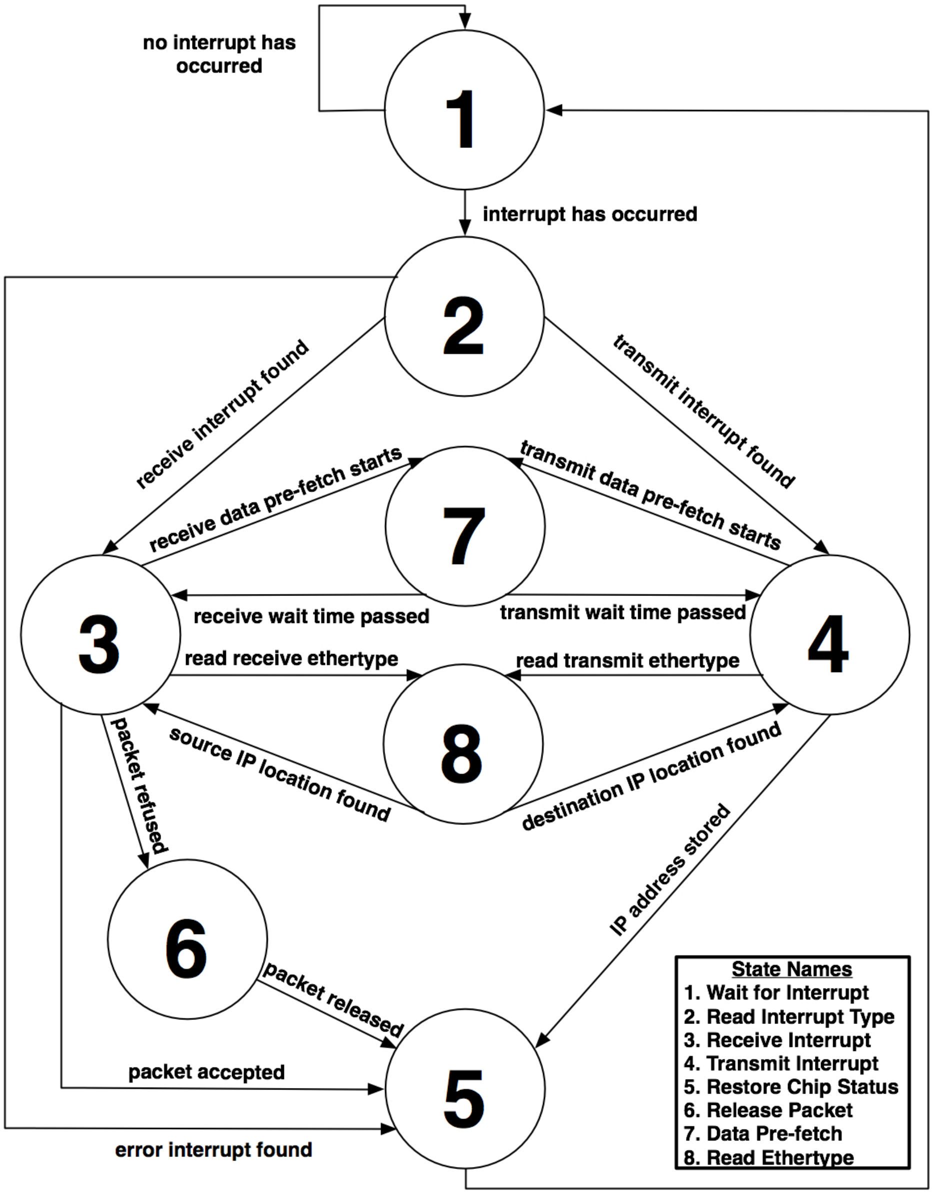
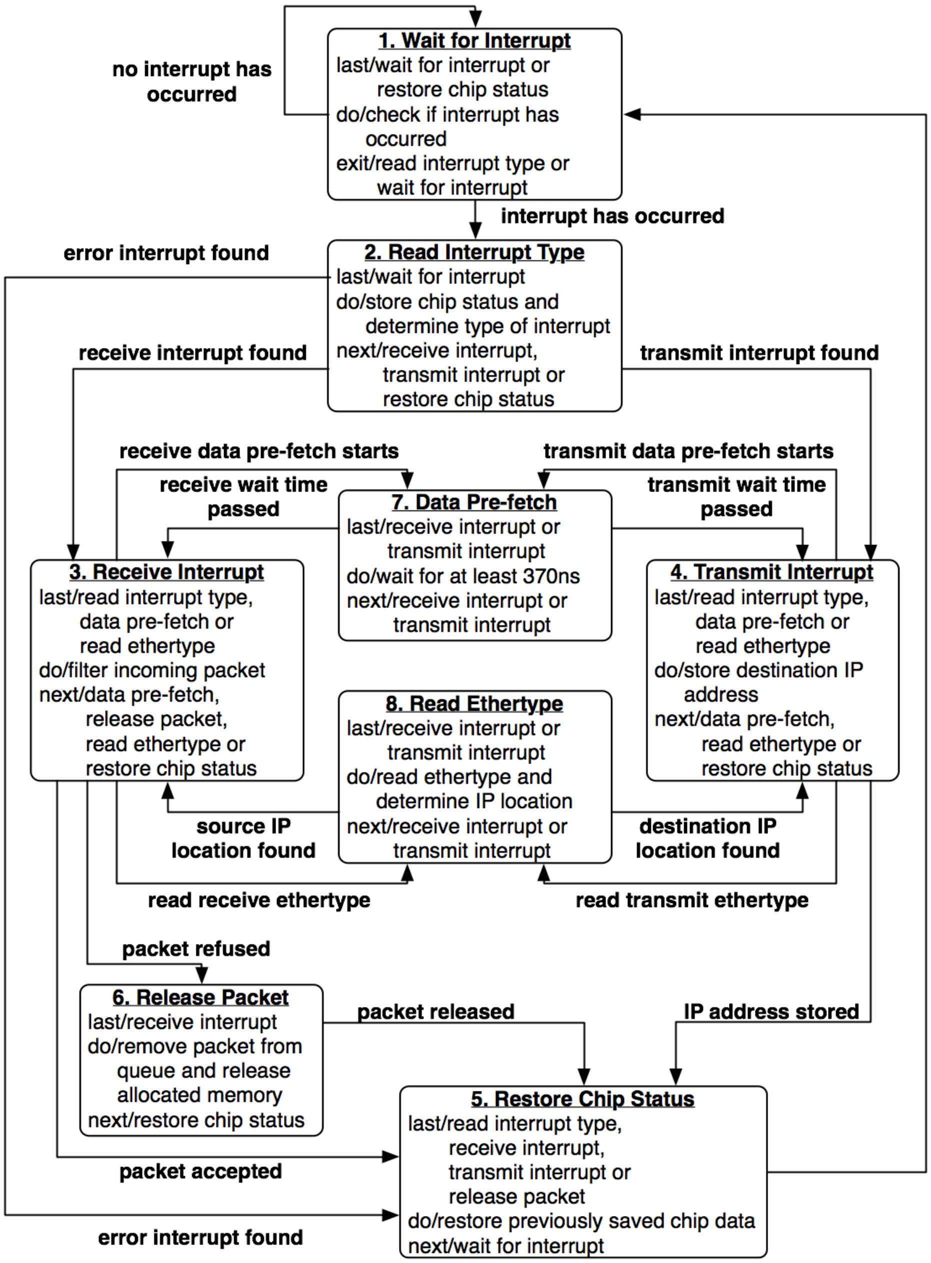


Figure 4.3: State Diagram for Silicon Firewall



1. Wait for Interrupt
last/wait for interrupt or restore chip status
do/check if interrupt has occurred
exit/read interrupt type or wait for interrupt
no interrupt has occurred
interrupt has occurred
2. Read Interrupt Type
last/wait for interrupt
do/store chip status and determine type of interrupt
next/receive interrupt, transmit interrupt or restore chip status
error interrupt found
receive interrupt found
transmit interrupt found
receive data pre-fetch starts
transmit data pre-fetch starts
receive wait time passed
transmit wait time passed
7. Data Pre-fetch
last/receive interrupt or transmit interrupt
do/wait for at least 370ns
next/receive interrupt or transmit interrupt
3. Receive Interrupt
last/read interrupt type, data pre-fetch or read ethertype
do/filter incoming packet
next/data pre-fetch, release packet, read ethertype or restore chip status
4. Transmit Interrupt
last/read interrupt type, data pre-fetch or read ethertype
do/store destination IP address
next/data pre-fetch, read ethertype or restore chip status
8. Read Ethertype
last/receive interrupt or transmit interrupt
do/read ethertype and determine IP location
next/receive interrupt or transmit interrupt
source IP location found
destination IP location found
read receive ethertype
read transmit ethertype
packet refused
6. Release Packet
last/receive interrupt
do/remove packet from queue and release allocated memory
next/restore chip status
packet released
IP address stored
5. Restore Chip Status
last/read interrupt type, receive interrupt, transmit interrupt or release packet
do/restore previously saved chip data
next/wait for interrupt
packet accepted
error interrupt found

Figure 4.4: Detailed State Diagram for Silicon Firewall

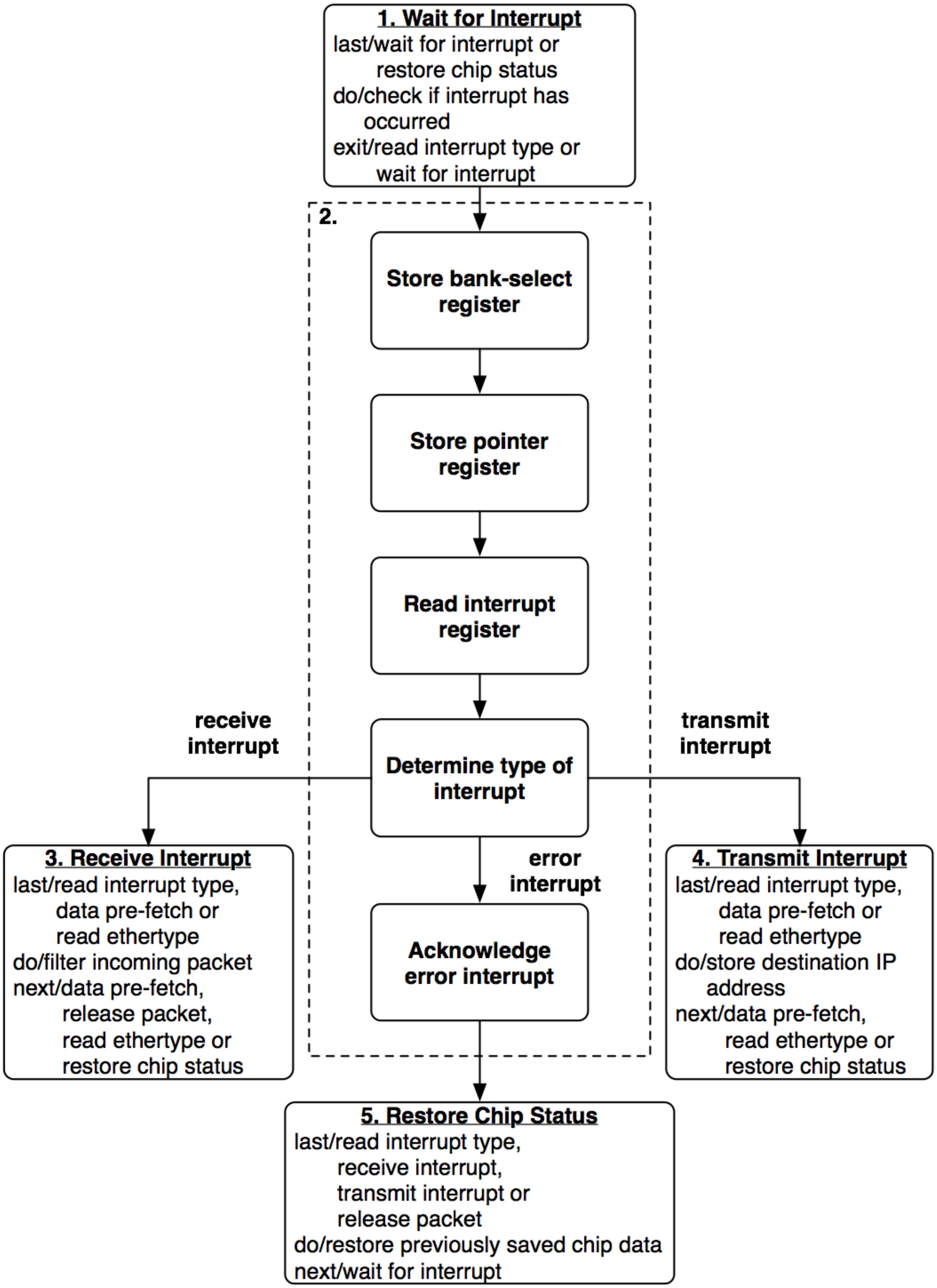

1. Wait for Interrupt
last/wait for interrupt or restore chip status
do/check if interrupt has occurred
exit/read interrupt type or wait for interrupt
2.
Store bank-select register
Store pointer register
Read interrupt register
Determine type of interrupt
receive interrupt
transmit interrupt
error interrupt
Acknowledge error interrupt
3. Receive Interrupt
last/read interrupt type, data pre-fetch or read ethertype
do/filter incoming packet
next/data pre-fetch, release packet, read ethertype or restore chip status
4. Transmit Interrupt
last/read interrupt type, data pre-fetch or read ethertype
do/store destination IP address
next/data pre-fetch, read ethertype or restore chip status
5. Restore Chip Status
last/read interrupt type, receive interrupt, transmit interrupt or release packet
do/restore previously saved chip data
next/wait for interrupt

Figure 4.5: Read Interrupt Type State Diagram

to the higher levels of the network stack, like the transport layer. Therefore, the error condition is the easiest to deal with, as it simply requires an acknowledgement of the interrupt and the restoration of the system to allow future transmissions and receptions. In this case, the second state transitions to the fifth state (Restore Chip Status).

4.4.2 Transmit Interrupt State

In the case of a transmission interrupt, the second state (Read Interrupt Type) transitions to the fourth state (Transmit Interrupt). A diagram of the Transmit Interrupt state is shown in figure 4.6. The purpose of the fourth state is to obtain and store the destination IP address from the last successfully transmitted Ethernet packet. First, the packet number of the last transmitted packet must be read from the transmission-completed queue and written to the packet number register. This makes the data of the last transmitted packet available at the transmit area in memory. After a data pointer is set to the location of the status word of the last transmitted packet, in transmit memory, a data pre-fetch must occur before reading occurs. Due to the way the LAN91C111 chip was used by Altera® on their development board, this pre-fetch requires a wait period of at least 370ns. This wait period is done in state seven (Data Pre-fetch), so a state transition to state seven occurs whenever a data pre-fetch is required. After the wait period has passed, a transition from the seventh state back to the previous state occurs. Once the data pre-fetch has occurred, the status word of the last transmitted packet can be read. This status word determines if the last transmission completed successfully. If it did not, and some error occurred, then the transmit interrupt is acknowledged and a transition to state five (Restore Chip Status) occurs. If the transmission succeeded, then the location of the destination IP address to store needs to be determined. The location of the IP address is different depending on if the network layer protocol of the packet in question is IP or ARP. The determination of the Ethertype and the corresponding IP address location is done in state eight (Read Ethertype). A transition to state eight occurs after it has been determined that the transmission succeeded, and a

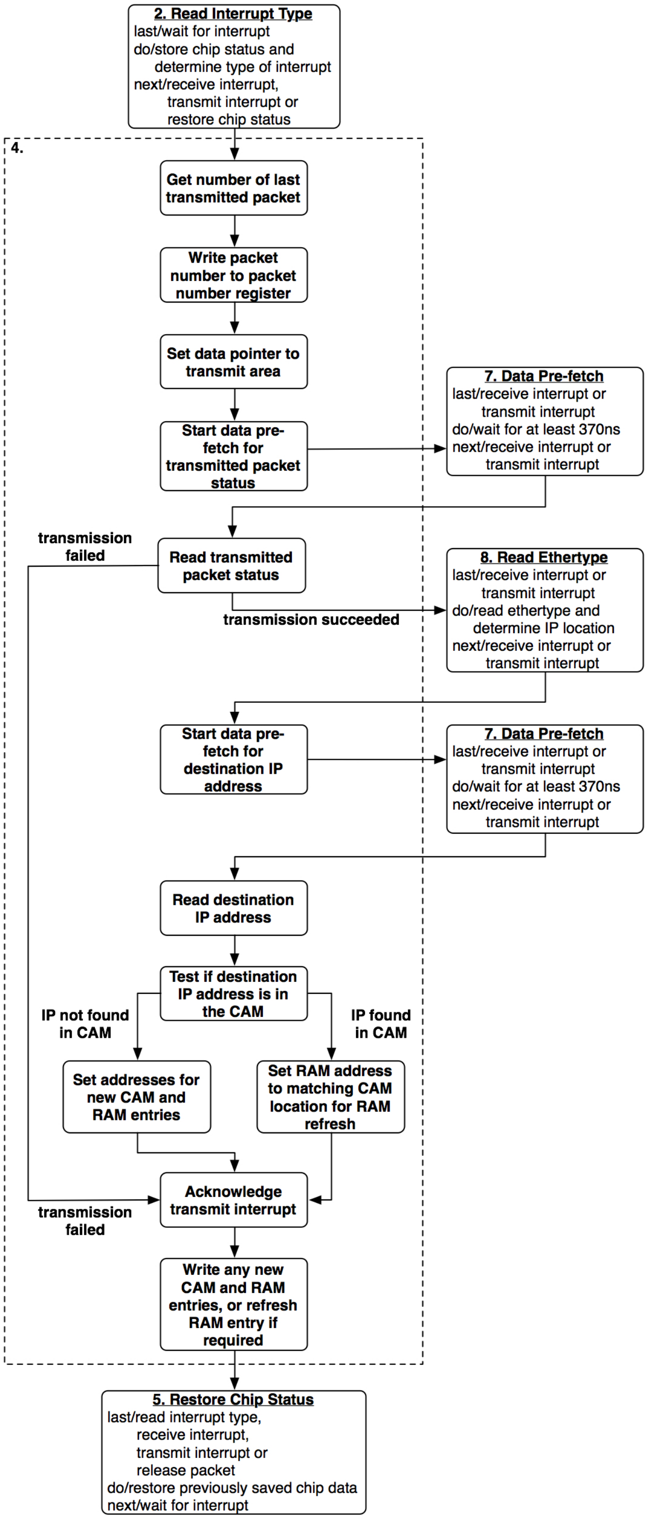

2. Read Interrupt Type
last/wait for interrupt
do/store chip status and
determine type of interrupt
next/receive interrupt,
transmit interrupt or
restore chip status
4.
Get number of last transmitted packet
Write packet number to packet number register
Set data pointer to transmit area
Start data pre-fetch for transmitted packet status
7. Data Pre-fetch
last/receive interrupt or
transmit interrupt
do/wait for at least 370ns
next/receive interrupt or
transmit interrupt
transmission failed
Read transmitted packet status
transmission succeeded
8. Read Ethertype
last/receive interrupt or
transmit interrupt
do/read ethertype and
determine IP location
next/receive interrupt or
transmit interrupt
Start data pre-fetch for destination IP address
7. Data Pre-fetch
last/receive interrupt or
transmit interrupt
do/wait for at least 370ns
next/receive interrupt or
transmit interrupt
Read destination IP address
Test if destination IP address is in the CAM
IP not found in CAM
IP found in CAM
Set addresses for new CAM and RAM entries
Set RAM address to matching CAM location for RAM refresh
Acknowledge transmit interrupt
transmission failed
Write any new CAM and RAM entries, or refresh RAM entry if required
5. Restore Chip Status
last/read interrupt type,
receive interrupt,
transmit interrupt or
release packet
do/restore previously saved chip data
next/wait for interrupt

Figure 4.6: Transmit Interrupt State Diagram

return to state four occurs after the IP address location has been determined. The data pointer is then pointed to the IP address location and another pre-fetch occurs. After this pre-fetch has finished, the destination IP can be read. This IP address is then tested to see if it is already in the CAM, in which case the previous entry simply needs to be refreshed. If the IP address is not in the CAM, then it is stored as one of the 32 CAM words, and the entry is marked as fresh. Entries are marked fresh or stale to provide the dynamic aspect of the project. A 32-bit RAM is used for this, where the 32 bits represent the 32 CAM addresses. When an entry in the CAM is written, the corresponding address in the RAM has its data value set to one. A value of one represents a fresh entry, while a zero represents a stale entry. Entries are only marked fresh by the Verilog module, and only marked stale by software running on the Nios® CPU. Stale entries are periodically removed from the CAM by software. After the destination IP has been stored and marked fresh, a transition to state five occurs, where the Ethernet chip is restored to the enable future receptions or transmissions, and where the system is restored to an initial state where it is ready to handle future interrupts.

