SILICON FIREWALL PROTOTYPE

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

Jin Cheng

ABSTRACT

The Internet is a technological advance that provides access to information, and the ability to publish information, in revolutionary ways. There is also a major danger that provides the ability to corrupt and destroy information as well. When a computer is connected to the Internet, three things are put at risk: the data storage, the computing resources and the user's reputation. In order to balance the advantages and risks, the contact between a computer and the Internet or the contact between different networks should be controlled carefully.

A firewall is a form of protection that allows a network to connect to the Internet or to another network while maintaining a degree of security. The firewall is an effective type of network security, and in most situations, it is the most effective tool for doing that.

With the availability of larger bandwidth, it is becoming more and more difficult for traditional software firewalls to function over a high-speed connection. In addition, the advances in network hardware technology, such as routers, and new applications of firewalls have caused the software firewall to be an impediment to high throughput. This network bottleneck leads to the requirement for new solutions to balance performance and security. Replacing software with hardware could lead to improved performance, enabling the firewalls to handle significantly larger amounts of data.

The goal of this project is to investigate if and how existing desktop computer firewall technology could be improved by replacing software functionality with hardware (i.e., silicon). A hardware-based Silicon Firewall system has been designed by choosing the appropriate architecture and implemented using Altera FPGA (Field Programmable Gate Array) on a SOPC (System On a Programmable Chip) Board. The performance of the Silicon Firewall is tested and compared with the software firewall.

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation and gratitude to my supervisor, Dr. Ron. Bolton, for his guidance, support and encouragement throughout this project and preparation of this thesis.

I would also like to thank the management and staff of Telecommunications Research Laboratories (TRLabs) for providing financial assistance and the use of their facilities during my research.

Finally, I would like to express my special thanks to my husband, Xiang Li, thanks for your love and patience. Also, my special thanks go to my family. My parents have always been supportive in my education and always been there when I need them.

DEDICATION

To my parents, Guangan Cheng and Suting Yang, and my brother, Yang Cheng.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ALU	Arithmetic logic unit
ARM	Advanced RISC Machines
ARP	Address Resolution Protocol
ARPANET	Advanced Research Projects Agency Network
ASIC	Application Specific Integrated Circuit
ASSP	Application Specific Standard Product
AUI	Attachment Unit Interface
BNC	British Naval Connector
CAM	Content Addressable Memory
CPU	Central Processing Unit
DA	Destination Address
DDN	Defense Data Network
DMA	Direct Memory Access
DOD	Department of Defense
DSP	Digital Signal Processing
EDA	Electronic Design Automation
EDK	Ethernet Development Kit
EEPROM	Electrically Erasable Programmable Read-Only Memory
ESB	Embedded System Blocks
FCS	Frame Check Sequence
FPGA	Field Programmable Gate Array
GERMS	G—Go, E—Erase flash, R—Relocate next download,

	M—Memory set and dump, S—S-records
HDK	Hardware Development Kit
HDL	Hardware Description Language
ICMP	Internet Control Message Protocol
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IHL	Internet Header Length
I/O	Input/Output
IP	Internet Protocol
IRQ	Interrupt
ISA	Industry Standard Architecture
ISO	International Organization for Standardization
ISQ	Interrupt Status Queue
JTAG	Joint Test Action Group
LAN	Local Area Network
LCD	Liquid Crystal Display
LE	Logic Elements
LED	Light-Emitting Diode
LLC	Logical Link Control
MAC	Media Access Control
MHz	Megahertz
MIPS	Million Instructions Per Second
NDK	Nios Development Kit

NEDK	Nios Ethernet Development Kit
NRZ	Non-Return to Zero
OS	Operating system
OSI	Open Systems Interconnection
PAR	Positive Acknowledgment with Re-transmission
PBM	Peripheral Bus Module
PC	Personal Computer
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PHY	Physical
PIO	Parallel Input Output
PLD	Programmable Logic Device
PMC	PCI Mezzanine Card
PROM	Programmable Read-Only Memory
RAM	Random Access Memory
RFC	Requests For Comment
RISC	Reduce Instruction Set Computer
ROM	Read Only Memory
RTOS	Real-Time Operating System
SA	Source Address
SDK	Software Development Kit
SDRAM	Synchronous Dynamic Random Access Memory
SFD	Start-of-Frame Delimiter

SOC	System-On-a-Chip
SODIMM	Small Outline Dual In-line Memory Module
SOPC	System-On-a-Programmable-Chip
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SSRAM	Synchronous Static Random Access Memory
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
VHDL	Very High Speed Integrated Circuit Hardware Description
WAN	Wide-Area Network

Chapter 1 INTRODUCTION

In this chapter, the Internet security problem is introduced and different kinds of risk types and attack types are discussed. A number of security models are presented. The firewall is introduced as the most effective method. Finally, Silicon Firewall feasibility and the objective of this research project are introduced.