4.4.3 Receive Interrupt State

In the case of a reception interrupt, the second state transitions to the third state (Receive Interrupt). A diagram of the Receive Interrupt state is shown in figure 4.7. The purpose of the third state is to obtain the source IP address from each incoming packet and to compare that address with those stored in the CAM. When matches are found between source IP and CAM entries, the packet is accepted. However, if no matching entry is found, then the packet is dropped from the reception queue. The first step in obtaining the source IP address is to check the status of the last received packet. The packet at the top of the reception queue is available at the receive area in memory. As in the case with transmitted packets, a data pointer is pointed to the location of the status word in the received packet, after which a data pre-fetch must occur. This requires a transition to state seven for a wait period, followed by a transition back to the previous state. Once the pre-fetch has finished,

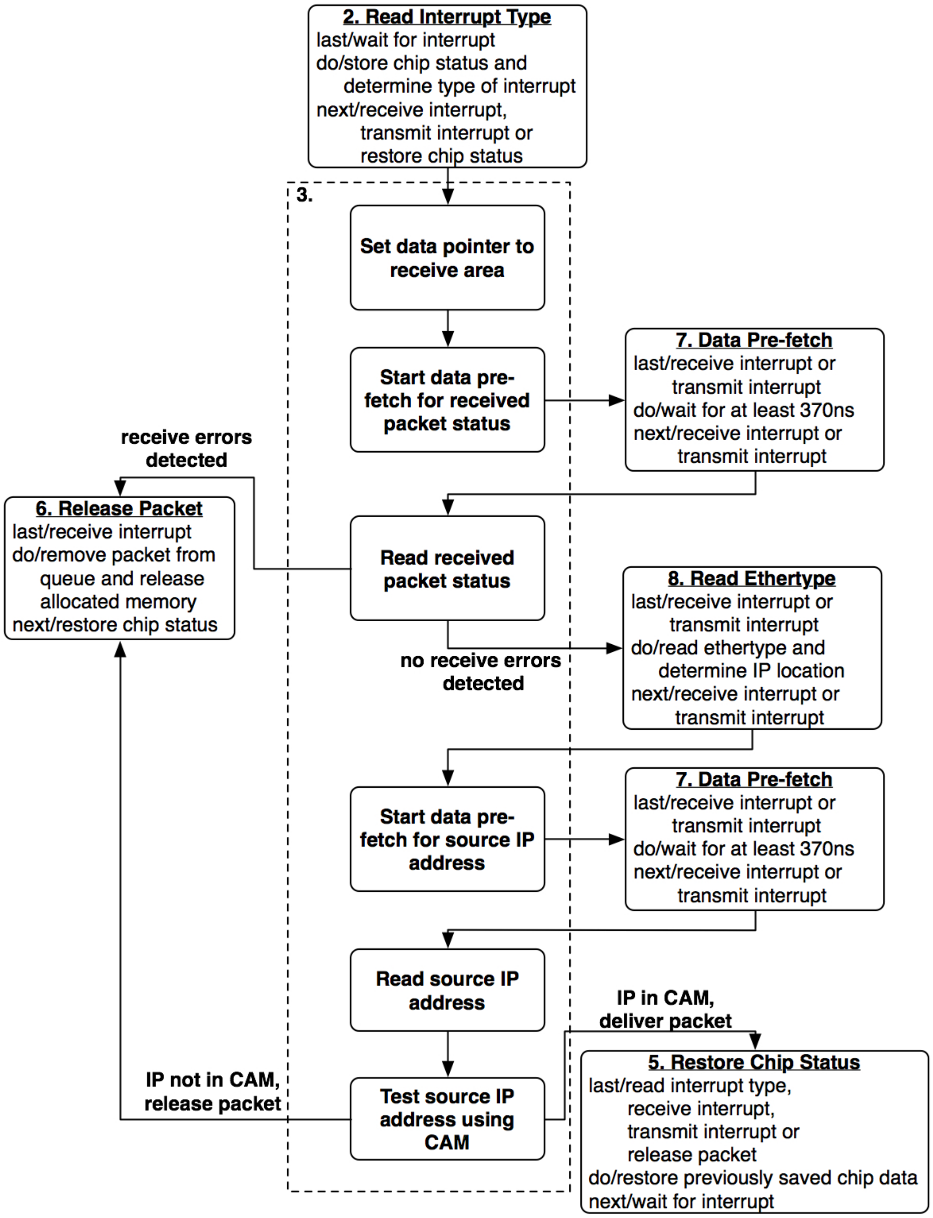


Figure 4.7: Receive Interrupt State Diagram

the status word is read and it is determined if any errors occurred during reception. If there were any reception errors, the packet needs to be dropped from the reception queue. This action is performed in state six (Release Packet), requiring a transition from state three to state six. If no errors occurred, then the Ethertype of the packet must be read, in order to determine the source IP address location. This occurs as in the transmit interrupt case. A transition to state eight occurs where the Ethertype is read and the source IP address location is determined. After that, the data pointer is pointed to the IP address location, and a transition to state seven occurs for a data pre-fetch, after which the IP address is read. This source IP address is provided to the CAM, where a search occurs. If the source IP address is found in the CAM, then the packet will not be filtered-out, and a transition to state five will occur, where the Ethernet chip is restored to enable future receptions and transmissions. A flag is stored, denoting whether any higher level entities waiting for incoming packets need to be notified that a successfully received packet is waiting to be read. In the case that the source IP address was not found in the CAM, the packet needs to be dropped from the reception queue. This means a transition will occur from state three to state six (Release Packet). A diagram of the Release Packet state can be seen in figure 4.8. In state six, a command that removes the packet at the top of the reception queue and releases the associated memory is given to the LAN91C111. The Ethernet chip is polled to determine when these actions have finished. Once the remove and release command has finished successfully, a transition to state five occurs, where the Ethernet chip is restored in order to receive and transmit future packets, and the system is restored to an initial state where it is ready to handle future interrupts.

4.4.4 Common States

A few of the system states are common to the way both the receive and transmit interrupts are handled. As mentioned, the Data Pre-fetch state is required in order to provide a delay while information is read from the packet memory in the LAN91C111. This is due to the way the Altera® daughtercard, containing the LAN91C111 chip,

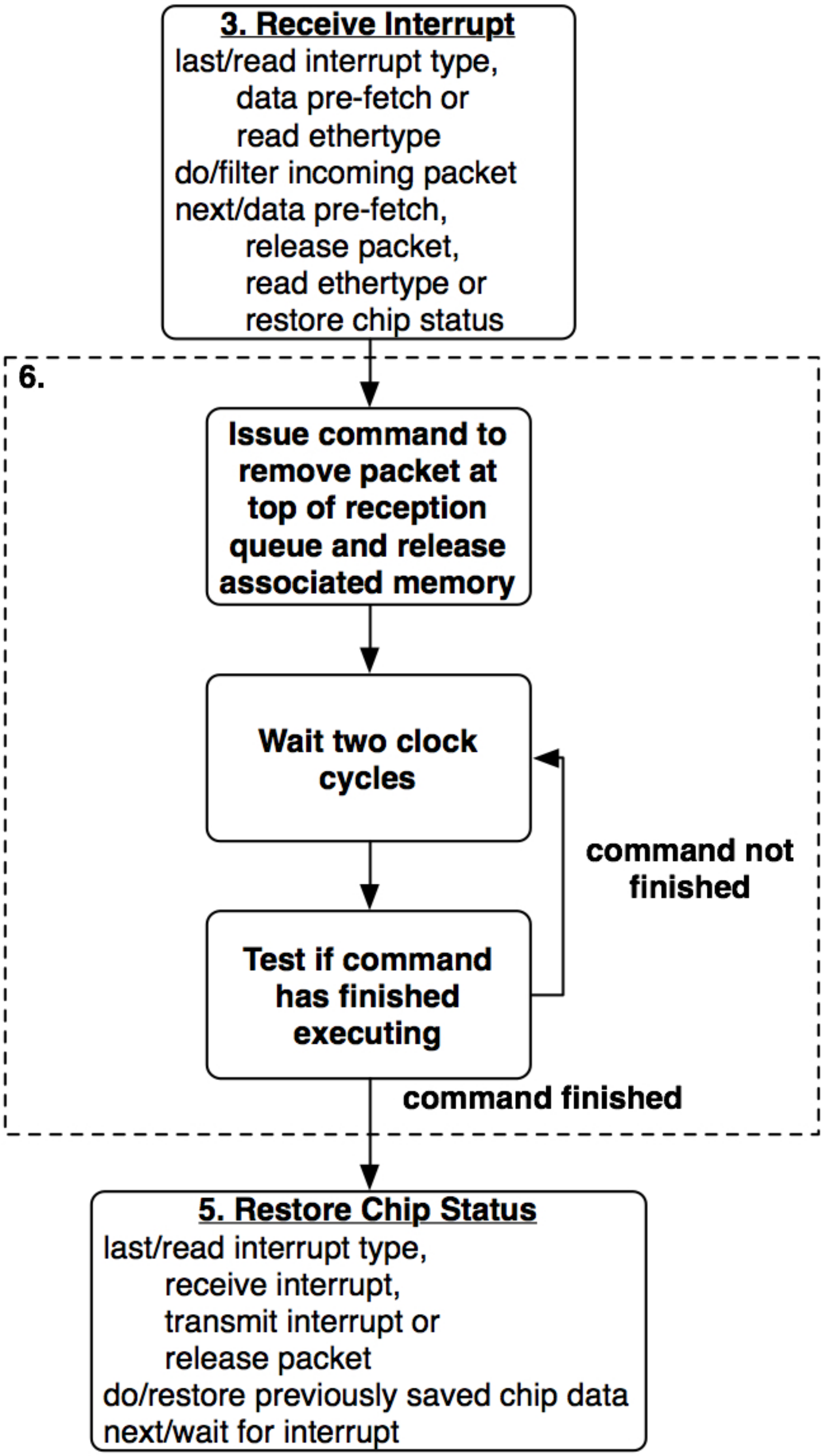


Figure 4.8: Release Packet State Diagram

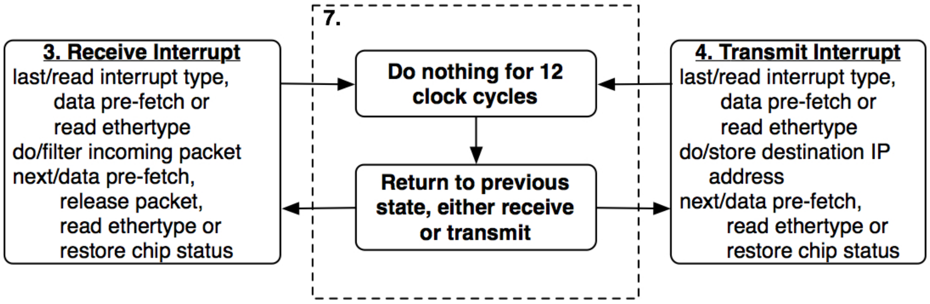

7.
3. Receive Interrupt
last/read interrupt type, data pre-fetch or read ethertype
do/filter incoming packet
next/data pre-fetch, release packet, read ethertype or restore chip status
Do nothing for 12 clock cycles
Return to previous state, either receive or transmit
4. Transmit Interrupt
last/read interrupt type, data pre-fetch or read ethertype
do/store destination IP address
next/data pre-fetch, read ethertype or restore chip status

Figure 4.9: Data Pre-fetch State Diagram

was designed. The Altera® development board communicates with the Ethernet daughtercard using an asynchronous bus. For asynchronous bus operation, a pin named ARDY (Asynchronous Ready) is provided on the Ethernet chip to denote when valid data has been delivered after a pre-fetch operation. However, there is no connection between the ARDY pin and the daughtercard in the Altera® implementation. The LAN91C111 datasheet says that a minimum wait time of 370ns should be allowed for the data register to fill, if the ARDY pin is not being used on the chip [27]. This wait period is done in state seven, where 12 clock cycles pass before returning to the previous state. However, 13 clock cycles pass in total, because an additional clock cycle is needed to transition into state seven. Since the system clock has a 30ns period, a total of 390ns pass each time a data pre-fetch occurs. This state is illustrated in figure 4.9.

Another commonly used state is the Read Ethertype state. It is in this state that IP address locations are determined. A diagram showing the Read Ethertype state is shown in figure 4.10. In the transmit interrupt case, the destination IP address location is of interest, while in the receive interrupt case, it is the source IP address location which is of interest. In addition to the differences in source and destination IP locations, the IP address location is different depending on the network layer protocol in the packet. The first step in determining the IP address location is to determine the network layer protocol in question. To do this, the previous state must be inspected to determine whether the data pointer needs to point into receive or transmit memory. Once this is determined, the data pointer is set to the location of the Ethertype and a data pre-fetch occurs. After the pre-fetch has occurred, the location of the IP address can be determined to be in one of four locations depending on if the packet was in receive or transmit memory, and if the Ethertype showed that the network protocol was either IP or ARP. The final step in this state is to return to the previous state with the IP address location stored.

Four of the states in the system end by transitioning to state five, Restore Chip. State five is illustrated in figure 4.11. This state is responsible for restoring the Ethernet chip to a status where it can transmit or receive future packets, and for restoring the system to an initial state where it is ready to handle future interrupts. State five is entered into after errors have been found in received or transmitted packets, or after an interrupt has been handled by the system. The first action in this state is to restore the registers that were stored in the Read Interrupt state to their pre-interrupt values. Next, if the flag was set that marks an incoming packet as one accepted through the firewall, any higher level entities waiting for incoming packets must be notified. In our test system, this would mean sending an interrupt to the Nios® processor to notify it of a waiting packet. Finally, all flags and variables used in the Verilog module must be re-initialized so that the system is ready to handle future Ethernet interrupts. Once this is completed, state five transitions back to state one, where the system awaits a new interrupt.

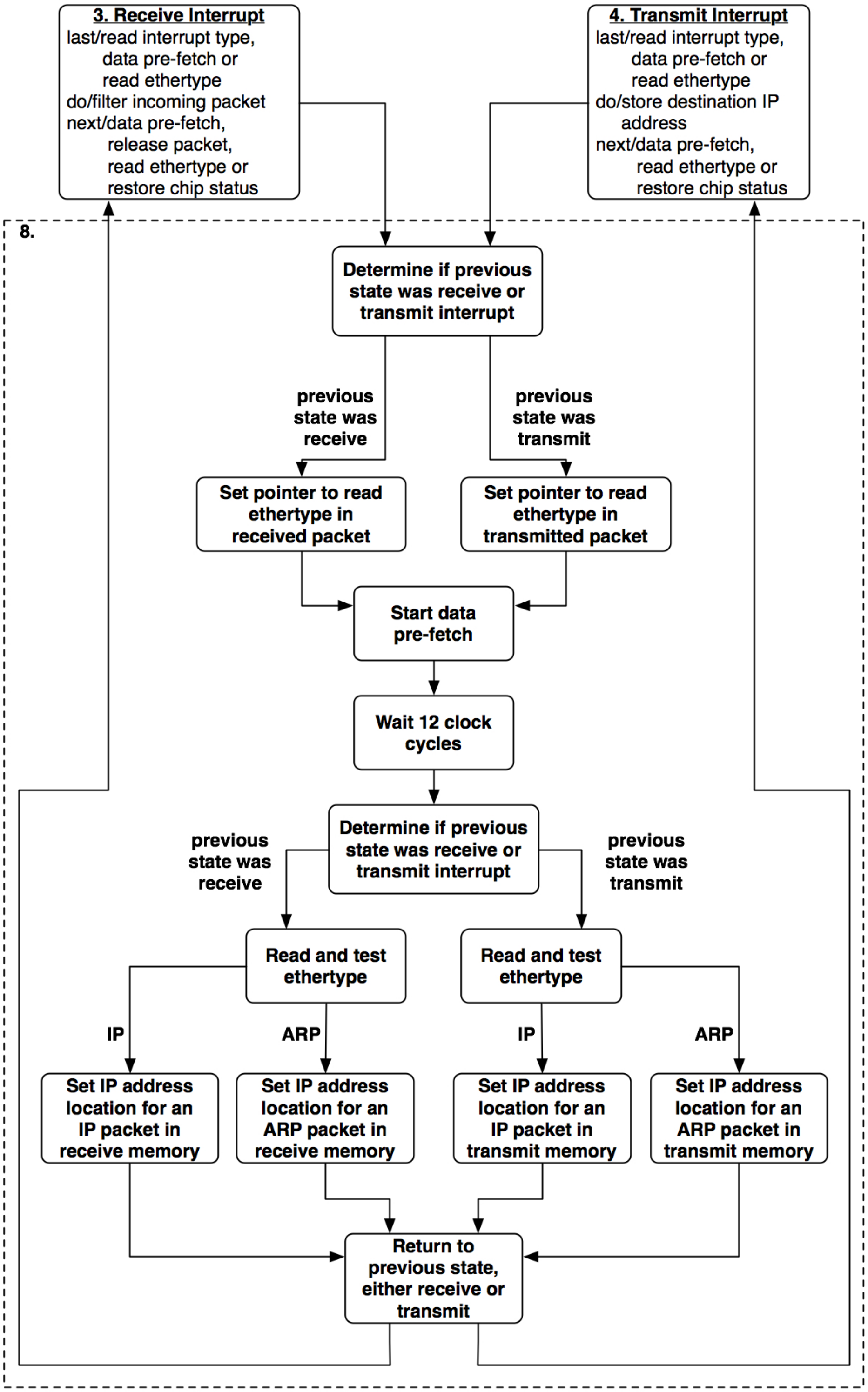

3. Receive Interrupt
last/read interrupt type,
data pre-fetch or
read ethertype
do/filter incoming packet
next/data pre-fetch,
release packet,
read ethertype or
restore chip status
4. Transmit Interrupt
last/read interrupt type,
data pre-fetch or
read ethertype
do/store destination IP
address
next/data pre-fetch,
read ethertype or
restore chip status
8.
Determine if previous state was receive or transmit interrupt
previous state was receive
previous state was transmit
Set pointer to read ethertype in received packet
Set pointer to read ethertype in transmitted packet
Start data pre-fetch
Wait 12 clock cycles
Determine if previous state was receive or transmit interrupt
previous state was receive
previous state was transmit
Read and test ethertype
Read and test ethertype
IP
ARP
IP
ARP
Set IP address location for an IP packet in receive memory
Set IP address location for an ARP packet in receive memory
Set IP address location for an IP packet in transmit memory
Set IP address location for an ARP packet in transmit memory
Return to previous state, either receive or transmit

Figure 4.10: Read Ethertype State Diagram

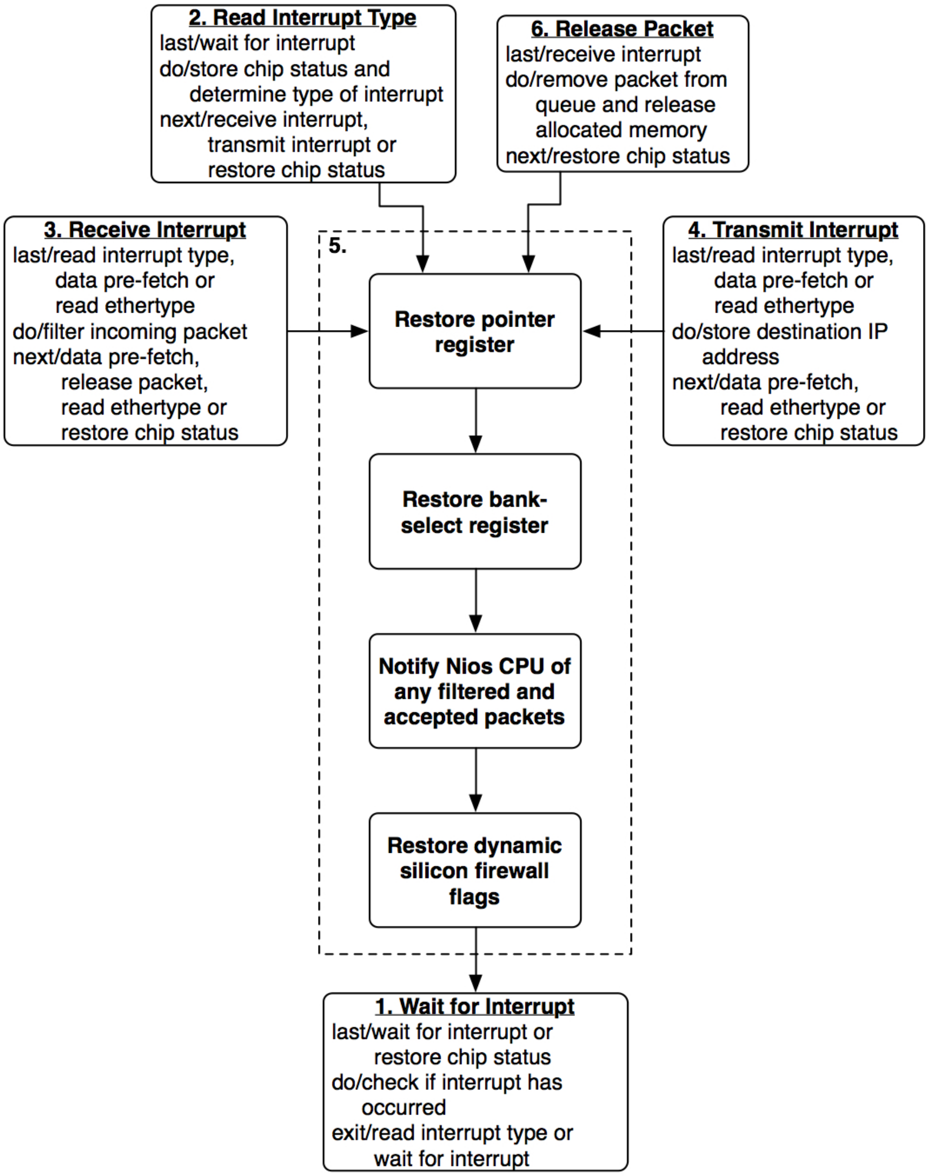

2. Read Interrupt Type
last/wait for interrupt
do/store chip status and determine type of interrupt
next/receive interrupt, transmit interrupt or restore chip status
6. Release Packet
last/receive interrupt
do/remove packet from queue and release allocated memory
next/restore chip status
3. Receive Interrupt
last/read interrupt type, data pre-fetch or read ethertype
do/filter incoming packet
next/data pre-fetch, release packet, read ethertype or restore chip status
5.
Restore pointer register
4. Transmit Interrupt
last/read interrupt type, data pre-fetch or read ethertype
do/store destination IP address
next/data pre-fetch, read ethertype or restore chip status
Restore bank-select register
Notify Nios CPU of any filtered and accepted packets
Restore dynamic silicon firewall flags
1. Wait for Interrupt
last/wait for interrupt or restore chip status
do/check if interrupt has occurred
exit/read interrupt type or wait for interrupt

Figure 4.11: Restore Chip State Diagram

Chapter 5
Silicon Firewall Software Components

The software components of the dynamic silicon firewall will be discussed in this chapter. These software modules rely upon example software supplied by Altera® corporation. The functionality of this example software will be covered, as well as the changes made to the example which were required for the dynamic silicon firewall project.

5.1 Altera® Plugs Ethernet Library

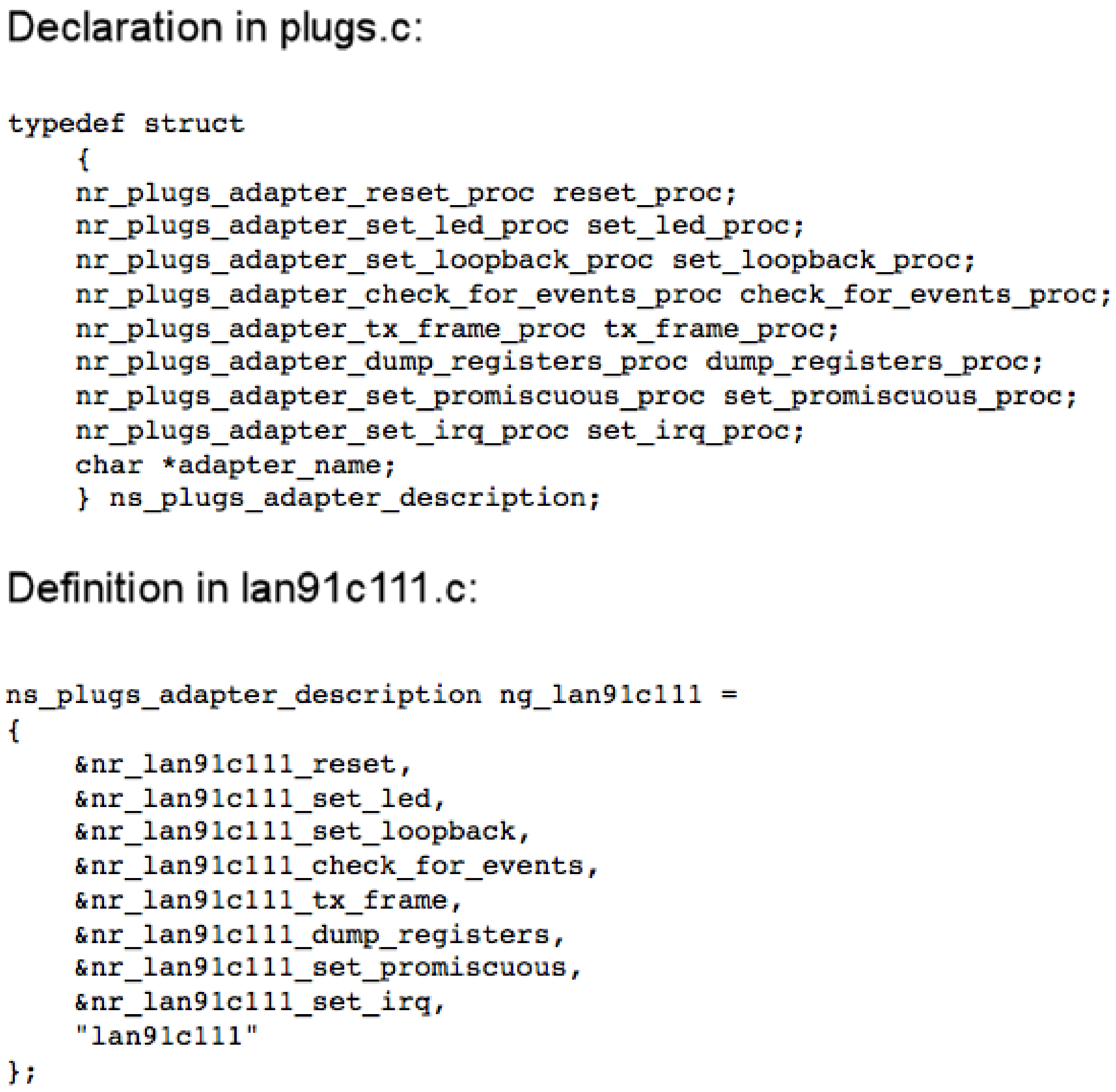
Altera® provides software libraries to support network connections when using the Nios® soft-core processor. These software libraries are collectively known as the Plugs Ethernet libraries and they essentially are the TCP/IP stack implementation for the Nios® processor. Plugs offer similar functionality to the Nios® processor, as traditional network sockets do to an operating system running on a general purpose computer. The Plugs Ethernet libraries support a number of network protocols, including those with which the dynamic silicon firewall concerns itself, namely Ethernet, IP and ARP. The Plugs source code is abstracted from the underlying hardware so that it is compatible with differing Ethernet hardware. A change in hardware simply requires a change in the underlying hardware drivers relied upon by Plugs to ensure functionality.

5.1.1 Changes to the Plugs Ethernet Library

The software routines supplied by the Plugs Ethernet libraries can be used by including the plugs.h header file in any applications designed for the Nios® processor. The majority of the hardware-abstracted routines, defined in plugs.h, are implemented in the plugs.c file, while the lower-level hardware driver routines for the LAN91C111 Ethernet chip are implemented in the lan91c111.c file. Due to the way various routines were implemented by Altera®, a few changes were required in the plugs.c and lan91c111.c files to ensure the functionality of the dynamic silicon firewall project.

Since both the silicon firewall module and the Nios® CPU need to communicate with the LAN91C111 Ethernet chip, it must be ensured that only one of these entities are in communication with the chip at any one time. This is handled by forcing the Nios® processor to request access to the Ethernet chip before communicating with it. This request is sent to the silicon firewall, and then the firewall module grants the request when the Ethernet chip is not being used. The time it takes until a request is answered will depend on which state the firewall module is in when the request is made. If the firewall module is in the middle of communication with the Ethernet chip, the request is not granted until the end of communication. Examples of this delay are the time it would take the Verilog firewall module to handle an incoming packet, or the time it takes to store the destination IP address of an outgoing packet. The silicon firewall module checks for any transmit requests from the Nios® processor after each packet is handled, meaning the Nios® processor will never be completely prevented from communicating with the Ethernet chip. After the Nios® CPU has finished communicating with the LAN91C111 it must notify the silicon firewall module that it no longer requires access to the Ethernet adapter.

Transmit requests needed to be inserted into the plugs.c file in all places where chip-specific routines from lan91c111.c were being called. The lan91c111.c routines are accessed through a structure representing the specific Ethernet adapter functions. This structure is declared in plugs.c but defined in lan91c111.c. Figure 5.1 shows the declaration and definition of the ns_plugs_adapter_description structure. As can be seen, all communication between the hardware-abstracted plugs routines and the LAN91C111-specific routines go through the adapter description structure and the 8 routines contained therein. Of the 8 routines in the adapter description, only 6 are called in the plugs.c file. The routines nr_lan91c111_dump_registers and



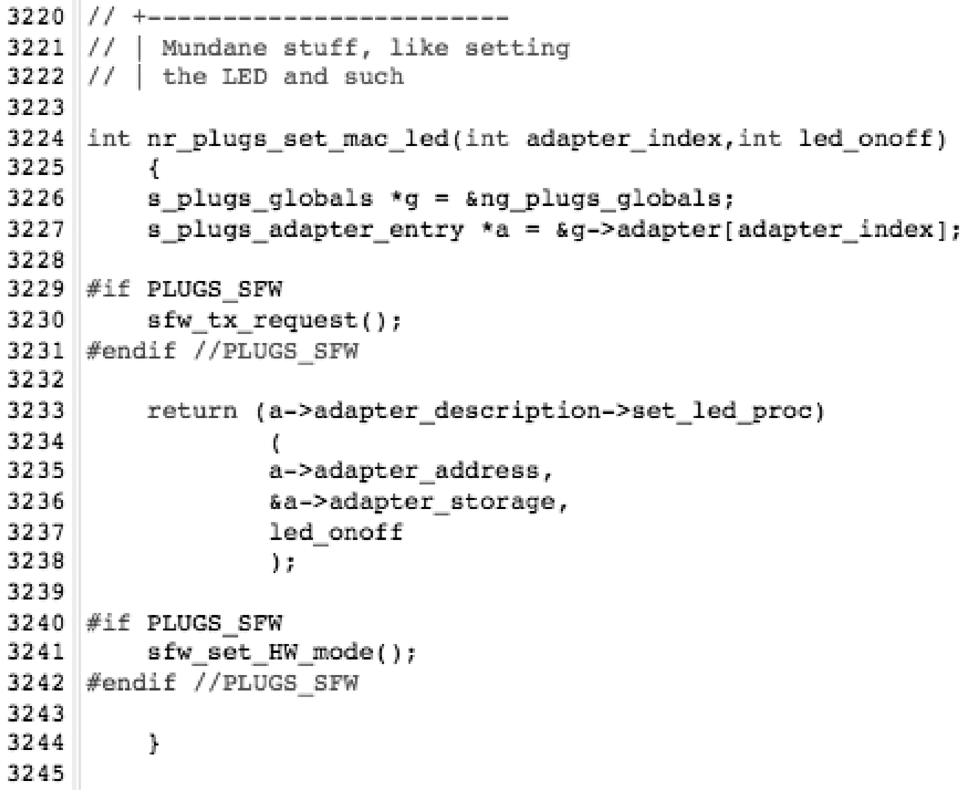
Declaration in plugs.c:

```
typedef struct
    {
    nr_plugs_adapter_reset_proc reset_proc;
    nr_plugs_adapter_set_led_proc set_led_proc;
    nr_plugs_adapter_set_loopback_proc set_loopback_proc;
    nr_plugs_adapter_check_for_events_proc check_for_events_proc;
    nr_plugs_adapter_tx_frame_proc tx_frame_proc;
    nr_plugs_adapter_dump_registers_proc dump_registers_proc;
    nr_plugs_adapter_set_promiscuous_proc set_promiscuous_proc;
    nr_plugs_adapter_set_irq_proc set_irq_proc;
    char *adapter_name;
    } ns_plugs_adapter_description;
```

Definition in lan91c111.c:

```
ns_plugs_adapter_description ng_lan91c111 =
{
    &nr_lan91c111_reset,
    &nr_lan91c111_set_led,
    &nr_lan91c111_set_loopback,
    &nr_lan91c111_check_for_events,
    &nr_lan91c111_tx_frame,
    &nr_lan91c111_dump_registers,
    &nr_lan91c111_set_promiscuous,
    &nr_lan91c111_set_irq,
    "lan91c111"
};
```

Figure 5.1: Plugs Adapter Description



```
// +------------------------
// | Mundane stuff, like setting
// | the LED and such

int nr_plugs_set_mac_led(int adapter_index,int led_onoff)
    {
    s_plugs_globals *g = &ng_plugs_globals;
    s_plugs_adapter_entry *a = &g->adapter[adapter_index];

#if PLUGS_SFW
    sfw_tx_request();
#endif //PLUGS_SFW

    return (a->adapter_description->set_led_proc)
            (
            a->adapter_address,
            &a->adapter_storage,
            led_onoff
            );

#if PLUGS_SFW
    sfw_set_HW_mode();
#endif //PLUGS_SFW

    }

```

Figure 5.2: Plugs Example Routine

nr_lan91c111_set_promiscuous are never called by any routines in plugs.c.

There are a total of 16 calls in plugs.c to the 6 remaining adapter description routines, requiring each of these calls to be wrapped by code which requests access to the Ethernet adapter before being called, and code which notifies the Verilog firewall that communication has finished after the routine returns. Figure 5.2 illustrates an example routine from plugs.c named nr_plugs_set_mac_led which is simply used to turn the media access control activity light-emitting diode on or off. This simple routine calls another routine from lan91c111.c, requiring a transmit request before the set_led_proc routine is called, and a notification that communication has finished after the set_led_proc routine returns.

In addition to the changes made in plugs.c, some modifications were needed in lan91c111.c. As previously stated, lan91c111.c contains the software routines that are specific to the actual hardware Ethernet adapter being used. Two important changes were made in this file. Upon receipt of a packet, the routine where received packets are read from the on-chip buffer memory was designed to read multiple queued incoming packets. Without modification, this would mean that once the packet at the front of the queue had passed through the firewall, all subsequent packets in the queue would also pass through by default. This is undesirable, since we want to ensure that each packet came from a trusted IP address for security purposes. A modification was made so that packets are individually read from the Ethernet buffer memory, meaning each packet is tested by the firewall before being read by the Nios® software. The other modification required for the silicon firewall to function properly concerns the LAN91C111 interrupts. By default, the Plugs libraries do not enable transmit interrupts. Transmissions were conducted by queueing the outgoing packet without monitoring if the transmission was successful. This is acceptable in the sense that Ethernet is an unreliable protocol, where any communication reliability is left to higher level protocols such as TCP. However, the silicon firewall project depends on knowing when packets have been transmitted, so that it can find and store the destination IP address. Therefore, the transmit interrupt needed to be added to the interrupt mask used when the LAN91C111 interrupts are enabled during initialization of the adapter.

5.2 Project Software

In addition to the Plugs Ethernet library, custom software was needed to manage three aspects of the project. The dynamic aspect of the project was provided by a software-controlled timer and associated routines. Also, as previously mentioned, the Plugs libraries were modified to ensure either the Nios® CPU or the firewall module is in sole communication with the Ethernet chip. Some custom software routines were also needed to accomplish this. Finally, for ease of use in the prototype test system, initialization of the project components was handled in software. Provisions to provide for these three additional project components are in the file sfw.c and the associated header, sfw.h.

5.2.1 Silicon Firewall Routines

To ensure that IP addresses are not trusted forever, a timer is used to periodically remove them from the trusted list stored in the CAM. For the prototype test system, a timer interrupt was generated after each two minute period. While the timer is a hardware component to the system, software handles the IP removal during an interrupt service routine. The interrupt service routine is installed when the timer is started and the Plugs software is initialized. When called, the interrupt service routine cycles through each address in the RAM, checking whether the associated data is a one or zero. As previously discussed, a one represents a fresh IP address is stored in the CAM at the same memory address, while a zero represents a stale entry. For each RAM address containing a one in data, the interrupt service routine simply re-writes the data with a zero. In this way, IP addresses are marked stale every minute, on average, using the two minute timer period. For the RAM entries denoting stale IP addresses, the matching CAM entry is deleted. Therefore, an IP address is trusted for an average of three minutes from the time an outgoing packet is sent; an average of one minute until a CAM entry is marked stale, followed by two full minutes until it is deleted.

Two software routines are used by the Nios® CPU to keep track of what piece of hardware is in communication with the LAN91C111 Ethernet chip. The system was designed so that the Nios® processor must request Ethernet access from the silicon firewall. This request and clear to transmit process is accomplished with two signals between the Nios® processor and silicon firewall, tx_request and tx_allowed. When the Nios® processor needs to transmit a packet onto the network, it calls a routine which sets the tx_request signal to one. The silicon firewall checks if a transmit request has been made on the rising edge of every clock cycle that is not being used to process an Ethernet interrupt. If a transmit request has occurred, then the silicon firewall sets the tx_allowed signal to one, notifying the Nios® CPU that it may communicate with the Ethernet chip. In addition to the two signals used for negotiating access to the Ethernet chip, another signal is used to keep track of whether the Nios® processor or the silicon firewall is in communication. This signal is called the mode signal. The entire system can also be viewed as being separated into two modes, hardware mode, where the silicon firewall is in communication with the LAN91C111, and software mode, where the Nios® processor is in communication. The mode signal is set by the Nios® processor, but monitored by both the Nios® processor and the silicon firewall. Two software routines are used by the Nios® processor to set and get the value of the mode signal. When the Nios® processor successfully requests access to the Ethernet chip, it sets the mode signal to zero, representing software mode. After communication is over, it returns the signal to one, representing hardware mode. The silicon firewall module monitors the mode and only processes packets in hardware mode, when it has exclusive access to the Ethernet chip. Just as the silicon firewall module checks for pending transmit requests from the Nios® processor after each packet is processed, the Nios® processor sets the mode signal back to hardware mode after each transmission. This way, neither the Nios® processor nor the silicon firewall could block communication with the Ethernet chip (and associated external network) forever.

To simplify testing of the silicon firewall project, a number of initialization routines were added and used in the system. As mentioned, a timer initialization routine was used to install an interrupt service routine for the periodic removal of IP addresses from the CAM. In addition, RAM and CAM initialization routines were used for setting up known states for both the RAM and CAM entries. These were used in testing various aspects of the project and will be covered again when discussing the testing and results of the project.

Chapter 6

Testing and Results

The SignalTap® [20] embedded logic analyzer and Ethereal® [28] network analyzer were used to monitor and test the dynamic silicon firewall. Both pieces of software will be introduced, and their functionality explained before the testing method and results of the project are presented.

6.1 Analytical Software Tools

The SignalTap® embedded logic analyzer is a software tool provided by Altera® corporation for use in debugging a system on a programmable chip created using the Quartus® II software. It allows engineers to monitor all signals in the system in real time during operation. Since the SignalTap® logic analyzer is a soft-core tool, multiple instances may be used if desired, each monitoring a different clock source in the system. Furthermore, it is capable of monitoring up to 1,024 signals while taking up to 128 thousand samples per signal. SignalTap® also supports up to ten trigger levels to denote when the capture of data begins. Since each SignalTap® instance resides on the same PLD as the system being monitored, all of the above features are dependent upon the resources available on the device being used. Smaller logic devices may not have enough memory to support the full amount of signals or samples. As samples are taken, the information is stored on the PLD and then streamed via a communication cable to the Quartus® II software running on a computer. Quartus® II then displays the waveform output for each signal captured. The sampled data can also be exported in various formats for verification using other software tools.

Ethereal® is an open-source software tool used to monitor network traffic. It can capture packets directly off of an Ethernet network, or open a set of previously saved network packets, and then display detailed information about the data contained in those packets. It should be noted that Ethereal® supports a few data link layer protocols other than Ethernet, but since this project concerns itself solely with Ethernet, the functionality of Ethereal® on other networks will not be discussed. Ethereal® can dissect 750 higher-level network protocols (application, transport, etc.), displaying the bytes of the various packet fields in a graphical user interface. It also has the capability to sort a set of captured packets based on packet fields, such as TCP ports or the IP addresses. The real-time capture of data off of a functioning network is accomplished by monitoring all network packets seen by the network interface in the computer on which Ethereal® is running. This requires the network adapter to be in promiscuous mode, and that the computer running Ethereal® is not on a switched network. Switched networks direct traffic destined for a particular computer to the specific portion of the network containing the destination. Non-switched networks send all data packets to each adapter on the network, relying on each network adapter to determine if the data is destined for itself. Promiscuous mode allows the adapter to see packets that are not destined for its own particular MAC address. Therefore, a combination of a non-switched network and a promiscuous mode adapter is needed to monitor all traffic on a network.

6.2 Testing Method

To test the functionality of the dynamic silicon firewall, a real-world situation (some application layer protocol) needed to be chosen. Web browsing was chosen because Altera® provides a sample web server application for the Nios® processor, and because the HyperText Transfer Protocol (HTTP) is a simple application layer protocol which is extensively used by most people who have internet access.