1.1 Internet Security

The Internet is a technological advance that provides access to information, and the ability to publish information, in revolutionary ways [1]. The benefits include, but are not limited to, information gathering, customer service, and improved publicity. With the rapid development of the information superhighway, millions of people are exchanging information through the Internet.

But it’s also a major danger that provides the ability to pollute and destroy information. This means the Internet is a two-edge sword and, because it influences millions of people, the risks are obviously high on the Internet.

1.1.1 Risk Types

When computers are connected to the Internet, three things are put at risk:

- The data stored on the computers
- The computer system resources
- The user’s reputation

a. Data Storage

    1) Privacy: keeping it confidential.

    2) Integrity: preventing it from being changed by other people.

    3) Availability: accessing it freely.

b. Computing Resources

    Computing resources will not be wasted or destroyed if they are not used, because they are neither natural resources, nor are they limited resources. Since people spend money and time on their computing resources, they should have the right to determine their computers are used.

c. User’s Reputation

    Sometimes an intruder appears on the Internet with another person’s identity, and anything that is done appears to come from the stolen identity. Generally, people who choose to do this aim for maximum impact, rather than believability. However even if only a few people believe it, it will take a long time to clean up your reputation, and it can be humiliating. Even if an intruder doesn’t use another person’s identity, unauthorized access to computers is not good for any organization, because it shakes people’s confidence in that organization. In addition, most intruders will attempt to go from an organization they broke in to others. This is going to make their next victim think that the breached organization is a platform for computer criminals.

1.1.2 Attack Types

There are many types of attacks [1], and many ways of categorizing them. In this section, the attacks are broken down into three basic categories:

- Intrusion
- Denial of service
- Information theft

a. Intrusion

   Intrusions are the most common attacks on systems. Intruders are actually able to control your computers as if they were legitimate users.

b. Denial of Service

   A denial of service attack is such an attack that aimed entirely at preventing you from using your computer.

c. Information Theft

   Usually these attacks exploit Internet services that are intended to give out information, including modifications to give out more information than was intended, or to give information to unauthorized users. The attacker can obtain information without ever having to directly use your computers.

1.2 Security Models

There are a variety of security models to protect against the kinds of attacks outlined in the previous section, ranging from no security through obscurity, host security, to network security.

No security model can solve all the problems. Why bother then? Security may not prevent every single incident, but it can keep an incident from seriously damaging or even shutting down the whole system.

1.2.1 Security through Obscurity

With this model, a system is presumed to be secure provided nobody knows about it.

This approach seldom works for long time because there are too many ways to find an attractive target.

Many people presume that even though attackers can find them, they won't bother to. They think that a small company or a home machine just isn't going to be of interest to intruders. But the fact is that many intruders aren't aiming at particular targets; their purpose is just breaking into as many machines as possible. They look at small companies and home machines as easy targets. Even if they won't stay long, they will attempt to break in and they may do substantial damage if they do get in as they try to cover their tracks.

A site has to do at least a minimal amount of registration in order to function on any network including the Internet. Intruders watch out for new connections, hoping that these sites won't yet have security measures in place.

The number of ways that someone can determine security-sensitive information is amazing. For example, knowing the hardware and software and the version of the operating system gives the intruders important clues about what security holes they might try. They can often get this information from the host registration, or by trying to connect to the computer. Sometime, an intruder even doesn't need to access the computer to get it, since many computers disclose their type of operating system in the greeting you get before you log in.

1.2.2 Host Security

Host security probably is the most common model for computer security [1]. With this model, the security of each host machine is enforced separately, and the effort is made to avoid or alleviate all the known security problems that might affect that particular host. However, the host security doesn’t scale to large numbers of machines, even though it does works on individual machines.