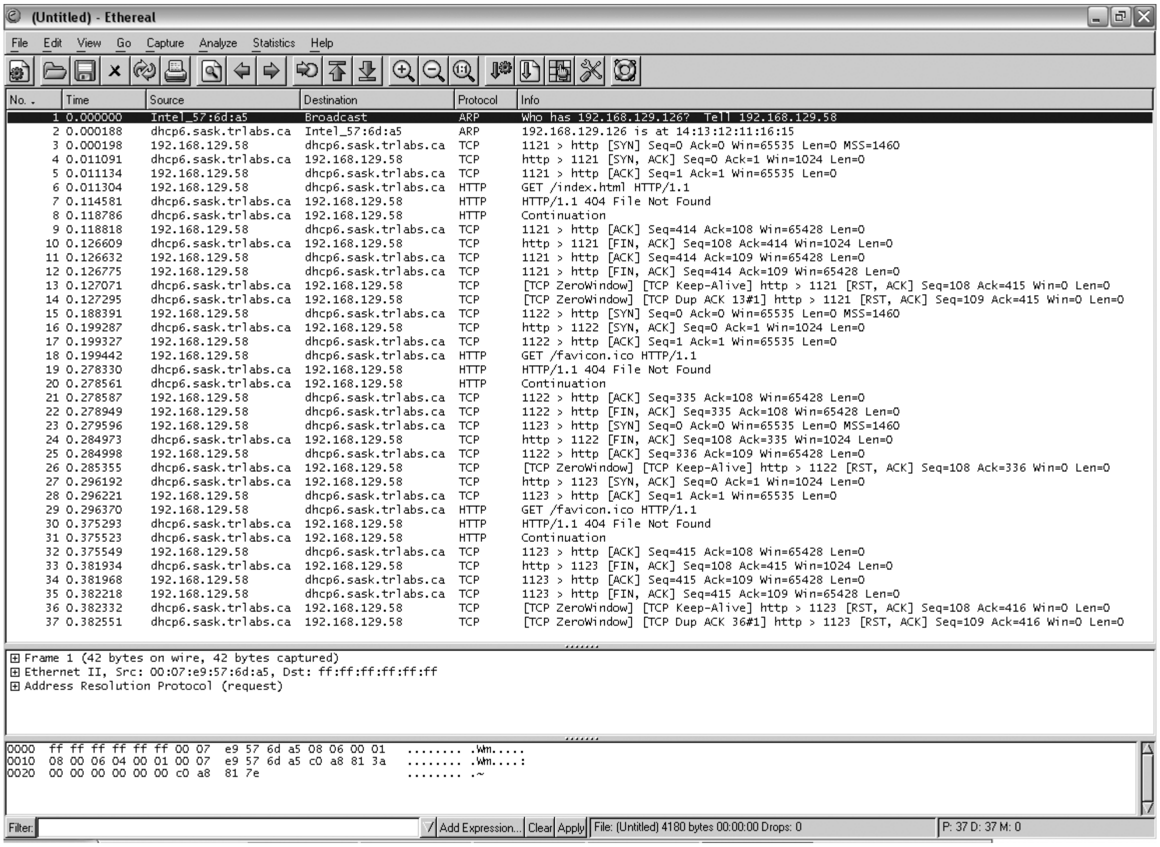
A local area network (LAN) was set up, containing the Stratix® development board and multiple computers. The web server application was set up to run on the Nios® processor, serving HTTP requests to the computers on the network. The dynamic silicon firewall was used to protect the web server's Ethernet interface from unwanted access. As previously mentioned, software routines were used during initialization to specify some IP addresses granted initial access, since the dynamic silicon firewall only accepts incoming packets from IP addresses to which packets have previously been transmitted. Due to the dynamic aspect of the project, the IP addresses written to the CAM during start up are trusted for an average of three minutes unless communication during this period continues.

To test basic functionality, computers on the LAN attempted to access the web server running on the development board, while the Ethernet traffic was monitored using Ethereal®. In this way, it could be seen if the computers whose IP addresses were entered into the CAM during initialization received access, as they should, while checking whether all requests from other computers were ignored. The dynamic aspect of the project could also be tested using this set up, by checking whether continued communication between the web server and trusted computers resulted in continued access. Finally, sending another HTTP request after a period of inactivity, and monitoring the network traffic, allowed verification of the IP removal from the trusted list stored in the CAM.

Running the above tests while monitoring on-chip operation using SignalTap® provided a means for verification of timing requirements and state machine operation. SignalTap® triggers could be set up for each state of the finite state machine, allowing the capture and verification of signals between the Nios® processor, dynamic silicon firewall HDL module and the LAN91C111 Ethernet chip.

6.3 Results

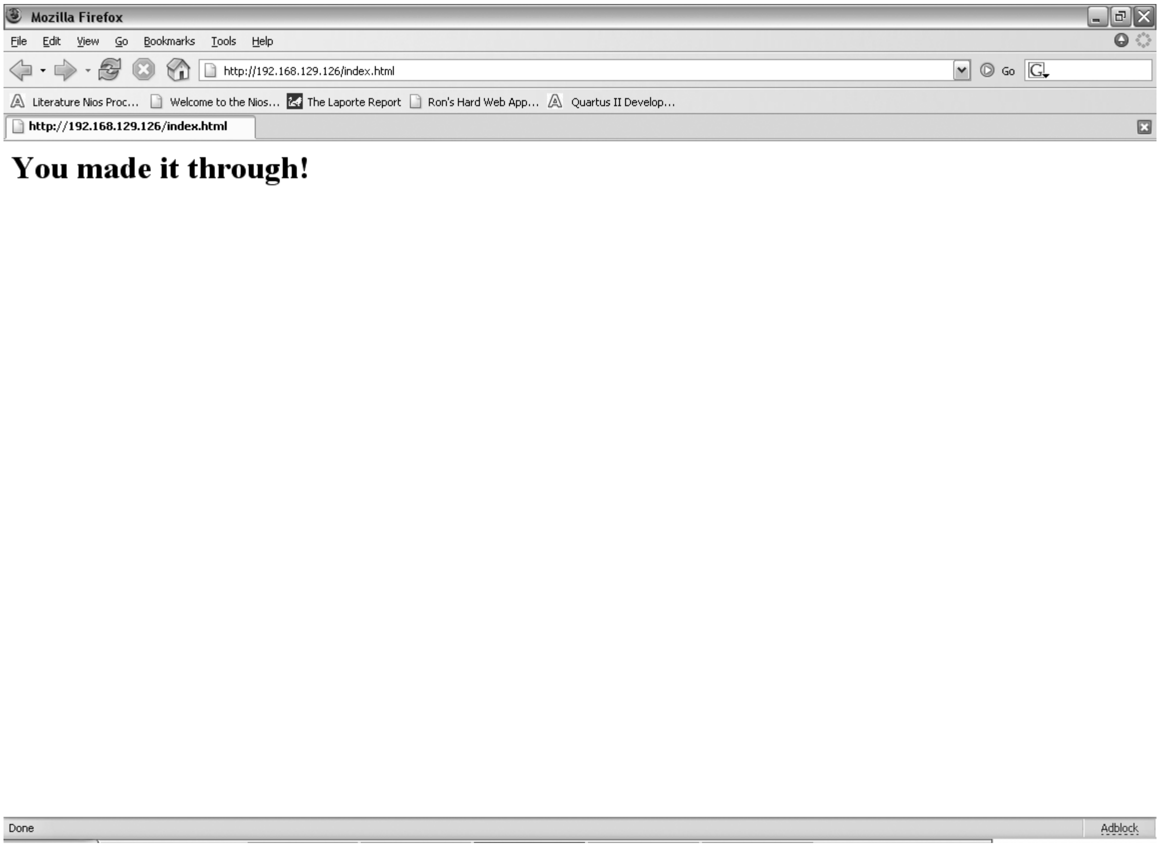
The basic tests of functionality showed that the dynamic silicon firewall operated as intended. Using the testing method described above, it was seen that the IP addresses entered into the CAM during initialization were trusted as intended. The Ethernet packets transferred between a trusted host and the Stratix® development



```
No. . Time     Source                Destination           Protocol Info
    1 0.000000 Intel_57:6d:a5        Broadcast             ARP      Who has 192.168.129.126?  Tell 192.168.129.58
    2 0.000188 dhcp6.sask.trlabs.ca  Intel_57:6d:a5        ARP      192.168.129.126 is at 14:13:12:11:16:15
    3 0.000198 192.168.129.58        dhcp6.sask.trlabs.ca  TCP      1121 > http [SYN] Seq=0 Ack=0 Win=65535 Len=0 MSS=1460
    4 0.011091 dhcp6.sask.trlabs.ca  192.168.129.58        TCP      http > 1121 [SYN, ACK] Seq=0 Ack=1 Win=1024 Len=0
    5 0.011134 192.168.129.58        dhcp6.sask.trlabs.ca  TCP      1121 > http [ACK] Seq=1 Ack=1 Win=65535 Len=0
    6 0.011304 192.168.129.58        dhcp6.sask.trlabs.ca  HTTP     GET /index.html HTTP/1.1
    7 0.114581 dhcp6.sask.trlabs.ca  192.168.129.58        HTTP     HTTP/1.1 404 File Not Found
    8 0.118786 dhcp6.sask.trlabs.ca  192.168.129.58        HTTP     Continuation
    9 0.118818 192.168.129.58        dhcp6.sask.trlabs.ca  TCP      1121 > http [ACK] Seq=414 Ack=108 Win=65428 Len=0
   10 0.126609 dhcp6.sask.trlabs.ca  192.168.129.58        TCP      http > 1121 [FIN, ACK] Seq=108 Ack=414 Win=1024 Len=0
   11 0.126632 192.168.129.58        dhcp6.sask.trlabs.ca  TCP      1121 > http [ACK] Seq=414 Ack=109 Win=65428 Len=0
   12 0.126775 192.168.129.58        dhcp6.sask.trlabs.ca  TCP      1121 > http [FIN, ACK] Seq=414 Ack=109 Win=65428 Len=0
   13 0.127071 dhcp6.sask.trlabs.ca  192.168.129.58        TCP      [TCP ZeroWindow] [TCP Keep-Alive] http > 1121 [RST, ACK] Seq=108 Ack=415 Win=0 Len=0
   14 0.127295 dhcp6.sask.trlabs.ca  192.168.129.58        TCP      [TCP ZeroWindow] [TCP Dup ACK 13#1] http > 1121 [RST, ACK] Seq=109 Ack=415 Win=0 Len=0
   15 0.188391 192.168.129.58        dhcp6.sask.trlabs.ca  TCP      1122 > http [SYN] Seq=0 Ack=0 Win=65535 Len=0 MSS=1460
   16 0.199287 dhcp6.sask.trlabs.ca  192.168.129.58        TCP      http > 1122 [SYN, ACK] Seq=0 Ack=1 Win=1024 Len=0
   17 0.199327 192.168.129.58        dhcp6.sask.trlabs.ca  TCP      1122 > http [ACK] Seq=1 Ack=1 Win=65535 Len=0
   18 0.199442 192.168.129.58        dhcp6.sask.trlabs.ca  HTTP     GET /favicon.ico HTTP/1.1
   19 0.278330 dhcp6.sask.trlabs.ca  192.168.129.58        HTTP     HTTP/1.1 404 File Not Found
   20 0.278561 dhcp6.sask.trlabs.ca  192.168.129.58        HTTP     Continuation
   21 0.278587 192.168.129.58        dhcp6.sask.trlabs.ca  TCP      1122 > http [ACK] Seq=335 Ack=108 Win=65428 Len=0
   22 0.278949 192.168.129.58        dhcp6.sask.trlabs.ca  TCP      1122 > http [FIN, ACK] Seq=335 Ack=108 Win=65428 Len=0
   23 0.279596 192.168.129.58        dhcp6.sask.trlabs.ca  TCP      1123 > http [SYN] Seq=0 Ack=0 Win=65535 Len=0 MSS=1460
   24 0.284973 dhcp6.sask.trlabs.ca  192.168.129.58        TCP      http > 1122 [FIN, ACK] Seq=108 Ack=335 Win=1024 Len=0
   25 0.284998 192.168.129.58        dhcp6.sask.trlabs.ca  TCP      1122 > http [ACK] Seq=336 Ack=109 Win=65428 Len=0
   26 0.285355 dhcp6.sask.trlabs.ca  192.168.129.58        TCP      [TCP ZeroWindow] [TCP Keep-Alive] http > 1122 [RST, ACK] Seq=108 Ack=336 Win=0 Len=0
   27 0.296192 dhcp6.sask.trlabs.ca  192.168.129.58        TCP      http > 1123 [SYN, ACK] Seq=0 Ack=1 Win=1024 Len=0
   28 0.296221 192.168.129.58        dhcp6.sask.trlabs.ca  TCP      1123 > http [ACK] Seq=1 Ack=1 Win=65535 Len=0
   29 0.296370 192.168.129.58        dhcp6.sask.trlabs.ca  HTTP     GET /favicon.ico HTTP/1.1
   30 0.375293 dhcp6.sask.trlabs.ca  192.168.129.58        HTTP     HTTP/1.1 404 File Not Found
   31 0.375523 dhcp6.sask.trlabs.ca  192.168.129.58        HTTP     Continuation
   32 0.375549 192.168.129.58        dhcp6.sask.trlabs.ca  TCP      1123 > http [ACK] Seq=415 Ack=108 Win=65428 Len=0
   33 0.381934 dhcp6.sask.trlabs.ca  192.168.129.58        TCP      http > 1123 [FIN, ACK] Seq=108 Ack=415 Win=1024 Len=0
   34 0.381968 192.168.129.58        dhcp6.sask.trlabs.ca  TCP      1123 > http [ACK] Seq=415 Ack=109 Win=65428 Len=0
   35 0.382218 192.168.129.58        dhcp6.sask.trlabs.ca  TCP      1123 > http [FIN, ACK] Seq=415 Ack=109 Win=65428 Len=0
   36 0.382332 dhcp6.sask.trlabs.ca  192.168.129.58        TCP      [TCP ZeroWindow] [TCP Keep-Alive] http > 1123 [RST, ACK] Seq=108 Ack=416 Win=0 Len=0
   37 0.382551 dhcp6.sask.trlabs.ca  192.168.129.58        TCP      [TCP ZeroWindow] [TCP Dup ACK 36#1] http > 1123 [RST, ACK] Seq=109 Ack=416 Win=0 Len=0

⊞ Frame 1 (42 bytes on wire, 42 bytes captured)
⊞ Ethernet II, Src: 00:07:e9:57:6d:a5, Dst: ff:ff:ff:ff:ff:ff
⊞ Address Resolution Protocol (request)

0000  ff ff ff ff ff ff 00 07  e9 57 6d a5 08 06 00 01   ........ .Wm.....
0010  08 00 06 04 00 01 00 07  e9 57 6d a5 c0 a8 81 3a   ........ .Wm....:
0020  00 00 00 00 00 00 c0 a8  81 7e                     ........ .~

File: (Untitled) 4180 bytes 00:00:00 Drops: 0          P: 37 D: 37 M: 0
```

Figure 6.1: Host Communication Succeeding

board were captured using Ethereal® and are presented as an example in figure 6.1. In the example shown, the host computer has an IP address of 192.168.129.58, while the development board has the IP address 192.168.129.126, which can be translated to the domain name dhcp6.sask.trlabs.ca. The first packet sent from the host is an ARP request, because the host needs to know the MAC address of the development board for communication to succeed. Since the host is trusted, an ARP reply is sent out, followed by the expected TCP handshake and HTTP request packets. The web server has the capability to access and transmit web pages stored in flash memory on the development board, however, this feature was not utilized due to hardware problems resulting from the way flash memory was connected to the FPGA on the development board. A simple hard-coded message is delivered in response to all web page requests successfully delivered to the development board. This is shown figure 6.2, which is a screen capture of the web browser running on the trusted host

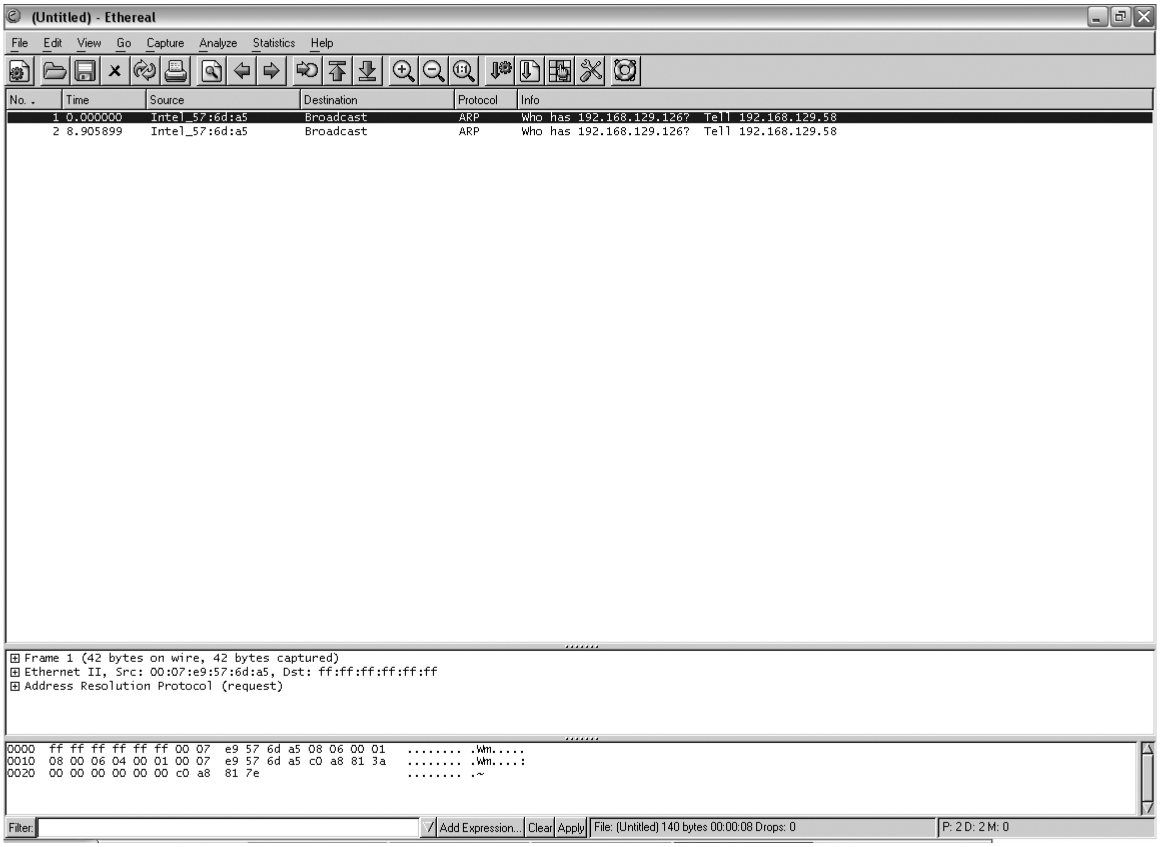

Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://192.168.129.126/index.html
Go
Literature Nios Proc... Welcome to the Nios... The Laporte Report Ron's Hard Web App... Quartus II Develop...
http://192.168.129.126/index.html
Done
Adblock

You made it through!

Figure 6.2: Host Web Browser Received Message

computer.

As designed, the dynamic silicon firewall prevented untrusted hosts from communicating with the development board. Also, trusted hosts were purged after a period of inactivity as expected. Ethernet traffic for the case of a previously trusted host attempting to contact the development board after a period of inactivity is shown in figure 6.3. In this case, because the computer making the HTTP request is not trusted, the dynamic silicon firewall drops the packets containing the ARP request. All that is seen in the captured packet data are two unanswered ARP requests from the host. There is no response from the web server, since the packets were dropped, and thus the host making the web request can not even tell that the development board is connected to the network. The host web browser sent two ARP requests before timing out and reporting an error, as seen in the web browser image shown in figure 6.4.



```
No. .   Time        Source            Destination       Protocol  Info
      1 0.000000    Intel_57:6d:a5    Broadcast         ARP       Who has 192.168.129.126?  Tell 192.168.129.58
      2 8.905899    Intel_57:6d:a5    Broadcast         ARP       Who has 192.168.129.126?  Tell 192.168.129.58
```

```
⊞ Frame 1 (42 bytes on wire, 42 bytes captured)
⊞ Ethernet II, Src: 00:07:e9:57:6d:a5, Dst: ff:ff:ff:ff:ff:ff
⊞ Address Resolution Protocol (request)
```

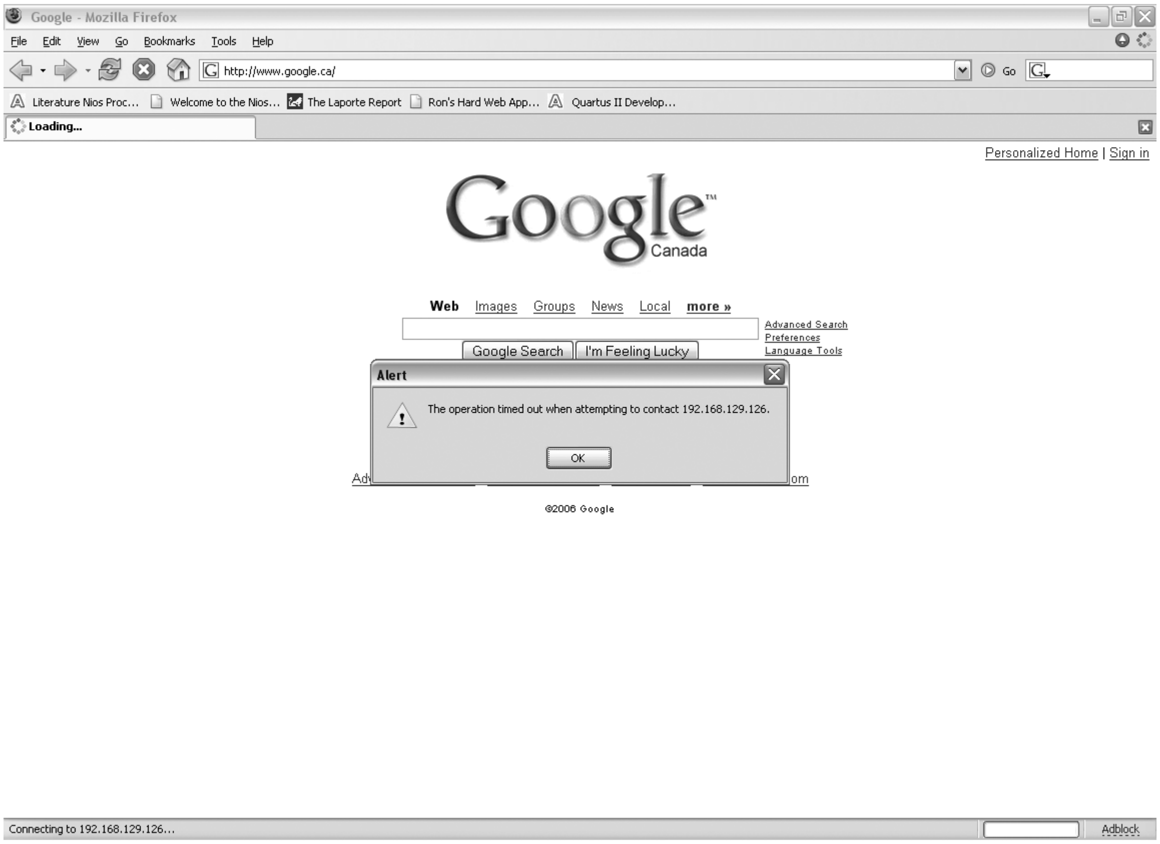
```
0000  ff ff ff ff ff ff 00 07  e9 57 6d a5 08 06 00 01   ........ .Wm.....
0010  08 00 06 04 00 01 00 07  e9 57 6d a5 c0 a8 81 3a   ........ .Wm....:
0020  00 00 00 00 00 00 c0 a8  81 7e                     ........ .~
```

```
File: (Untitled) 140 bytes 00:00:08 Drops: 0    P: 2 D: 2 M: 0
```

Figure 6.3: Host Communication Failing

The SignalTap® logic analyzer was used to capture a set of signal samples for each state of the dynamic silicon firewall. These samples were then displayed in Quartus® II as a set of waveforms, allowing verification of the operation of each state. Figure 6.5 shows sample waveforms from the reception interrupt state. The signals were sampled on the rising edge of the system clock, with each clock cycle numbered across the top of the image, starting from the initial trigger point. The command_cycle and timing_cycle signals show the state of the system, while the enet_Data and enet_Address signals show the communication lines between the dynamic silicon firewall and the Ethernet chip. While testing, additional signals were captured to show the operation of the CAM in the system, as well as the timer function and IP address purge.

The SignalTap® waveform data also offered a way to ensure that all timing requirements were met in the project. The LAN91C111 data sheet [27] illustrates


Google - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://www.google.ca/
Go
Literature Nios Proc... Welcome to the Nios... The Laporte Report Ron's Hard Web App... Quartus II Develop...
Loading...
Personalized Home | Sign in
Google
Canada
Web Images Groups News Local more »
Advanced Search
Preferences
Language Tools
Google Search
I'm Feeling Lucky
Alert
OK
©2006 Google
Adblock

The operation timed out when attempting to contact 192.168.129.126.

Connecting to 192.168.129.126...

Figure 6.4: Host Web Browser Timed Out

a number of setup and hold times required for use of the chip. These timing requirements are shown in figure 6.6. Since the system clock operates using a 30ns period, the timing requirements of the Ethernet chip were easily met when signals were changed once per clock cycle. As an example, for the dynamic silicon firewall to read information from the LAN91C111 it is required that a register address is set, then the Ethernet read line must be dropped low, held low for 15ns, after which the data can be read from the Ethernet chip. Each of these operations are done in one 30ns clock cycle, satisfying the timing requirements for the LAN91C111 since the greatest requirement is a hold time of 15ns. The timing for communication with the Ethernet chip was verified using SignalTap®, which could show that each of the LAN91C111 signal lines were changed in compliance with the specifications laid out in the data sheet. Also discussed in the LAN91C111 data sheet is a wait period of 370ns which is required between the time the pointer register is directed to buffer

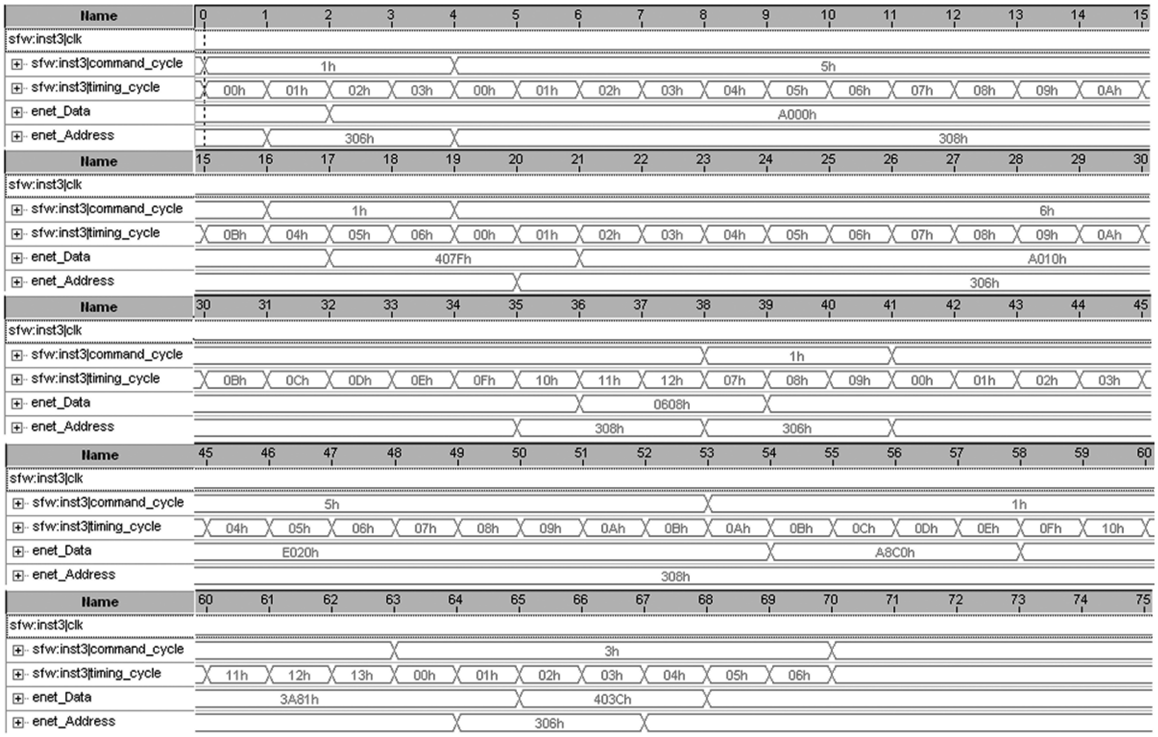
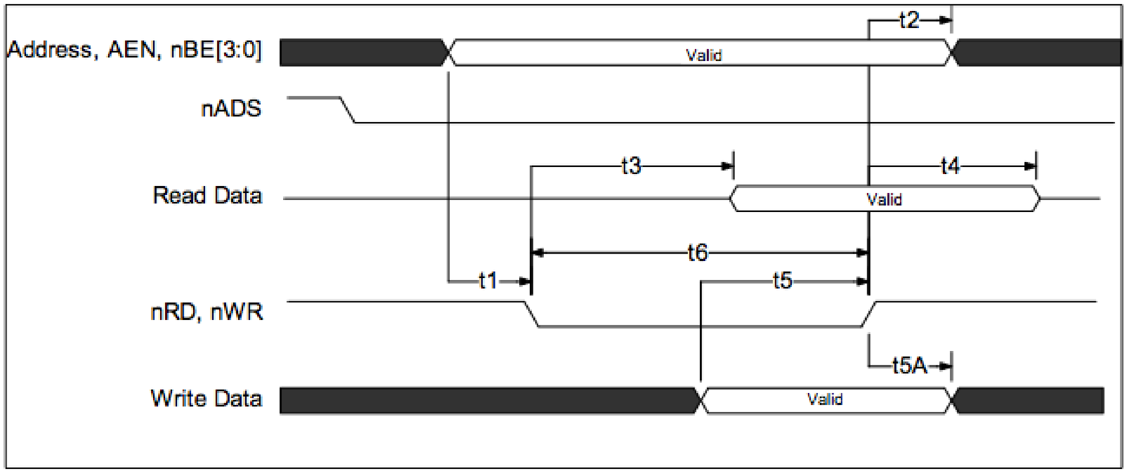
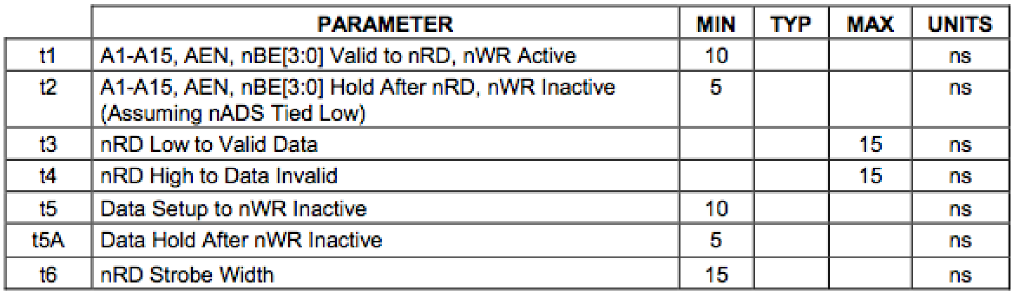


Figure 6.5: SignalTap® Waveform Data

memory, and when that data can be accessed. Since the system clock period is 30ns, 13 clock cycles were used, resulting in a wait period of 390ns, complying with the 370ns requirement. This wait period for the data pre-fetch is required due to the fact that the asynchronous ready signal of the LAN91C111 was not available in the implementation used on the Stratix® development board, and therefore could not be monitored by the system to determine when the data pre-fetch finished and valid data was available.

Finally, it is important that the dynamic silicon firewall does not negatively impact network transmission speeds in any great way. The LAN91C111 chip is capable of 100Mbps transfer speeds, and the dynamic silicon firewall project was designed while keeping in mind that these speeds should not be hampered due to the firewall processing delay. By counting the number of 30ns clock cycles required for a transmit or receive interrupt to be processed by the dynamic silicon firewall, it can be shown that 2.91μs is the maximum amount of time a packet will be delayed by processing. This is the case for a successfully transmitted packet, though the





	PARAMETER	MIN	TYP	MAX	UNITS
t1	A1-A15, AEN, nBE[3:0] Valid to nRD, nWR Active	10			ns
t2	A1-A15, AEN, nBE[3:0] Hold After nRD, nWR Inactive (Assuming nADS Tied Low)	5			ns
t3	nRD Low to Valid Data			15	ns
t4	nRD High to Data Invalid			15	ns
t5	Data Setup to nWR Inactive	10			ns
t5A	Data Hold After nWR Inactive	5			ns
t6	nRD Strobe Width	15			ns

Figure 6.6: LAN91C111 Timing Requirements

case for a received packet from an untrusted source is roughly the same, requiring 2.85μs to process. Three data pre-fetch wait periods of 370ns each are required in the above two cases, making up about a third of the maximum processing delay. As previously mentioned, the data pre-fetch delay is constant due to the development board design, and a reduction could be made using a different hardware design where the asynchronous ready signal is available to the dynamic silicon firewall. This aspect of the project will be discussed further in the future work section.

Chapter 7

Summary, Conclusion and Future Work

This thesis has sought to address some of the network security issues prevalent in home networking by improving upon network firewall technology. Current firewalls targeted to the home user often require knowledge regarding the underlying network communication protocols to set up and manage. Furthermore, they usually do not have a default setting which works for the majority of home users. The dynamic silicon firewall was designed to mitigate these two main problems. In this chapter, a summary of the entire thesis will be presented, followed by the conclusions drawn from the work, and suggestions on the direction future work should take.

7.1 Summary

This thesis began by offering introductory information on the functionality of computer networking and the communication protocols used every day in home networks. The problem of network security was then presented by looking at how the majority of common computer intrusions begin and the security strategies which have been developed to counter these network attacks. Background on embedded systems, and system on a programmable chip design was then offered before introducing the dynamic silicon firewall project as an embedded system created using system on a chip design tools.

After the introductory and background information had been presented, chapter 4 began by discussing the core hardware components of the dynamic silicon firewall. These components consist of the Ethernet controller, the CAM, the Nios® soft-core embedded processor, and the Verilog-synthesized firewall module. The rest of that chapter discussed the design and functionality of the custom HDL module, described as a finite state machine. This illustrated the manner in which information flows through the system by explaining the interaction between the Nios® processor, the Verilog filter module and the Ethernet controller. Chapter 5 completed the picture of the entire system by describing the software used in the design. The software consisted of both custom code, and code which was provided by Altera® corporation and modified for use with this design. The custom software components were given the responsibility of initializing the system and providing the dynamic aspect to the project. In chapter 6, two software analytical tools were introduced, followed by a discussion on how these tools could be used to monitor the dynamic silicon firewall system to prove functionality. The testing method of using the system to protect a simple web server was presented, followed by the results of monitoring the testbed.

7.2 Conclusion

The goal of this research was to investigate how an embedded hardware firewall could improve upon existing firewall technology. Furthermore, the goal was set to simplify the administration of the firewall for home users who are naive to network technology, while maintaining the security benefits provided by firewalls.

Test results have shown that the dynamic silicon firewall filters Ethernet network traffic at the network layer. Administration has been minimized by monitoring the user's network activity and basing the filter rules on whom the user chooses to contact. The ability to dynamically trust selected network hosts for limited periods of time adds a significant layer of protection from remote network attacks.

An analysis of the processing delay has shown that the dynamic silicon firewall requires approximately 3μs to filter a network packet. This value is dependent on the clock speed of the system, which was 33MHz for the system under test. The 3μs value represents a threefold increase in speed over the only previous embedded hardware firewall found in research, and a hundredfold increase in speed over a software firewall run on the Nios® embedded processor [1]. By increasing the clock speed, the dynamic silicon firewall should easily scale to network transfer speeds above the tested 100Mbps link speed.

It should be noted that there is an embedded hardware firewall on the market, intended for large corporate networks [2]. Unlike the dynamic silicon firewall, the 3Com® embedded hardware firewall uses filtering rules administered from a central location. This affords a central point of failure, which would lead to a network-wide compromise of security if the central computer administrating the filtering rules was accessed by unauthorized individuals. Also, since the filtering rules are not determined by user activity, a significant amount of setup is required. However, since this product is intended for corporations, the setup and maintenance required would be handled by information technology professionals employed by corporations. The 3Com® embedded hardware firewall is protected intellectual property, so no direct comparisons in performance could me made, and since it is targetted toward a different audience than the dynamic silicon firewall, it will not be discussed further.

The design of the dynamic silicon firewall required 138kB of memory and 3339 logic elements when synthesized on the development board. This represents a small amount of resources, 13% of the logic elements and 7% of the memory available on the Stratix® EP1S25F1020C5 FPGA. Therefore, if this design were developed as an ASIC product, it would result in a small device. Also, this design would be efficiently incorporated into an existing ASIC due to the small size of the dynamic silicon firewall.

This research has found a number of ways to improve upon existing network security technology [29]. This project filters network packets using dedicated hardware. A host computer protected by the dynamic silicon firewall instead of a software firewall would not experience general performance degradation during stressful network conditions such as those experienced during a denial of service attack. A software firewall running on a general purpose computer would be using the host processor's resources to filter packets. Secondly, the dynamic silicon firewall is designed to be in each host on a network, so there is no large concentration of financial resources required to manage a large network. This also eliminates the single point of failure problem, which could render a large number of systems unprotected if a network were only protected by a single hardware firewall. Another advantage to embedding the dynamic silicon firewall as close to the ethernet controller as possible is the reduction in queueing delay, which occurs when packets are concentrated at a single exit point to the external network because packet filtering requires time. Finally, since the firewall rules are dependant upon network use, there is no need to learn packet filtering syntax before being able to use the firewall. The dynamic silicon firewall offers a simple network security solution for non-tech-saavy internet users.

7.3 Future Work

Though the overall goals of this research were met, there are a number of improvements that would be required before this project could be included as part of a larger device. To be included on an Ethernet network interface card for use with a general purpose home computer, a bus interface and associated operating system drivers would be needed. When designing any future hardware boards containing the LAN91C111 and the dynamic silicon firewall, the ARDY signal should be used in conjunction with the asynchronous bus interface to the Ethernet chip. This would reduce the time it takes to filter a network packet by roughly one-third. Care should also be taken to limit the reliance of the final design on any software. For example, the dynamic aspect of the project is currently provided by a hardware timer whose interrupts trigger a software routine to remove IP addresses from CAM. If the Nios® processor were hacked, a malicious programmer could potentially cause early removal of IP addresses, leading to a denial of service attack. This is quite unlikely, since the Nios® processor does not run a real-time operating system, however the timer interrupt could be fed into another custom Verilog module, which then does the IP address removal. This would limit the ability to remove any individual IP addresses early, and further isolate the embedded firewall from the general purpose computer.

The dynamic silicon firewall could also be tailored for use with other systems, such as cellular phones or other network-capable devices. Viruses and malicious programs have been written to prove that portable network devices, such as cellular phones and personal digital assistants (PDAs), are susceptible to network attack. Since these devices have limited resources, using a software firewall to protect them would impact device performance. Including the dynamic silicon firewall in a PDA would be an effective solution to the problem of mobile device network security. This would likely require a number of changes to the dynamic silicon firewall system to operate with any wireless networking protocols being used by the device.

Administrative capabilities could be developed for information technology experts that may be using the dynamic silicon firewall as part of a larger layered network security approach. In this case, the dynamic silicon firewall would be distributed in each computer in a corporate network, and there would still likely be a large gateway firewall separating the internal network from the internet. Currently, packets which are filtered out by the dynamic silicon firewall are simply dropped. If this project was used as part of a large corporate network, it would be desirable to keep information such as how many packets have been dropped and where those dropped packets originated. An administrative access point could be created so that information technology experts could then retrieve this stored data for use in improving overall network security. This would also open up the ability to distribute or synchronize the firewall filtering rules amongst the different dynamic silicon firewalls in the local network.

Changes to the dynamic silicon firewall could be made to add filtering capabilities to different network layers. For example, the same technology being used for IP address filtering could be used for port filtering at the transport layer or MAC address filtering at the data link layer. This would offer additional capabilities, like restricting specific network applications such as instant messaging or web browsing. Due to the parallelism afforded by use of content-addressable memory, there would be little additional delay in adding these features, at the expense of a larger overall design in terms of the logic elements required to synthesize the project.

Finally, stateful packet filtering could be investigated to provide additional functionality. Stateless packet filters make filtering decisions by comparing the filter rules with the static header information contained in each network packet. Stateful packet filters add information based on communication history to the filtering decision. This could add a layer of protection when connection-oriented network protocols are in operation. When stateful communication is initiated with an outgoing packet, information about the source and destination sockets would be stored. Incoming packets would then be inspected to examine the connection state before a filtering decision would be made. This is needed for connection-oriented protocols, such as TCP, when connection state needs to be maintained, regardless of how often data packets are sent and received. In the current design of the dynamic silicon firewall state is not maintained, and if there is no constant communication, then the connection is broken when the IP addresses are flushed from the CAM. To implement stateful packet filtering, additional information would need to be stored, such as transport layer ports, connection state for connection-oriented protocols, and a time-stamp for each individual entry. This way, each entry would only be removed after the connection was closed, or after an individual pre-determined time interval had passed.

References

[1] J. Cheng. Silicon firewall prototype. Master's thesis, University of Saskatchewan, 2003.

[2] 3Com Corporation. *3Com Embedded Hardware Firewall Reviewer's Guide*, 101155-004 edition, March 2003.

[3] J. Kurose and K. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet.* Addison Wesley, 2001.

[4] J. Postel. *RFC 793: Transmission Control Protocol.* Technical report, IETF, September 1981.

[5] J. Postel. *RFC 768: User Datagram Protocol.* Technical report, IETF, August 1980.

[6] J. Postel. *RFC 791: Internet Protocol.* Technical report, IETF, September 1981.

[7] D. C. Plummer. *RFC 826: Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48-bit Ethernet address for transmission on Ethernet hardware.* Technical report, IETF, November 1982.

[8] *The Ethernet: A Local Area Network.* Technical report, Digital Equipment Corporation, Intel Corporation and Xerox Corporation, 1980.

[9] J. Erickson. *Hacking: The Art of Exploitation.* No Starch Press, Inc., 2003.

[10] L. Cheng. *Time to live on the network.* Technical report, Avantgarde Marketing & Design, 2004.

[11] E. Zwicky, S. Cooper, and D. Chapman. *Building Internet Firewalls.* O'Reilly & Associates Inc., 1995.

[12] D. Chapman. "Network (in)security through ip packet filtering,". *In Proceedings of USENIX UNIX Security Symposium '92*, 1992, pages 63–76, Baltimore, USA.

[13] A. Feldman and S. Muthukrishnan. "Tradeoffs for packet classification,". *In Proceedings of IEEE INFOCOM '00*, 2000, pages 1193–1202, Tel-Aviv, Israel.

[14] S. Wasti. Towards hardware-based packet filtering through the use of content addressable memory. Master's thesis, University of Saskatchewan, 2004.

[15] P. Marwedel. *Embedded System Design.* Kluwer Academic Publishers, 2003.