The complexity and diversity are the major impediment to effective host security in modern computing environments. Machines from multiple vendors are included in most modern environments, each with its own operating system and its own set of security problems. Even if the site has machines from only one vendor, the security problems for different releases of the same operating system could be significantly different. Also, there could be a problem even if all these machines are from a single vendor and run a single release of the operating system, since different configurations (different services enabled, and so on) can bring different subsystems into play (and into conflict) and lead to different sets of security problems.

Further more, even if the machines are completely identical, the sheer number of them at some sites can make securing all of them difficult. Thus, effectively implementing and maintaining host security takes a significant amount of upfront and ongoing work. Even if all that work has been done correctly, host security still often fails due to bugs in vendor software, or due to a lack of suitable software for some required functions.

Also, host security relies on both the good intentions and the skill of everyone who has privileged access to any machine. With the increasing number of machines, the number of privileged users generally increases as well. Compared to attaching a machine to a network, securing a machine is much more difficult, therefore insecure machines may appear on your network as unexpected surprises.

A host security model may be more appropriate for small sites, or sites with extreme security requirements. Indeed, some level of host security should be included in all the site’s overall security plans. Even if a network security model is adopted, certain systems will benefit from the strongest host security. The problem is that making the host security model work requires too many restrictions and too many people, so this model alone isn’t cost-effective for any but small or simple sites.

1.2.3 Network Security

A network security model tries to control network access to local hosts and their services, rather than secure them individually. Network security models involve building firewalls to protect internal systems and networks and using strong authentication approaches and encryption to protect particularly sensitive data as it transits over the network. This model is an efficient and effective method to secure a larger and more diverse computer environment.

1.3 Firewalls

A firewall is a component or a set of components that restrict access between a protected network and the Internet, or between sets of networks [1]. It is a very effective type of network security. There are two kinds of firewalls: Internet firewalls and internal firewalls.

1.3.1 Internet Firewalls

A firewall is designed to keep a fire from spreading from one part of the building to another during building construction. A similar purpose is served by an Internet firewall. In theory: It prevents the dangers of the Internet from spreading to the internal network. In practice, an Internet firewall serves multiple purposes:

- Restrict people to entering at a carefully controlled point.
- Restrict people to leaving at a carefully controlled point.

- Prevent attackers from getting close to your other defenses.

An Internet firewall is often installed at the point where the protected internal network connects to the Internet, as shown in Figure 1.1.