[16] E. Sutter. *Embedded Systems Firmware Demystified.* CMP Books, 2002.

[17] J. Labrosse. *Embedded Systems Building Blocks, Second Edition.* CMP Books, 2002.

[18] D. Perry. *VHDL: Programming by Example.* McGraw-Hill, 2002.

[19] D. Thomas and P. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1995.

[20] Altera Corporation. *Quartus II Development Software Handbook v5.1 (Complete Five-Volume Set)*, qii51015-5.1.1 edition, December 2005.

[21] Altera Corporation. *Stratix PCI Development Board Data Sheet*, ds-pcidvbd-1.0 edition, May 2003.

[22] Altera Corporation. *Plugs Ethernet Library Reference Manual*, mnl-sopcthrnt-1.2 edition, July 2003.

[23] J. Brelet. *Using Block RAM for High Performance Read/Write CAMs.* Xilinx Corporation, xapp204 v1.2 edition, May 2000.

[24] Altera Corporation. *TriMatrix Memory Selection Using the Quartus II Software*, an-207-2.1 edition, November 2002.

[25] Altera Corporation. *Nios 3.0 CPU Data Sheet*, ds-nioscpu-2.1 edition, October 2004.

[26] Altera Corporation. *Nios Embedded Processor Peripherals Reference Manual*, mnl-niosperiph-1.1 edition, April 2002.

[27] Standard Microsystems Corporation. *LAN91C111: 10/100 Non-PCI Ethernet Single Chip MAC + PHY Datasheet*, rev. 1.4 edition, December 2003.

[28] U. Lamping, R. Sharpe, and E. Warnicke. *Ethereal User's Manual*, 2005.

[29] D. Laturnas and R. Bolton. "Dynamic silicon firewall,". *In Proceedings of IEEE CCECE '05*, 2005, pages 304–307, Saskatoon, Canada.