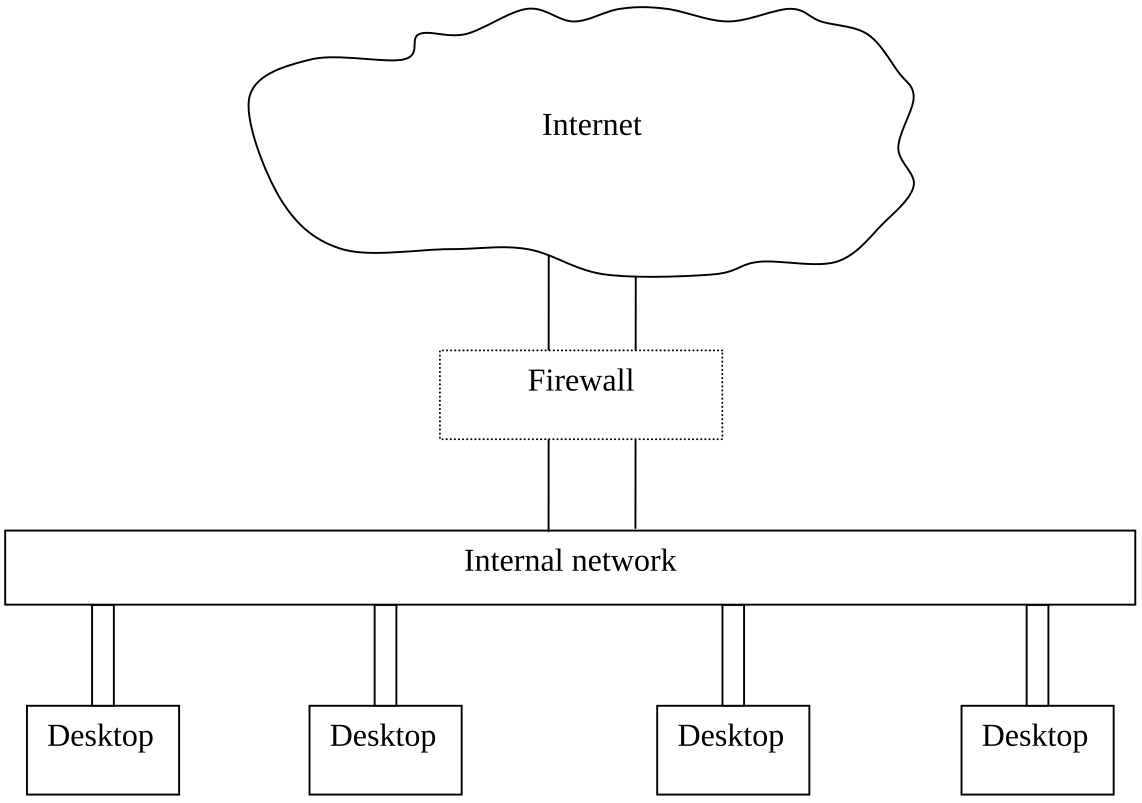


Figure 1.1 Internet Firewall

All traffic coming from the Internet or going out from the internal network passes through the firewall. Consequently, the firewall has the opportunity to ensure that the traffic -- email, file transfer, remote logins, or any kinds of interaction between specific systems -- conform to the security policy. These security policies are set by each site; some are highly restrictive and others fairly open.

1.3.2 Internal Firewalls

In some situations, parts of the internal network need to be protected from other parts. There are a number of reasons to do this:

- There are test or lab networks with unpredictable things going on there.
- There are networks that are less secure than the rest of the site.
- There are networks that are more secure than the rest of the site.

Firewalls are a useful technology in another situation. In some cases, a firewall sits between two parts of the same organization, or between two separate organizations that share a network, rather than between a single organization and the Internet. This kind of firewall is called an *internal firewall.* Figure 1.2 shows an internal firewall architecture within a laboratory network.

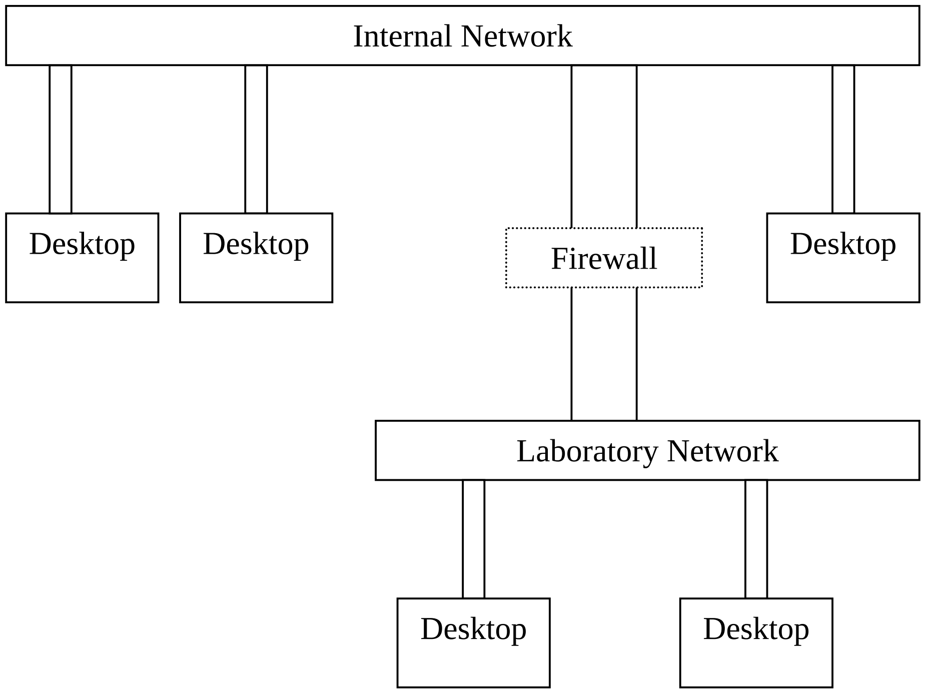


Figure 1.2 Internal Firewall

Logically, a firewall is a restrictor, a separator, and an analyzer [1]. There are various ways to configure this equipment; the configuration will depend upon a site's particular security policy, budget, and overall operation.

1.4 Firewall Capability

Firewalls can do a lot for site's security, but they also have some drawbacks.