Appendix A
Verilog Code

```
/*
sfw.v - Silicon Firewall
Darrell Laturnas, University of Saskatchewan
TRLabs, Saskatoon, Saskatchewan, Canada
*/


module sfw(clk, enablen, reset, nedk_int, tx_request,
   nios_nedk_ior, nios_nedk_iow, nios_nedk_addr,
   nios_cam_pattern, nios_cam_ctl, nios_bidir_data,
   nios_ram_ctl, ram_data_out, enet_data, mode_in, tx_allowed
   , enet_ior, enet_iow, enet_addr, nios_nedk_int, leds);

/*********************************************
* Inputs
*********************************************/

input clk; //system clock
input reset; //reset signal
input enablen; //enable active low signal
input mode_in; //nios mode signal
input nedk_int; //91c111 interrupt signal
input tx_request; //nios requesting it wants to transmit
input nios_nedk_ior; //nios nedk read pulse
input nios_nedk_iow; //nios nedk write pulse
input [2:0] nios_nedk_addr; //nios nedk address
input [31:0] nios_cam_pattern; //nios cam pattern
input [7:0]nios_cam_ctl;  //nios cam address, enable and
   read or write command
input [10:0] nios_ram_ctl; //nios ram enable, addresses

/*********************************************
* Bidirectional ports
*********************************************/

inout [15:0] nios_bidir_data; //enet data to nios
inout [15:0] enet_data; //91c111 data

/*********************************************
* Outputs
```

```
*********************************************/

output enet_ior; //91c111 read enable (active low)
output enet_iow; //91c111 write enable (active low)
output [2:0] enet_addr; //91c111 address
output nios_nedk_int; //interrupt nios thinks is from nedk
output tx_allowed; //allow nios to tx
output ram_data_out; //ram output for nios

/*********************************************
* Outputs for testing purposes
*********************************************/

output [1:0] leds;

/*********************************************
* Registers for Outputs
*********************************************/

reg enet_ior, enet_iow;
reg [2:0] enet_addr;
reg nios_nedk_int;
reg tx_allowed;
reg ram_data_out; //ram output for nios
reg [1:0] leds;

/*********************************************
* Registers for local use
*********************************************/

reg [31:0] cam_pattern; //pattern to write or read in CAM
reg [4:0] wraddress_sig; //cam write address
reg cam_cmd; //cam read when 0, write when 1
reg cam_en;      //cam clock enable
reg cam_mfound_d, cam_mfound_s; //cam match found
reg wrbusy_sig_d, wrbusy_sig_s; //cam write busy (might not
   use)

reg [15:0] data_to_91c111;         //sfw data to enet
reg [15:0] data_from_91c111; //sfw data from enet

reg [2:0] command_cycle; //hw mode states
reg [4:0] timing_cycle; //clock cycle counter for states
reg arp_rx_tx; //rx or tx occuring for ARP handling
reg irq_in_progress; //1 if an irq is in progress
reg [15:0] saved_bank; //saved bank select register
reg [15:0] saved_pointer; //saved pointer register
reg [15:0] ISR_data; //interrupt register
reg [5:0] last_packet_num; //number of last transmitted
   packet
reg [1:0] wait_return; //register for returning from wait
   state
reg sw_mode_flag; //flag to tell sfw to turn over to Nios

reg [2:0] sfw_nedk_addr; //sfw address for enet
reg sfw_nedk_ior, sfw_nedk_iow; //sfw nedk read/write pulses

reg sfw_cam_cmd, sfw_cam_en; //sfw cam enable and read or
   write command
reg [31:0] sfw_cam_pattern; //sfw cam pattern
reg [4:0] sfw_wraddress_sig; //sfw cam write address
reg [4:0] maddress_d, maddress_s;  //cam match address

reg sfw_ram_wren; //sfw ram write enable
reg [4:0] sfw_ram_wraddress; //sfw ram write address
reg ram_wren; //ram write enable
reg [4:0] ram_wraddress; //ram write address
reg ram_write_required;

/****************************************
* parameterize 91c111 register IDs so the code
* is more readable, or at least more "Altera-like"
* more information on 91c111 registers in datasheet
****************************************/

parameter mmu_reg = 3'd0;
parameter pnr_reg = 3'd1;
parameter fifo_reg = 3'd2;
parameter pointer_reg = 3'd3;
parameter data1_reg = 3'd4;
parameter data2_reg = 3'd5;
parameter interrupt_reg = 3'd6;
parameter bsr_reg = 3'd7;

// Hardware/Software Mode Parameters
parameter SW = 1'd0;
parameter HW = 1'd1;

/************************
* The Core
```

```
**************************/

always @ (posedge clk)
begin

if (enablen)    //firewall disabled
 begin
  tx_allowed <= 1;
  nios_nedk_int <= nedk_int;
  irq_in_progress <= 0;
  sw_mode_flag <= 0;
  command_cycle <= 0;
  timing_cycle <= 0;
  sfw_nedk_ior <= 1;
  sfw_nedk_iow <= 1;
  sfw_cam_pattern <= 0;
  sfw_cam_en <= 0;
  sfw_cam_cmd <= 0;
  sfw_ram_wren <= 0;
  sfw_ram_wraddress <= 0;
  leds[1] <= 1;
  leds[0] <= 1;
 end

else if ((nedk_int==1)&&(irq_in_progress==0)&&(mode_in==HW)
   &&(tx_allowed==0)&&(nios_nedk_int==0))
//new interrupt case occurs when ethernet interrupt occurs,
   there is no ethernet interrupt already being processed,
   the system is in hardware mode, and no communication is
   allowed between Nios and 91c111
 begin //start processing ethernet interrupt in case
    statements
  command_cycle <= 0;
  timing_cycle <= 0;
  irq_in_progress <= 1;
  leds[0] <= 1;
 end
else if (irq_in_progress==0)
 begin
// we're doing nothing, neither a new int, nor one in
   progress, this may mean software mode, or communication is
    already allowed between Nios and 91c111

// let's see if Nios wants to talk to the 91c111, if so, let
    it, since we are doing nothing with the 91c111
  if ((tx_request==1)&&(mode_in==HW))
   tx_allowed <= 1;
  else
   tx_allowed <= 0;

//if we're already in software mode, then the Nios is
   handling, or has handled, the most recent interrupt from
   the SFW, make sure we don't ask the Nios to check for
   interrupts needlessly
  if (mode_in==SW)
   nios_nedk_int <= 0;

//prepare system for next 91c111 interrupt
  command_cycle <= 0;
  timing_cycle <= 0;
  leds[0] <= 0;
 end
else //int in progress, so handle it
 begin
  case (command_cycle)

//save chip status, read interrupt to determine next state
  0:  begin
   timing_cycle = timing_cycle + 5'd1;
    case (timing_cycle)
    1: sfw_nedk_addr = bsr_reg;   //get bank select register
    2: sfw_nedk_ior = 0;
     //3: wait for valid read
    4:  begin
     sfw_nedk_ior = 1;
     data_to_91c111 = 16'd2;  //go to bank 2
     saved_bank = data_from_91c111;  //save bsr
     end
    5: sfw_nedk_iow = 0;   //write bsr to bank 2
    6: sfw_nedk_iow = 1;
    7: sfw_nedk_addr = pointer_reg; //get pointer register
    8: sfw_nedk_ior = 0;
     //9: wait for valid read
    10: begin
     saved_pointer = data_from_91c111;  //save pointer
     sfw_nedk_ior = 1;
     end
    11: sfw_nedk_addr = interrupt_reg;
    12: sfw_nedk_ior = 0;
     //13: wait for valid read
    14: begin
     ISR_data = data_from_91c111; //read int register
     sfw_nedk_ior = 1;
     end
    15: if (ISR_data[5:4] == 2'b11) //both overrrun & EPH ints
      begin
      data_to_91c111 = {ISR_data[15:8],8'h30};
      sfw_nedk_iow = 0; //ack the eph and overrun ints
      end
     else if (ISR_data[4] == 1'b1) //just overrrun int
      begin
      data_to_91c111 = {ISR_data[15:8],8'h10};
      sfw_nedk_iow = 0;    //ack the overrun int
      end
     else if (ISR_data[5] == 1'b1) //just EPH int
      begin
      data_to_91c111 = {ISR_data[15:8],8'h20};
      sfw_nedk_iow = 0;    //ack the eph int
      end
     else if (ISR_data[0] == 1'b1) //recieve int
      begin
      command_cycle = 1;
      timing_cycle = 0;
      end
     else if (ISR_data[1] == 1'b1) //transmit int
      begin
      command_cycle = 2;      //tx interrupt
      timing_cycle = 0;
      end
     else command_cycle = 3; //no int, restore chip
    16: begin     //should only get past 15 if eph or ovrn
     sfw_nedk_iow = 1;
     timing_cycle = 0;
     if (ISR_data[0] == 1'b1) //recieve int
      command_cycle = 1;
     else if (ISR_data[1] == 1'b1) //transmit int
      command_cycle = 2;      //tx interrupt
     else         // no int
      command_cycle = 3; //restore chip
     end
   endcase
  end

//read recieved packet status
```

```
1: begin
 timing_cycle = timing_cycle + 5'd1;
  case (timing_cycle)
  1:  begin
   sfw_nedk_addr = pointer_reg;
   data_to_91c111 = 16'hA000; //read from start rx area
   end
  2: sfw_nedk_iow = 0;
  3:  sfw_nedk_iow = 1;
  4: begin
   sfw_nedk_addr = data1_reg;  //where to get status from
   command_cycle = 5;     //wait for prefetch
   timing_cycle = 0;
   wait_return = 0;
   end
  5: sfw_nedk_ior = 0;
   //6: wait for valid read
  7: begin
   if (data_from_91c111 & 16'hAC00) //if there were ANY
      rcv errors
    begin
    command_cycle = 4; //release packet
    timing_cycle = 0;
    end
   else
    begin
    command_cycle = 6; //read ethertype
    timing_cycle = 0;
    arp_rx_tx = 0;
    end
   sfw_nedk_ior = 1;
   end
  8: sfw_nedk_iow = 0;  //write to pointer address
  9: sfw_nedk_iow = 1;
  10: begin
   sfw_nedk_addr = data1_reg;
   command_cycle = 5;     //wait for prefetch
   timing_cycle = 0;
   wait_return = 1;
   end
  11: sfw_nedk_ior = 0;
   //12: wait for valid read
  13: begin
   sfw_cam_pattern[31:16] = data_from_91c111; //upper ip
      word
```

```
     sfw_nedk_ior = 1;
     end
     //14: sfw_nedk_addr = data2_reg;
    15: sfw_nedk_ior = 0;
     //16: wait for valid read
    17: begin
     sfw_cam_pattern[15:0] = data_from_91c111; //lower ip word
     sfw_nedk_ior = 1;
     sfw_cam_cmd = 0; //set cam to read
     end
    18: sfw_cam_en = 1;  //enable cam clocks
    20: begin
     if ((cam_mfound_d==1)|(cam_mfound_s==1))
      begin
      //prepare turn over to nios
      sw_mode_flag = 1;
      command_cycle = 3; //restore chip status
      timing_cycle = 0;
      end
     else
      begin
      command_cycle = 4; //release packet
      timing_cycle = 0;
      end
     sfw_cam_en = 0;  //disable cam
     end
    endcase
   end

//read transmit packet number
  2:  begin
   timing_cycle = timing_cycle + 5'd1;
    case (timing_cycle)
    1: sfw_nedk_addr = fifo_reg;  //get tx pkt num
    2: sfw_nedk_ior = 0;
     //3: wait for valid read
    4:  begin
     last_packet_num = data_from_91c111[5:0];
     sfw_nedk_ior = 1;
     end
    5:  begin
     data_to_91c111 = {10'd0,last_packet_num[5:0]};
     sfw_nedk_addr = pnr_reg;
     end
    6: sfw_nedk_iow = 0;    //write pkt num to pnr
    7:  sfw_nedk_iow = 1;      //get tx status
    8: begin
     sfw_nedk_addr = pointer_reg;
     data_to_91c111 = 16'h2000;  //set pointer, read tx area
     end
    9: sfw_nedk_iow = 0;
    10: sfw_nedk_iow = 1;
    11: begin
     sfw_nedk_addr = data1_reg;
     command_cycle = 5;     //wait for prefetch
     timing_cycle = 0;
     wait_return = 2;
     end
    12: sfw_nedk_ior = 0;
     //13: wait for valid read
    14: begin
     if (data_from_91c111[0] == 1'b0)  //tx failed
      begin
      sfw_cam_cmd = 0; //make sure cam is in read mode
      timing_cycle = 28;//do the tx int ack (with cam read)
      end
     ram_write_required = 0;
     sfw_nedk_ior = 1;
     end
    15: begin //read tx ethertype
     command_cycle = 6;
     timing_cycle = 0;
     arp_rx_tx = 1;
     end
//put ip into cam for tx
    16: sfw_nedk_iow = 0;  //tx succeeded get ip
    17: sfw_nedk_iow = 1;
    18: begin
     sfw_nedk_addr = data1_reg;
     command_cycle = 5;     //wait for prefetch
     timing_cycle = 0;
     wait_return = 3;
     end
    19: sfw_nedk_ior = 0;
     //20: wait for valid read
    21: begin
     sfw_cam_pattern[31:16] = data_from_91c111; //upper word
     sfw_nedk_ior = 1;
     end
```

```
 //22: sfw_nedk_addr = data2_reg;
23: begin
 sfw_nedk_ior = 0;
 sfw_cam_cmd = 0; //ensure cam read mode
 end
 //24: wait for valid read
25: begin
 sfw_cam_pattern[15:0] = data_from_91c111;  //lower word
 sfw_nedk_ior = 1;
 end
26: sfw_cam_en = 1; //enable cam clock to check if ip is
    in cam
28: begin
 if ((cam_mfound_d==0)&(cam_mfound_s==0)) //IP not in
    CAM
  begin
  //we need to put IP in cam
  sfw_cam_cmd = 1; //write to cam
  //putting IP in RAM cache at same address as CAM
  sfw_ram_wraddress = wraddress_sig;
  ram_write_required = 1;
  end
 else //IP in CAM
  begin
   if (cam_mfound_d==1)
   begin
   //set ram address to match address of CAM
   sfw_ram_wraddress = maddress_d;
   ram_write_required = 1;
   end
  end
 end
29: begin
 sfw_nedk_addr = interrupt_reg; //goto int reg to ack TX
     int
 data_to_91c111 = {ISR_data[15:8],8'h02};//ack TX int
 end
30: begin
 sfw_cam_en = 0;  //needed max 2 clock cycles if a cam
    write
 sfw_nedk_iow = 0;
 if (ram_write_required==1)
  sfw_ram_wren = 1;
 end
31: begin
```

```
     sfw_nedk_iow = 1;
     sfw_ram_wren = 0;
     sfw_cam_cmd = 0;
     //prepare turn over to nios
     command_cycle = 3; //restore chip status
     timing_cycle = 0;
     end
    endcase
   end

//restore chip status
  3:  begin
   timing_cycle = timing_cycle + 5'd1;
    case (timing_cycle)
    1:  begin  //restore pointer
     sfw_nedk_addr = pointer_reg;
     data_to_91c111 = saved_pointer;
     end
    2: sfw_nedk_iow = 0;
    3:  sfw_nedk_iow = 1;
    4: begin
     sfw_nedk_addr = bsr_reg; //restore bank
     data_to_91c111 = saved_bank;
     end
    5: sfw_nedk_iow = 0;
    6:  begin
     sfw_nedk_iow = 1;
      if (sw_mode_flag == 1)
      nios_nedk_int=1;
     end
    7: begin
     sw_mode_flag = 0;
     irq_in_progress = 0;
     timing_cycle = 0;
     command_cycle = 0;
     end
    endcase
   end

//release packet
  4:  begin
   timing_cycle = timing_cycle + 5'd1;
    case (timing_cycle)
    1:  begin
     sfw_nedk_addr = mmu_reg;
     data_to_91c111 = 16'h0080;  //issue remove & release
     end
    2: sfw_nedk_iow = 0;  //write r&r
    3:  sfw_nedk_iow = 1;
    //wait a couple clock cycles between polling
    6:  sfw_nedk_ior = 0;  //read busy bit
     //7: wait for valid read
    8: begin
     if (data_from_91c111[0]) //then busy
      timing_cycle = 4;
     sfw_nedk_ior = 1;
     end
    9: begin
     command_cycle = 3; //after remove and release is
        restore chip
     timing_cycle = 0;
     end
    endcase
   end

//wait state for nios data prefetch (370ns total, 12 clocks
   here though)
  5:  begin
   timing_cycle = timing_cycle + 5'd1;
    case (timing_cycle)
    12: if(wait_return==2'd0)
      begin
      command_cycle = 1; //rx status
      timing_cycle = 4;
      end
     else if(wait_return==2'd1)
      begin
      command_cycle = 1; //rx ip
      timing_cycle = 10;
      end
     else if(wait_return==2'd2)
      begin
      command_cycle = 2; //tx status
      timing_cycle = 11;
      end
     else
      begin
      command_cycle = 2; //tx ip
      timing_cycle = 18;
      end
    endcase
   end

//wait state for reading ethertype
  6:  begin
   timing_cycle = timing_cycle + 5'd1;
    case (timing_cycle)
    1: begin //set pointer to read ethertype
     sfw_nedk_addr = pointer_reg;

     if (arp_rx_tx==0)
      begin
      data_to_91c111 = 16'hA010; //rx
      end
     else
      begin
      data_to_91c111 = 16'h2010; //tx
      end

     end
    2: sfw_nedk_iow = 0;
    3: sfw_nedk_iow = 1;
    //4-15: wait for pre-fetch
    16: sfw_nedk_addr = data1_reg;
    17: sfw_nedk_ior = 0;
     //18: wait for valid read
    19: begin
     sfw_nedk_ior = 1;
     if (arp_rx_tx == 0)  //receive
      begin
      //set pointer for source ip
      if (data_from_91c111 == 16'h0608)
       begin
       sfw_nedk_addr = pointer_reg; //ARP
       data_to_91c111 = 16'hE020;
       command_cycle = 1;
       timing_cycle = 7;

       //prepare turn over to nios for ARP pkt
       //sw_mode_flag = 1;
       //command_cycle = 3; //restore chip status
       //timing_cycle = 0;
       end
      else
       begin
       sfw_nedk_addr = pointer_reg; //IP
       data_to_91c111 = 16'hE01E;
       command_cycle = 1;
       timing_cycle = 7;
       end
      end
     else     //transmit
      begin
      if (data_from_91c111 == 16'h0608)
       begin
       //command_cycle = 2;     //ARP
       //timing_cycle = 28;
       sfw_nedk_addr = pointer_reg;
       data_to_91c111 = 16'h602A;
       command_cycle = 2;
       timing_cycle = 15;
       end
      else
       begin
       sfw_nedk_addr = pointer_reg; //IP
       data_to_91c111 = 16'h6022;
       command_cycle = 2;
       timing_cycle = 15;
       end
      end
     end
    endcase
   end
  endcase
 end
end

//instantiate the module that writes sequential addresses
   for the CAM
cam_addresser ca_inst(
 .reset ( enablen ),
 .wraddress ( sfw_wraddress_sig ),
 .sfw_cam_cmd ( sfw_cam_cmd )
 );

//instantiate the dynamic cam
sfwcam dyn_cam_inst (
 .pattern ( cam_pattern ),
 .wraddress ( wraddress_sig ),
 .wren ( cam_cmd ),
 .inclock ( ~clk ),
 .inclocken ( cam_en ),
 .outclock ( clk ),
 .outclocken ( cam_en ),
 .maddress ( maddress_d ),
 .mfound ( cam_mfound_d ),
 .wrbusy ( wrbusy_sig_d )
 );

//instantiate the static cam for testing
sfwcam stat_cam_inst (
 .pattern ( cam_pattern ),
 .wraddress ( wraddress_sig ),
 .wren ( nios_cam_ctl[7] ),
 .inclock ( ~clk ),
 .inclocken ( cam_en ),
 .outclock ( clk ),
 .outclocken ( cam_en ),
 .maddress ( maddress_s ),
 .mfound ( cam_mfound_s ),
 .wrbusy ( wrbusy_sig_s )
 );

//instantiate the ram, nios & sfw need write access
ip_ram ip_ram_inst(
 .data ( mode_in ), //just because we want data=1 when HW
    mode and 0 during SW mode it works out well
 .wraddress ( ram_wraddress ),
 .rdaddress ( nios_ram_ctl[4:0] ),
 .wren ( ram_wren ),
 .clock ( ~clk ),
 .q ( ram_data_out )
 );

//instantiate the glue logic (muxes and such)

glue_logic glue_logic_inst (
 .sfw_nedk_ior ( sfw_nedk_ior ),
 .nios_nedk_ior ( nios_nedk_ior ),
 .enet_ior ( enet_ior ),
 .sfw_nedk_iow ( sfw_nedk_iow ),
 .nios_nedk_iow ( nios_nedk_iow ),
 .enet_iow ( enet_iow ),
 .sfw_nedk_addr ( sfw_nedk_addr ),
 .nios_nedk_addr ( nios_nedk_addr ),
 .enet_addr ( enet_addr ),
 .sfw_wraddress_sig ( sfw_wraddress_sig ),
 .nios_wraddress_sig ( nios_cam_ctl[4:0] ),
 .wraddress_sig ( wraddress_sig ),
 .data_from_91c111 ( data_from_91c111 ),
 .data_to_91c111 ( data_to_91c111 ),
 .nios_bidir_data ( nios_bidir_data ),
 .enet_data ( enet_data ),
 .mode ( mode_in ),
 .sfw_cam_pattern ( sfw_cam_pattern ),
 .nios_cam_pattern ( nios_cam_pattern ),
 .cam_pattern ( cam_pattern ),
 .sfw_cam_cmd ( sfw_cam_cmd ),
 .nios_cam_cmd ( nios_cam_ctl[5] ),
 .cam_cmd ( cam_cmd ),
 .sfw_cam_en ( sfw_cam_en ),
 .nios_cam_en ( nios_cam_ctl[6] ),
 .cam_en ( cam_en ),
 .sfw_ram_wraddress ( sfw_ram_wraddress ),
 .nios_ram_wraddress ( nios_ram_ctl[9:5] ),
 .ram_wraddress ( ram_wraddress ),
 .sfw_ram_wren ( sfw_ram_wren ),
 .nios_ram_wren ( nios_ram_ctl[10] ),
 .ram_wren ( ram_wren ),
 .clk (clk)
 );

endmodule

/*
cam_addresser.v - a simple module to increment the address
   used when storing IP addresses in the CAM
Darrell Laturnas, University of Saskatchewan
TRLabs, Saskatoon, Saskatchewan, Canada
*/

module cam_addresser (reset, wraddress, sfw_cam_cmd);

input reset;
input sfw_cam_cmd;
output [4:0] wraddress;
reg [4:0] wraddress;

//changing "negedge wrbusy" to "posedge sfw_cam_cmd"
```

```
always @(posedge reset or posedge sfw_cam_cmd)

if (reset)
        wraddress = 0;
else
        wraddress = wraddress + 5'd1;

endmodule


/*
ip_ram.v - a dual-port ram used in providing the dynamic
   aspect of the SFW project, built with the Altera ram
   lpm_ram_dp using the megafunction wizard
Darrell Laturnas, University of Saskatchewan
TRLabs, Saskatoon, Saskatchewan, Canada
*/

// megafunction wizard: %LPM_RAM_DP%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: lpm_ram_dp

module ip_ram (
        data,
        wraddress,
        rdaddress,
        wren,
        clock,
        q);

        input   [0:0]   data;
        input   [4:0]   wraddress;
        input   [4:0]   rdaddress;
        input     wren;
        input     clock;
        output  [0:0]   q;

        wire [0:0] sub_wire0;
        wire [0:0] q = sub_wire0[0:0];

        lpm_ram_dp      lpm_ram_dp_component (
                                .rdclock (clock),
                                .wren (wren),
                                .wrclock (clock),
                                .data (data),
                                .rdaddress (rdaddress),
                                .wraddress (wraddress),
                                .q (sub_wire0));
        defparam
                lpm_ram_dp_component.lpm_width = 1,
                lpm_ram_dp_component.lpm_widthad = 5,
                lpm_ram_dp_component.rden_used = "FALSE",
                lpm_ram_dp_component.intended_device_family
                   = "APEX20KE",
                lpm_ram_dp_component.lpm_type = "LPM_RAM_DP"
                   ,
                lpm_ram_dp_component.lpm_indata = "
                   REGISTERED",
                lpm_ram_dp_component.lpm_wraddress_control =
                    "REGISTERED",
                lpm_ram_dp_component.lpm_rdaddress_control =
                    "REGISTERED",
                lpm_ram_dp_component.lpm_outdata = "
                   UNREGISTERED",
                lpm_ram_dp_component.use_eab = "ON";


endmodule

//
   ============================================================

// CNX file retrieval info
//
   ============================================================

// Retrieval info: PRIVATE: WidthData NUMERIC "1"
// Retrieval info: PRIVATE: WidthAddr NUMERIC "5"
// Retrieval info: PRIVATE: Clock NUMERIC "0"
// Retrieval info: PRIVATE: rden NUMERIC "0"
// Retrieval info: PRIVATE: UseDPRAM NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "
   APEX20KE"
// Retrieval info: PRIVATE: REGdata NUMERIC "1"
// Retrieval info: PRIVATE: REGwraddress NUMERIC "1"
// Retrieval info: PRIVATE: REGwren NUMERIC "1"
// Retrieval info: PRIVATE: REGrdaddress NUMERIC "1"
// Retrieval info: PRIVATE: REGrren NUMERIC "1"
// Retrieval info: PRIVATE: REGq NUMERIC "0"
// Retrieval info: PRIVATE: enable NUMERIC "0"
// Retrieval info: PRIVATE: CLRdata NUMERIC ”0”
// Retrieval info: PRIVATE: CLRwraddress NUMERIC ”0”
// Retrieval info: PRIVATE: CLRwren NUMERIC ”0”
// Retrieval info: PRIVATE: CLRrdaddress NUMERIC ”0”
// Retrieval info: PRIVATE: CLRrren NUMERIC ”0”
// Retrieval info: PRIVATE: CLRq NUMERIC ”0”
// Retrieval info: PRIVATE: BlankMemory NUMERIC ”1”
// Retrieval info: PRIVATE: MIFfilename STRING ””
// Retrieval info: PRIVATE: UseLCs NUMERIC ”0”
// Retrieval info: CONSTANT: LPM_WIDTH NUMERIC ”1”
// Retrieval info: CONSTANT: LPM_WIDTHAD NUMERIC ”5”
// Retrieval info: CONSTANT: RDEN_USED STRING ”FALSE”
// Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING ”
   APEX20KE”
// Retrieval info: CONSTANT: LPM_TYPE STRING ”LPM_RAM_DP”
// Retrieval info: CONSTANT: LPM_INDATA STRING ”REGISTERED”
// Retrieval info: CONSTANT: LPM_WRADDRESS_CONTROL STRING ”
   REGISTERED”
// Retrieval info: CONSTANT: LPM_RDADDRESS_CONTROL STRING ”
   REGISTERED”
// Retrieval info: CONSTANT: LPM_OUTDATA STRING ”
   UNREGISTERED”
// Retrieval info: CONSTANT: USE_EAB STRING ”ON”
// Retrieval info: USED_PORT: data 0 0 1 0 INPUT NODEFVAL
   data[0..0]
// Retrieval info: USED_PORT: q 0 0 1 0 OUTPUT NODEFVAL q
   [0..0]
// Retrieval info: USED_PORT: wraddress 0 0 5 0 INPUT
   NODEFVAL wraddress[4..0]
// Retrieval info: USED_PORT: rdaddress 0 0 5 0 INPUT
   NODEFVAL rdaddress[4..0]
// Retrieval info: USED_PORT: wren 0 0 0 0 INPUT VCC wren
// Retrieval info: USED_PORT: clock 0 0 0 0 INPUT NODEFVAL
   clock
// Retrieval info: CONNECT: @data 0 0 1 0 data 0 0 1 0
// Retrieval info: CONNECT: q 0 0 1 0 @q 0 0 1 0
// Retrieval info: CONNECT: @wraddress 0 0 5 0 wraddress 0 0
   5 0
// Retrieval info: CONNECT: @rdaddress 0 0 5 0 rdaddress 0 0
   5 0
// Retrieval info: CONNECT: @wren 0 0 0 0 wren 0 0 0 0
// Retrieval info: CONNECT: @wrclock 0 0 0 0 clock 0 0 0 0
// Retrieval info: CONNECT: @rdclock 0 0 0 0 clock 0 0 0 0
// Retrieval info: LIBRARY: lpm lpm.lpm_components.all
```

```
/*
sfwcam2.v - this module creates a 32 entry, 32-bit CAM. It
   does this by connecting together  16 entry, 5-bit CAM
   modules which, in turn, are created using dual port RAM
   instantiated from the Altera ram altsyncram component.
   More information on the method used can be found in the
   Xilinx document ”xapp204.pdf”
Darrell Laturnas, University of Saskatchewan
TRLabs, Saskatoon, Saskatchewan, Canada
*/

module sfwcam2(clk0,clken0,clk1,clken1,pattern,wrpos,wren,
   writeerase_,pos,mfound);
input          clk0,clken0,clk1,clken1;
input   [31:0] pattern;
input   [4:0]  wrpos;
input          wren,writeerase_;
output  [31:0] pos;
output  mfound;

wire  [15:0]  p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13;

cam16x5  cam0(clk0,clken0,clk1,clken1,pattern[4:0],wrpos
   [3:0],wren&~wrpos[4],writeerase_,p0[15:0]);

cam16x5  cam1(clk0,clken0,clk1,clken1,pattern[9:5],wrpos
   [3:0],wren&~wrpos[4],writeerase_,p1[15:0]);

cam16x5  cam2(clk0,clken0,clk1,clken1,pattern[14:10],wrpos
   [3:0],wren&~wrpos[4],writeerase_,p2[15:0]);

cam16x5  cam3(clk0,clken0,clk1,clken1,pattern[19:15],wrpos
   [3:0],wren&~wrpos[4],writeerase_,p3[15:0]);

cam16x5  cam4(clk0,clken0,clk1,clken1,pattern[24:20],wrpos
   [3:0],wren&~wrpos[4],writeerase_,p4[15:0]);

cam16x5  cam5(clk0,clken0,clk1,clken1,pattern[29:25],wrpos
   [3:0],wren&~wrpos[4],writeerase_,p5[15:0]);

cam16x5  cam6(clk0,clken0,clk1,clken1,pattern[34:30],wrpos
   [3:0],wren&~wrpos[4],writeerase_,p6[15:0]);

cam16x5  cam7(clk0,clken0,clk1,clken1,pattern[4:0],wrpos
   [3:0],wren&wrpos[4],writeerase_,p7[15:0]);
```

```
cam16x5 cam8(clk0,clken0,clk1,clken1,pattern[9:5],wrpos
   [3:0],wren&wrpos[4],writeerase_,p8[15:0]);

cam16x5 cam9(clk0,clken0,clk1,clken1,pattern[14:10],wrpos
   [3:0],wren&wrpos[4],writeerase_,p9[15:0]);

cam16x5 cam10(clk0,clken0,clk1,clken1,pattern[19:15],wrpos
   [3:0],wren&wrpos[4],writeerase_,p10[15:0]);

cam16x5 cam11(clk0,clken0,clk1,clken1,pattern[24:20],wrpos
   [3:0],wren&wrpos[4],writeerase_,p11[15:0]);

cam16x5 cam12(clk0,clken0,clk1,clken1,pattern[29:25],wrpos
   [3:0],wren&wrpos[4],writeerase_,p12[15:0]);

cam16x5 cam13(clk0,clken0,clk1,clken1,pattern[34:30],wrpos
   [3:0],wren&wrpos[4],writeerase_,p13[15:0]);

assign pos={p13&p12&p11&p10&p9&p8&p7,p6&p5&p4&p3&p2&p1&p0};
assign mfound=pos[0]|pos[1]|pos[2]|pos[3]|pos[4]|pos[5]|pos
   [6]|pos[7]|pos[8]|pos[9]|pos[10]|pos[11]|pos[12]|pos[13]|
   pos[14]|pos[15]|pos[16]|pos[17]|pos[18]|pos[19]|pos[20]|
   pos[21]|pos[22]|pos[23]|pos[24]|pos[25]|pos[26]|pos[27]|
   pos[28]|pos[29]|pos[30]|pos[31];
endmodule

module cam16x5(clk0,clken0,clk1,clken1,pattern,wrpos,wren,
   writeerase_,pos);
input         clk0,clken0,clk1,clken1;
input  [4:0]  pattern;
input  [3:0]  wrpos;
input         wren,writeerase_;
output [15:0] pos;

        altsyncram      altsyncram_component (
                               .clocken0 (clken0),
                               .clocken1(clken1),
                               .wren_a (wren),
                               .clock0 (clk0),
                               .clock1 (clk1),
                               .address_a ({pattern,wrpos})
                                  ,
                               .address_b (pattern),
                               .data_a (writeerase_),
                              .q_b (pos));
        defparam
                altsyncram_component.operation_mode = ”
                   DUAL_PORT”,
                altsyncram_component.width_a = 1,
                altsyncram_component.widthad_a = 9,
                altsyncram_component.numwords_a = 512,
                altsyncram_component.width_b = 16,
                altsyncram_component.widthad_b = 5,
                altsyncram_component.numwords_b = 32,
                altsyncram_component.lpm_type = ”altsyncram”
                   ,
                altsyncram_component.width_byteena_a = 1,
                altsyncram_component.width_byteena_b = 1,
                altsyncram_component.outdata_reg_b = ”
                   UNREGISTERED”,
                altsyncram_component.outdata_reg_a = ”
                   UNREGISTERED”,
                altsyncram_component.indata_aclr_a = ”NONE”,
                altsyncram_component.wrcontrol_aclr_a = ”
                   NONE”,
                altsyncram_component.address_aclr_a = ”NONE”
                   ,
                altsyncram_component.address_reg_b = ”CLOCK1
                   ”,
                altsyncram_component.address_aclr_b = ”NONE”
                   ,
                altsyncram_component.outdata_aclr_b = ”NONE”
                   ,
                altsyncram_component.
                   read_during_write_mode_mixed_ports = ”
                   DONT_CARE”,
                altsyncram_component.ram_block_type = ”M512”
                   ,
                altsyncram_component.intended_device_family
                   = ”Stratix”;

endmodule

/*
glue_logic.v - this module instantiates all of the
   interconnect logic used to connect the Nios CPU, silicon
   firewall module, CAM and Lan91c111 ethernet chip
Darrell Laturnas, University of Saskatchewan
TRLabs, Saskatoon, Saskatchewan, Canada
```

```
*/


module glue_logic(sfw_nedk_ior, nios_nedk_ior, enet_ior,
   sfw_nedk_iow, nios_nedk_iow, enet_iow, sfw_nedk_addr,
   nios_nedk_addr, enet_addr,sfw_wraddress_sig,
   nios_wraddress_sig, wraddress_sig, data_from_91c111,
   data_to_91c111, nios_bidir_data, enet_data, mode,
   sfw_cam_pattern, nios_cam_pattern, cam_pattern,
   sfw_cam_cmd, nios_cam_cmd, cam_cmd, sfw_cam_en,
   nios_cam_en, cam_en, sfw_ram_wraddress, nios_ram_wraddress
   , ram_wraddress,sfw_ram_wren, nios_ram_wren, ram_wren, clk
   );

input sfw_nedk_ior, nios_nedk_ior;                          //
   read strobes
input sfw_nedk_iow, nios_nedk_iow;                   //write
   strobes
input [2:0] sfw_nedk_addr, nios_nedk_addr;           //enet
   addresses
input [15:0] data_to_91c111;                                //
   data going to 91c111
input mode;
                     //HW or SW mode
input sfw_cam_cmd, nios_cam_cmd;                            //
   cam read or write command
input sfw_cam_en, nios_cam_en;                              //
   cam clock enable
input [31:0] sfw_cam_pattern, nios_cam_pattern;
           //pattern for CAM
input [4:0] sfw_wraddress_sig, nios_wraddress_sig;
           //cam write address
input [4:0]      sfw_ram_wraddress, nios_ram_wraddress;
           //ram write address
input sfw_ram_wren, nios_ram_wren;
                              //ram write enable
input clk;


inout [15:0] nios_bidir_data;                        //nios
   bidirectional data line
inout [15:0] enet_data;                                     //91
   c111 data

output [15:0] data_from_91c111;                      //data
```

```
   coming from 91c111
output enet_ior;
           //91c111 read enable (active low)
output enet_iow;
           //91c111 write enable (active low)
output [2:0] enet_addr;                                          //91
   c111 address
output cam_en, cam_cmd;                                          //
   cam enable and read or write command
output [31:0] cam_pattern;                                       //
   pattern to write or read in CAM
output [4:0] wraddress_sig;                                      //
   cam write address
output [4:0] ram_wraddress;                                      //
   ram write address
output ram_wren;
           //ram write enable

reg delay_sig, delay_sfw_nedk_iow;

wire [31:0]      w_data_from_91c111;

always @ (negedge clk)
begin
        if (sfw_nedk_iow == 0)
                delay_sig = 0;
        else
                delay_sig = 1;
end

and and_inst (delay_sfw_nedk_iow, delay_sig, sfw_nedk_iow);

io_mux  io_mux_inst_ram_wren (
        .data1 ( sfw_ram_wren ),
        .data0 ( nios_ram_wren ),
        .sel ( mode ),
        .result ( ram_wren )
        );

cam_addr_mux ram_wraddr_mux_inst(
        .data1x ( sfw_ram_wraddress ),
        .data0x ( nios_ram_wraddress ),
        .sel ( mode ),
        .result( ram_wraddress )
        );
```

```
io_mux  io_mux_inst_nedk_ior (
        .data1 ( sfw_nedk_ior ),
        .data0 ( nios_nedk_ior ),
        .sel ( mode ),
        .result ( enet_ior )
        );

io_mux  io_mux_inst_nedk_iow (
        .data1 ( sfw_nedk_iow ),
        .data0 ( nios_nedk_iow ),
        .sel ( mode ),
        .result ( enet_iow )
        );

io_mux  io_mux_inst_cam_cmd (
        .data1 ( sfw_cam_cmd ),
        .data0 ( nios_cam_cmd ),
        .sel ( mode ),
        .result ( cam_cmd )
        );

io_mux  io_mux_inst_cam_en (
        .data1 ( sfw_cam_en ),
        .data0 ( nios_cam_en ),
        .sel ( mode ),
        .result ( cam_en )
        );

cam_addr_mux cam_addr_mux_inst(
        .data1x ( sfw_wraddress_sig ),
        .data0x ( nios_wraddress_sig ),
        .sel ( mode ),
        .result( wraddress_sig )
        );

cam_pattern_mux cam_pattern_mux_inst(
        .data1x ( sfw_cam_pattern ),
        .data0x ( nios_cam_pattern ),
        .sel ( mode ),
        .result ( cam_pattern )
        );

nedk_addr_mux   nedk_addr_mux_inst (
        .data1x ( sfw_nedk_addr ),
        .data0x ( nios_nedk_addr ),
        .sel ( mode ),
        .result ( enet_addr )
        );

data_latch data_latch_inst (
        .data ( w_data_from_91c111 ),
        .gate ( ~sfw_nedk_ior ),
        .q ( data_from_91c111 )
        );

nedk_bus_mux    nedk_bus_mux_inst_sfw (
        .data ( data_to_91c111 ),
        .enabledt ( ~delay_sfw_nedk_iow ),
        .enabletr ( ~sfw_nedk_ior ),
        .tridata ( enet_data ),
        .result ( w_data_from_91c111 )
        );

nedk_bus_mux    nedk_bus_mux_inst_nios (
        .data ( nios_bidir_data ),
        .enabledt ( ~nios_nedk_iow ),
        .enabletr ( ~nios_nedk_ior ),
        .tridata ( enet_data ),
        .result ( nios_bidir_data )
        );

endmodule


/*
data_latch.v - a data latch used to latch the data outputs
   of the 91c111 chip, built with the Altera lpm_latch
   function using the megafunction wizard
Darrell Laturnas, University of Saskatchewan
TRLabs, Saskatoon, Saskatchewan, Canada
*/

// megafunction wizard: %LPM_LATCH%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: lpm_latch

module data_latch (
        data,
        gate,
        q);

        input   [31:0]  data;
        input     gate;
        output  [31:0]  q;

        wire [31:0] sub_wire0;
        wire [31:0] q = sub_wire0[31:0];

        lpm_latch       lpm_latch_component (
                                .data (data),
                                .gate (gate),
                                .q (sub_wire0));
        defparam
                lpm_latch_component.lpm_width = 32,
                lpm_latch_component.lpm_type = "LPM_LATCH";


endmodule

//
   ============================================================

// CNX file retrieval info
//
   ============================================================

// Retrieval info: PRIVATE: nBit NUMERIC "32"
// Retrieval info: PRIVATE: aclr NUMERIC "0"
// Retrieval info: PRIVATE: aset NUMERIC "0"
// Retrieval info: CONSTANT: LPM_WIDTH NUMERIC "32"
// Retrieval info: CONSTANT: LPM_TYPE STRING "LPM_LATCH"
// Retrieval info: USED_PORT: data 0 0 32 0 INPUT NODEFVAL
   data[31..0]
// Retrieval info: USED_PORT: q 0 0 32 0 OUTPUT NODEFVAL q
   [31..0]
// Retrieval info: USED_PORT: gate 0 0 0 0 INPUT NODEFVAL
   gate
// Retrieval info: CONNECT: @data 0 0 32 0 data 0 0 32 0
// Retrieval info: CONNECT: q 0 0 32 0 @q 0 0 32 0
// Retrieval info: CONNECT: @gate 0 0 0 0 gate 0 0 0 0
// Retrieval info: LIBRARY: lpm lpm.lpm_components.all

/*
io_mux.v - a multiplexer for individual signal lines, built
```

```
   with the Altera mux lpm_mux using the megafunction wizard
Darrell Laturnas, University of Saskatchewan
TRLabs, Saskatoon, Saskatchewan, Canada
*/

// megafunction wizard: %LPM_MUX%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: lpm_mux

module io_mux (
        data1,
        data0,
        sel,
        result);

        input      data1;
        input      data0;
        input      sel;
        output     result;

        wire [0:0] sub_wire0;
        wire [0:0] sub_wire1 = sub_wire0[0:0];
        wire  result = sub_wire1;
        wire  sub_wire2 = sel;
        wire  sub_wire3 = sub_wire2;
        wire  sub_wire4 = data0;
        wire  sub_wire6 = data1;
        wire [1:0] sub_wire5 = {sub_wire6, sub_wire4};

        lpm_mux lpm_mux_component (
                                .sel (sub_wire3),
                                .data (sub_wire5),
                                .result (sub_wire0));
        defparam
                lpm_mux_component.lpm_size = 2,
                lpm_mux_component.lpm_widths = 1,
                lpm_mux_component.lpm_width = 1,
                lpm_mux_component.lpm_type = "LPM_MUX";


endmodule

//
   ============================================================
// CNX file retrieval info
//
   ================================================================

// Retrieval info: CONSTANT: LPM_SIZE NUMERIC "2"
// Retrieval info: CONSTANT: LPM_WIDTHS NUMERIC "1"
// Retrieval info: CONSTANT: LPM_WIDTH NUMERIC "1"
// Retrieval info: CONSTANT: LPM_TYPE STRING "LPM_MUX"
// Retrieval info: USED_PORT: result 0 0 0 0 OUTPUT NODEFVAL
    result
// Retrieval info: USED_PORT: data1 0 0 0 0 INPUT NODEFVAL
   data1
// Retrieval info: USED_PORT: data0 0 0 0 0 INPUT NODEFVAL
   data0
// Retrieval info: USED_PORT: sel 0 0 0 0 INPUT NODEFVAL sel
// Retrieval info: CONNECT: result 0 0 0 0 @result 0 0 1 0
// Retrieval info: CONNECT: @data 0 0 1 1 data1 0 0 0 0
// Retrieval info: CONNECT: @data 0 0 1 0 data0 0 0 0 0
// Retrieval info: CONNECT: @sel 0 0 1 0 sel 0 0 0 0
// Retrieval info: LIBRARY: lpm lpm.lpm_components.all

/*
nedk_addr_mux.v - a multiplexer for the Ethernet address
   lines, built with the Altera mux lpm_mux using the
   megafunction wizard
Darrell Laturnas, University of Saskatchewan
TRLabs, Saskatoon, Saskatchewan, Canada
*/

// megafunction wizard: %LPM_MUX%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: lpm_mux

module nedk_addr_mux (
        data1x,
        data0x,
        sel,
        result);

        input   [2:0]  data1x;
        input   [2:0]  data0x;
        input     sel;
        output  [2:0]  result;
        wire [2:0] sub_wire0;
        wire [2:0] result = sub_wire0[2:0];
        wire  sub_wire1 = sel;
        wire  sub_wire2 = sub_wire1;
        wire [2:0] sub_wire3 = data0x[2:0];
        wire [2:0] sub_wire5 = data1x[2:0];
        wire [5:0] sub_wire4 = {sub_wire5, sub_wire3};

        lpm_mux lpm_mux_component (
                                .sel (sub_wire2),
                                .data (sub_wire4),
                                .result (sub_wire0));
        defparam
                lpm_mux_component.lpm_size = 2,
                lpm_mux_component.lpm_widths = 1,
                lpm_mux_component.lpm_width = 3,
                lpm_mux_component.lpm_type = "LPMMUX";


endmodule

//
   ============================================================

// CNX file retrieval info
//
   ============================================================

// Retrieval info: CONSTANT: LPM_SIZE NUMERIC "2"
// Retrieval info: CONSTANT: LPM_WIDTHS NUMERIC "1"
// Retrieval info: CONSTANT: LPM_WIDTH NUMERIC "3"
// Retrieval info: CONSTANT: LPM_TYPE STRING "LPMMUX"
// Retrieval info: USED_PORT: result 0 0 3 0 OUTPUT NODEFVAL
    result[2..0]
// Retrieval info: USED_PORT: data1x 0 0 3 0 INPUT NODEFVAL
   data1x[2..0]
// Retrieval info: USED_PORT: data0x 0 0 3 0 INPUT NODEFVAL
   data0x[2..0]
// Retrieval info: USED_PORT: sel 0 0 0 0 INPUT NODEFVAL sel
// Retrieval info: CONNECT: result 0 0 3 0 @result 0 0 3 0
// Retrieval info: CONNECT: @data 0 0 3 3 data1x 0 0 3 0
// Retrieval info: CONNECT: @data 0 0 3 0 data0x 0 0 3 0
// Retrieval info: CONNECT: @sel 0 0 1 0 sel 0 0 0 0
// Retrieval info: LIBRARY: lpm lpm.lpm_components.all
```

```
/*
nedk_bus_mux.v - a tri-state multiplexer for the Ethernet
   data lines, built with the Altera mux lpm_mux using the
   megafunction wizard
Darrell Laturnas, University of Saskatchewan
TRLabs, Saskatoon, Saskatchewan, Canada
*/

// megafunction wizard: %LPM_BUSTRI%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: lpm_bustri

module nedk_bus_mux (
        data,
        enabledt,
        enabletr,
        tridata,
        result);

        input   [15:0]  data;
        input     enabledt;
        input     enabletr;
        inout   [15:0]  tridata;
        output  [15:0]  result;

        wire [15:0] sub_wire0;
        wire [15:0] result = sub_wire0[15:0];

        lpm_bustri      lpm_bustri_component (
                                .tridata (tridata),
                                .enabletr (enabletr),
                                .enabledt (enabledt),
                                .data (data),
                                .result (sub_wire0));
        defparam
                lpm_bustri_component.lpm_width = 16,
                lpm_bustri_component.lpm_type = "LPM_BUSTRI"
                   ;


endmodule

//
   ============================================================
// CNX file retrieval info
//
   ============================================================

// Retrieval info: PRIVATE: nBit NUMERIC "16"
// Retrieval info: PRIVATE: BiDir NUMERIC "1"
// Retrieval info: CONSTANT: LPM_WIDTH NUMERIC "16"
// Retrieval info: CONSTANT: LPM_TYPE STRING "LPM_BUSTRI"
// Retrieval info: USED_PORT: tridata 0 0 16 0 BIDIR
   NODEFVAL tridata[15..0]
// Retrieval info: USED_PORT: data 0 0 16 0 INPUT NODEFVAL
   data[15..0]
// Retrieval info: USED_PORT: enabledt 0 0 0 0 INPUT
   NODEFVAL enabledt
// Retrieval info: USED_PORT: result 0 0 16 0 OUTPUT
   NODEFVAL result[15..0]
// Retrieval info: USED_PORT: enabletr 0 0 0 0 INPUT
   NODEFVAL enabletr
// Retrieval info: CONNECT: tridata 0 0 16 0 @tridata 0 0 16
   0
// Retrieval info: CONNECT: @data 0 0 16 0 data 0 0 16 0
// Retrieval info: CONNECT: @enabledt 0 0 0 0 enabledt 0 0 0
   0
// Retrieval info: CONNECT: result 0 0 16 0 @result 0 0 16 0
// Retrieval info: CONNECT: @enabletr 0 0 0 0 enabletr 0 0 0
   0
// Retrieval info: LIBRARY: lpm lpm.lpm_components.all

/*
cam_addr_mux.v - a multiplexer for the cam address lines,
   built with the Altera mux lpm_mux using the megafunction
   wizard
Darrell Laturnas, University of Saskatchewan
TRLabs, Saskatoon, Saskatchewan, Canada
*/

// megafunction wizard: %LPM_MUX%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: lpm_mux

module cam_addr_mux (
        data1x,
        data0x,
        sel,
        result);

        input   [4:0]  data1x;
        input   [4:0]  data0x;
        input     sel;
        output  [4:0]  result;

        wire [4:0] sub_wire0;
        wire [4:0] result = sub_wire0[4:0];
        wire  sub_wire1 = sel;
        wire  sub_wire2 = sub_wire1;
        wire [4:0] sub_wire3 = data0x[4:0];
        wire [4:0] sub_wire5 = data1x[4:0];
        wire [9:0] sub_wire4 = {sub_wire5, sub_wire3};

        lpm_mux lpm_mux_component (
                                .sel (sub_wire2),
                                .data (sub_wire4),
                                .result (sub_wire0));
        defparam
                lpm_mux_component.lpm_size = 2,
                lpm_mux_component.lpm_widths = 1,
                lpm_mux_component.lpm_width = 5,
                lpm_mux_component.lpm_type = ”LPM_MUX”;


endmodule

//
   ================================================================

// CNX file retrieval info
//
   ================================================================

// Retrieval info: CONSTANT: LPM_SIZE NUMERIC ”2”
// Retrieval info: CONSTANT: LPM_WIDTHS NUMERIC ”1”
// Retrieval info: CONSTANT: LPM_WIDTH NUMERIC ”5”
// Retrieval info: CONSTANT: LPM_TYPE STRING ”LPM_MUX”
// Retrieval info: USED_PORT: result 0 0 5 0 OUTPUT NODEFVAL
    result[4..0]
// Retrieval info: USED_PORT: data1x 0 0 5 0 INPUT NODEFVAL
   data1x[4..0]
// Retrieval info: USED_PORT: data0x 0 0 5 0 INPUT NODEFVAL
```

```
   data0x[4..0]
// Retrieval info: USED_PORT: sel 0 0 0 0 INPUT NODEFVAL sel
// Retrieval info: CONNECT: result 0 0 5 0 @result 0 0 5 0
// Retrieval info: CONNECT: @data 0 0 5 5 data1x 0 0 5 0
// Retrieval info: CONNECT: @data 0 0 5 0 data0x 0 0 5 0
// Retrieval info: CONNECT: @sel 0 0 1 0 sel 0 0 0 0
// Retrieval info: LIBRARY: lpm lpm.lpm_components.all

/*
cam_pattern_mux.v - a multiplexer for the cam pattern lines,
     built with the Altera mux lpm_mux using the megafunction
   wizard
Darrell Laturnas, University of Saskatchewan
TRLabs, Saskatoon, Saskatchewan, Canada
*/

// megafunction wizard: %LPM_MUX%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: lpm_mux

module cam_pattern_mux (
        data1x,
        data0x,
        sel,
        result);

        input   [31:0]  data1x;
        input   [31:0]  data0x;
        input     sel;
        output  [31:0]  result;

        wire [31:0] sub_wire0;
        wire [31:0] result = sub_wire0[31:0];
        wire  sub_wire1 = sel;
        wire  sub_wire2 = sub_wire1;
        wire [31:0] sub_wire3 = data0x[31:0];
        wire [31:0] sub_wire5 = data1x[31:0];
        wire [63:0] sub_wire4 = {sub_wire5, sub_wire3};

        lpm_mux lpm_mux_component (
                                .sel (sub_wire2),
                                .data (sub_wire4),
                                .result (sub_wire0));
        defparam
```

```
                lpm_mux_component.lpm_size = 2,
                lpm_mux_component.lpm_widths = 1,
                lpm_mux_component.lpm_width = 32,
                lpm_mux_component.lpm_type = ”LPM_MUX”;


endmodule

//
  ============================================================

// CNX file retrieval info
//
  ============================================================

// Retrieval info: CONSTANT: LPM_SIZE NUMERIC ”2”
// Retrieval info: CONSTANT: LPM_WIDTHS NUMERIC ”1”
// Retrieval info: CONSTANT: LPM_WIDTH NUMERIC ”32”
// Retrieval info: CONSTANT: LPM_TYPE STRING ”LPM_MUX”
// Retrieval info: USED_PORT: result 0 0 32 0 OUTPUT
  NODEFVAL result[31..0]
// Retrieval info: USED_PORT: data1x 0 0 32 0 INPUT NODEFVAL
    data1x[31..0]
// Retrieval info: USED_PORT: data0x 0 0 32 0 INPUT NODEFVAL
    data0x[31..0]
// Retrieval info: USED_PORT: sel 0 0 0 0 INPUT NODEFVAL sel
// Retrieval info: CONNECT: result 0 0 32 0 @result 0 0 32 0
// Retrieval info: CONNECT: @data 0 0 32 32 data1x 0 0 32 0
// Retrieval info: CONNECT: @data 0 0 32 0 data0x 0 0 32 0
// Retrieval info: CONNECT: @sel 0 0 1 0 sel 0 0 0 0
// Retrieval info: LIBRARY: lpm lpm.lpm_components.all
```