1. What can a firewall do?
    a. A firewall is a focus for security decisions

        A firewall is a checkpoint through which all traffic in and out must pass. It lets you concentrate network security on this checkpoint where your network connects to the Internet. Focusing your security in this way is much more efficient than spreading security decisions and technologies around individual machines.

    b. A firewall can enforce security policy

        Many of the services that people need from the Internet are inherently not secure. The firewall enforces the site's security policy, allowing only services defined by the security rules to pass through.

    c. A firewall can log Internet activity efficiently

        Because all traffic passes through the firewall, the single point of access, the firewall can record the interactions between the protected network and the external network.

    d. A firewall limits your exposure

        A firewall will sometimes be used to keep one section of your site's network from another section to prevent the problems that impact one section from spreading through the entire network (internal firewall).

2. What can’t a firewall do?

   Although firewalls provide protection against most network threats, they are not a complete security solution. Certain threats are beyond the control of the firewall:

   a. A firewall can’t protect you against malicious insiders

      An inside user can attack your systems without passing through the firewall.

   b. A firewall can’t protect against connections that don’t go through it

      A firewall can’t control the traffic that doesn’t pass through it.

   c. A firewall can’t protect against new threats

      A firewall is designed to protect against known threats. However, no firewall can automatically defend every new threat that arises. You can’t set up a firewall once and expect it to protect you forever.

   d. A firewall can’t protect against viruses

      Firewalls can’t keep viruses out of a protected network. Firewalls scan all incoming traffic to determine whether it is allowed to pass through to the internal network. Nevertheless, the scanning is based on the source and destination addresses and port numbers, not on the content of data.

      As a consequence, you need to augment the firewall by incorporating function separation, physical security, host security, and user education into your overall security plan.

Given the limitations above, why bother to install a firewall? Because a firewall is the most effective way to connect a network to the Internet while maintaining a degree of security. And in most situations, it's the most effective tool for doing that.

1.5 Silicon Firewall Feasibility

With the availability of bandwidth, it is becoming more and more difficult for a traditional software firewall to function with a high-speed connection. In addition, the advances in network hardware technology, such as routers and new applications of firewalls have caused the software firewall to be an impediment to high throughput. This network bottleneck leads to the requirement of new solutions to balance the performance and security. Replacing software with a hardware (silicon) firewall could lead to improved performance, enabling the firewall to handle significantly larger amount of data.

1.6 Research Objectives

The goal of this project is to investigate if and how existing desktop computer firewall technology could be improved by replacing software functionality with hardware (silicon). A microprocessor-based system has been designed by choosing the appropriate architecture and implemented using Altera FPGA (Field Programmable Gate Array) on a SOPC (System on a Programmable Chip) Board. Furthermore, the performance of the Silicon Firewall is tested and compared with the software firewall.

1.7 Thesis Organization

This thesis is organized into seven chapters: In Chapter 2, packet filtering is presented. In Chapter 3, hardware software system codesign is described. In Chapter 4, the Altera Nios embedded processor is introduced. NDK (Nios Development Kit) and NEDK

(Nios Ethernet Development Kit) are also presented. In Chapter 5, The Silicon Firewall design is introduced in detail. Two major peripherals, the CAM (Content Addressable Memory) and the CS8900A are introduced as well as the two operating modes of the Silicon Firewall, hardware mode and software mode. In Chapter 6, testing and results are explained. In Chapter 7, a summary of this research is presented and future work is suggested.

Chapter 2 INTERNET FIREWALL SECURITY SYSTEMS

Basic Internet firewall security strategies are presented in this chapter. Packet filtering is introduced as the method selected by this research project to build the Silicon Firewall. For a good understanding of packet filtering, TCP/IP fundamentals are also presented.

2.1 Security Strategies

In this section some basic strategies employed in building firewalls and in enforcing security will be introduced.

2.1.1 Least Privilege

Least privilege is perhaps the most fundamental principle of any kind of security, not just computer and network security. Basically, the principle of least privilege means that any object (user, administrator, program, system, etc.) should have only the privilege the object needs to perform its assigned tasks—and no more [1]. Least privilege is an important principle for limiting the exposure to attacks and for limiting the damage caused by particular attacks.

For example, probably not every user needs to access every Internet service, probably not every user needs to modify (or even read) every file on the system, probably not every user needs to know the machine’s root password, probably not every system administrator needs to know the root password for all the system, probably not every system needs to access every other system’s files.

In order to apply the principle of least privilege, ways should be explored to reduce the privilege required for various operations.