Appendix B
C Code

```
/*
Silicon Firewall
Darrell Laturnas, University of Saskatchewan
TRLabs, Saskatoon, Saskatchewan, Canada
*/

//SFW pio manipulations for testing

#include "excalibur.h"

//parameterizations

#define CAM_READ        0xDF    //use &=
#define CAM_WRITE_D     0x20    //use |=
#define CAM_WRITE_S 0x80        //use |=
#define CAM_ENABLE      0x40    //use |=
#define CAM_DISABLE     0x00    //use &=

#define RAM_WRITE       0x400   //use |=

// prototypes

void sfw_initialize_cam(void);
void sfw_initialize_ram(void);
void sfw_tx_request(void);
int sfw_get_mode(void);
void sfw_set_mode(int mode);
void cam_flush_isr(int context);
void sfw_initialize_timer(void);

// +--------------------------
// sfw_initialize_cam
//
// put important ip addresses in CAM
//
// after testing, may still need to initialize with IPs
// dhcp on boot-up for example
void sfw_initialize_cam(void)
{
        int j;
```

```
//disable firewall
        na_enablen_pio->np_piodata = 1;

// set to software mode and talk to cam
        na_mode_out->np_piodata = 0;

// set to read mode and disable CAM
        na_cam_ctl_pio->np_piodata = CAM_DISABLE;

// set CAM pattern to zero
        na_cam_pattern->np_piodata = 0;

// loop through and initialize the CAM with empty entries
        for(j = 0; j < 32; j++)
        {

//set cam address, write and enable
        na_cam_ctl_pio->np_piodata = (j|CAM_WRITE_D|
           CAM_WRITE_S|CAM_ENABLE);

//disable clock to finish each write
        na_cam_ctl_pio->np_piodata = CAM_DISABLE;
        }

// insert some IP addresses into CAM for testing

// Ron's computer's IP address in HEX
        na_cam_pattern->np_piodata = 0xa8c02381;

// CAM address chosen for no reason, write and enable
        na_cam_ctl_pio->np_piodata = (25|CAM_WRITE_D|
           CAM_ENABLE);

//disable clock to finish each write
        na_cam_ctl_pio->np_piodata = CAM_DISABLE;

//Mike Mitzel's computer
        na_cam_pattern->np_piodata = 0xa8c03d81;

// CAM address chosen for no reason, write and enable
        na_cam_ctl_pio->np_piodata = (26|CAM_WRITE_D|
           CAM_ENABLE);

//disable clock
```

```
        na_cam_ctl_pio->np_piodata = CAM_DISABLE;

//my computer's IP
        na_cam_pattern->np_piodata = 0xa8c07981;

// CAM address chosen for no reason, write and enable
        na_cam_ctl_pio->np_piodata = (29|CAM_WRITE_S|
            CAM_ENABLE);

//disable clock
        na_cam_ctl_pio->np_piodata = CAM_DISABLE;

//dhcp server's IP
        na_cam_pattern->np_piodata = 0xa8c06981;

// CAM address chosen for no reason, write and enable
        na_cam_ctl_pio->np_piodata = (30|CAM_WRITE_D|
            CAM_ENABLE);

//disable clock
        na_cam_ctl_pio->np_piodata = CAM_DISABLE;

//broadcast packets
        na_cam_pattern->np_piodata = 0xffffffff;

// CAM address chosen for no reason, write and enable
        na_cam_ctl_pio->np_piodata = (31|CAM_WRITE_D|
            CAM_ENABLE);

//disable clock
        na_cam_ctl_pio->np_piodata = CAM_DISABLE;

//set CAM pattern to back to zero
        na_cam_pattern->np_piodata = 0;

//set hardware mode
        na_mode_out->np_piodata = 1;

//enable firewall
        na_enablen_pio->np_piodata = 0;
        na_tx_request->np_piodata = 0;
        //printf(”enabling firewall \n”);
}

// +------------------------------
// sfw_initialize_ram
//
// initialize RAM with zeros
// after testing this can be done in hardware
void sfw_initialize_ram(void)
{
        int j;

//disable firewall
        na_enablen_pio->np_piodata = 1;

//set to software mode and talk to ram
        na_mode_out->np_piodata = 0;

// initialize the RAM with zeros        in loop
        for(j = 0; j < 32; j++)
        {

// set ram data and enable
        na_ram_ctl_pio->np_piodata = ((j<<5)|RAM_WRITE);

//finish ram write
        na_ram_ctl_pio->np_piodata = 0;
        }

//set hardware mode
        na_mode_out->np_piodata = 1;

//enable firewall
        na_enablen_pio->np_piodata = 0;
        na_tx_request->np_piodata = 0;
}

// +----------------------------
// sfw_tx_request
//
// requests access to LAN91C111 for Nios
void sfw_tx_request(void)
{

//if we're in hardware mode, request the ability
//to transmit and do nothing until sfw says we can
        if (na_mode_in->np_piodata)
        {
                while (na_tx_allowed->np_piodata != 1)
                {
                na_tx_request->np_piodata = 1;
                }
        }

// stop tx request because it's been answered
// set mode to software mode
        na_tx_request->np_piodata = 0;
        sfw_set_SW_mode();
}

// +-----------------------------
// sfw_get_mode
//
// read if system is in hardware or software mode
int sfw_get_mode(void)
{
        int result;
        result = na_mode_in->np_piodata;
        return result;
}

// +-----------------------------
// sfw_set_mode
//
// set system into hardware or software mode
void sfw_set_mode(int mode)
{
        na_mode_out->np_piodata = mode;
}

// +-----------------------------
// sfw_initialize_timer
//
// set up interrupt service routine and
// start timer
void sfw_initialize_timer(void)
{
        int context = 0;

//connect the cam_flush_isr routine to the timer interrupt
        nr_installuserisr(na_timer2_irq, cam_flush_isr,
           context);

//start the timer
```

```
        na_timer2->np_timercontrol = 0x0007;
}

// +------------------------------
// cam_flush_isr
//
// interrupt service routine for
// dynamic removal of IPs from CAM
void cam_flush_isr(int context)
{
        int j, temp, mode;

//get current mode
        mode = sfw_get_mode();

//if HW mode, ask to go to SW mode
        if(mode)
                sfw_tx_request();

//set CAM delete value of 0
        na_cam_pattern->np_piodata = 0;

// set CAM to read mode and disable clock
        na_cam_ctl_pio->np_piodata = CAM_DISABLE;

// loop through every RAM address
        for(j = 0; j < 32; j++)
        {

// get data value at current ram address
        na_ram_ctl_pio->np_piodata = j;
        temp = na_ram_data_in->np_piodata;

// if current RAM data is 1,
// set entry to 0 to mark IP address as stale
                if (temp)
                {
                        //write 0 to address j
                        na_ram_ctl_pio->np_piodata = ((j<<5)
                           |RAM_WRITE);
                        na_ram_ctl_pio->np_piodata = 0;
                }
                else
                {
                        //delete stale cam entry
                        na_cam_ctl_pio->np_piodata = (j|
                            CAM_WRITE_D|CAM_ENABLE);
                        na_cam_ctl_pio->np_piodata =
                            CAM_DISABLE;
                        printf("Deleted entry %d \n", j);
                }
        }

// ensure CAM is in read mode and disable clock
        na_cam_ctl_pio->np_piodata = CAM_DISABLE;

//restore previous mode
        if(mode)
                sfw_set_mode(mode);

//clear timer interrupt
        na_timer2->np_timerstatus = 0;
}

// end of file
```