2.1.2 Defense in Depth

Defense in depth is another principle of any security system. It means not depending on just one security mechanism, instead, install multiple mechanisms that back each other up. The advantage of defense in depth is that the failure of any single security mechanism will not totally compromise the system.

Any security system—even the most seemingly impenetrable firewall—can be breached by attackers who are willing to take enough risk [1]. Multiple mechanisms can be adopted to provide backup and redundancy for each other: network security, host security and human security. All of these mechanisms are important and can be highly effective.

2.1.3 Choke Point

A choke point forces attackers to use a narrow channel that can be monitored and controlled. In network security, the firewall between the Internet and the internal network is such a choke point (assuming that it's the only connection between the Internet and the internal network). Any attacker who's going to attack the internal network from the Internet is going to have to come through that channel.

A choke point is useless if there's an effective way for an attacker to go around it [1]. In this way, chances are that an adequate job of defending any of the avenues of attack can't be done, or someone will slip through one while another is being defended.

2.1.4 Weakest Link

A fundamental property of security is that a chain is only as strong as its weakest linkand a wall is only as strong as its weakest point. The weak points are always hunted by smart attackers, so attention should be paid evenly to all aspects of the security system.

2.1.5 Fail-Safe Stance

According to [1]

> "Another fundamental principle of security is that, to the extent possible, systems should fail safe, that is, if they're going to fail, they should fail in such a way that they deny access to an attacker, rather than letting the attacker in. The failure may also result in denying access to legitimate users as well, until repairs are made, but this is usually an acceptable tradeoff."

2.1.6 Universal Participation

Most security systems require the universal participation (or at least the absence of active opposition) of a site's personnel in order to be fully effective. For example, everybody should report strange happenings that might be security-related, everybody should choose good passwords, change them regularly; and not to give them out to their friends, relatives, etc..

2.1.7 Diversity of Defense

The idea behind diversity of defense is that using security systems from different vendors may reduce the chances of a common bug or configuration error that compromises them all [1]. Just as using a number of different systems to provide depth of defense gives additional security, using a number of different types of systems also gives additional security. However, there is a tradeoff between complexity and cost, so evaluation and decision need to be made by each site concerning this issue.

2.1.8 Simplicity

There are two reasons for making simplicity a security strategy. First, it is easier to understand simpler things; second, nooks and crannies are provided by complexity for all sorts of things to hide in.

2.2 TCP (Transmission Control Protocol) /IP (Internet Protocol) Fundamentals

Today the two major approaches used to build firewalls are packet filtering and proxy services. The packet filtering method is selected for this research project. Since a good understanding of TCP /IP is needed to follow the details of the discussions of packet filtering, TCP/IP fundamentals will be presented.

2.2.1 Introduction to TCP/IP

TCP and IP are two of the most important protocols in the suite of data communications protocols, even though there are many other protocols. The TCP/IP protocol has the following features (according to [1]):

- "Open protocol standards, freely available and developed independently from any specific computer hardware or operating system. Because it is so widely supported, TCP/IP is ideal for uniting different hardware and software.
- Independence from specific physical network hardware. This allows TCP/IP to integrate many different kinds of networks. TCP/IP can be run over an Ethernet, a token ring, a dial-up line, an X.25 net, and virtually any other kind of physical transmission media.
- A common addressing scheme that allows any TCP/IP device to uniquely addresses any other device in the entire network, even if the network is as large as the worldwide Internet.

- Standardized high-level protocols for consistent, widely available user services."

2.2.2 OSI (Open Systems Interconnection) Reference Model

The International Standards Organization (ISO), began to develop its Open Systems Interconnection (OSI) networking suite in the 1980s [2]. There are two major components in OSI: an abstract model of networking (the Basic Reference Model, or seven-layer model) and a set of concrete protocols.

There are seven layers in the OSI Reference Model as shown in table 2.1. One or more entities implement its functionality at each layer. Each entity interacts directly only with the layer below it, and provides facilities for use by the layer on top of it. Protocols enable an entity in one host to interact with a corresponding entity, which is at the same layer in a remote host.

Table 2.1 Seven Layers of The OSI Basic Reference Model (after [2])

1.	Physical Layer
2.	Data link Layer
3.	Network Layer
4.	Transport Layer
5.	Session Layer
6.	Presentation Layer
7.	Application Layer

The seven layers of the OSI Basic Reference Model [2] are (from bottom to top):

1. The **Physical Layer** describes the physical properties of the various communications media, as well as the electrical properties and interpretation of the exchanged signals. For example, this layer defines the size of Ethernet coaxial cable, the type of BNC connector used, and the termination method.

2. The **Data Link Layer** describes the logical organization of data bits transmitted on a particular medium. For example, this layer defines the framing, addressing and checksumming of Ethernet packets.
3. The **Network Layer** describes how a series of exchanges over various data links can deliver data between any two nodes in a network. For example, this layer defines the addressing and routing structure of the Internet.
4. The **Transport Layer** describes the quality and nature of the data delivery. For example, this layer defines if and how retransmissions will be used to ensure data delivery.
5. The **Session Layer** describes the organization of data sequences larger than the packets handled by lower layers. For example, this layer describes how request and reply packets are paired in a remote procedure call.
6. The **Presentation Layer** describes the syntax of data being transferred. For example, this layer describes how floating point numbers can be exchanged between hosts with different math formats.
7. The **Application Layer** describes how real work actually gets done. For example, this layer would implement file system operations.

2.2.3 TCP/IP Protocol Architecture

There is no universal agreement about how to describe TCP/IP with a layered model. Generally, it is viewed as being composed of fewer than the seven layers used in the OSI model. Three to five functional levels in the protocol architecture are defined in most descriptions of TCP/IP. The four-level model illustrated in table 2.2 is based on the three layers (Application, Host-to-Host, and Network Access) shown in the

Department of Defense (DOD) Protocol Model in the Defense Data Network (*DDN*) *Protocol Handbook* - *Volume 1*, with the addition of a separate Internet layer [1]. This model provides a reasonable graphic representation of the layers in the TCP/IP protocol hierarchy.

Table 2.2 Four Layers In The TCP/IP Protocol Architecture (after [3])

4. Application Layer	Consists of applications and processes that use the network
3. Host-to-Host Transport Layer	Provides end-to-end data delivery services
2. Internet Layer	Defines the datagram and handles the routing of data
1. Network Access Layer	Consists of routines for accessing physical networks

1. Network Access Layer

The lowest layer of the TCP/IP protocol hierarchy is the *Network Access Layer*. In this layer, the protocols provide the ways for the system to transmit data to the other devices on a directly attached network. It defines how to use the network to deliver an IP datagram. Unlike higher-level protocols, the details of the underlying network (its packet structure, addressing, etc.) must be known by the Network Access Layer protocols to correctly format the data being transmitted to comply with the network constraints. The TCP/IP Network Access Layer can include the functions of all three lower layers of the OSI reference Model (Network, Data Link, and Physical).

Functions performed at this level include encapsulation of IP datagrams into the frames transmitted by the network, and mapping of IP addresses to the physical addresses used by the network [4]. The universal addressing scheme is one of

TCP/IP's strengths. The IP address must be converted into an address that is suitable for the physical network over which the datagram is transmitted.

2. Internet Layer

   The *Internet Layer* is the layer above the Network Access Layer in the protocol hierarchy. The Internet Protocol is the most significant protocol in the Internet Layer and is the heart of TCP/IP. IP provides the fundamental packet delivery service on which TCP/IP networks are built. All protocols, in the layers above and below IP, use the Internet Protocol to transmit data. All TCP/IP data flows through IP, incoming and outgoing, regardless of its ultimate destination.

   The Internet Protocol is the building block of the Internet [5]. Its functions include:

   - Defining the datagram, which is the basic unit of transmission in the Internet
   - Defining the Internet addressing scheme
   - Moving data between the Network Access Layer and the Host-to-Host Transport Layer
   - Routing datagrams to remote hosts
   - Performing fragmentation and re-assembly of datagrams

   The TCP/IP protocols were built to deliver data over the Advanced Research Projects Agency Network (ARPANET), which was a *packet switching network*)[5]. A *packet* is a block of data, which carries with it the information needed to deliver it - in a manner similar to a postal letter, which has an address written on its envelope. The addressing information in the packets is used by a packet switching network to switch packets from one physical network to another, moving them toward their final destination. Each packet travels the network separately.

The *datagram* is the packet format that defined by Internet Protocol [5]. Figure 2.1 is a graphic representation of an IP datagram. The control information in the first five or six 32-bit words are called the *header* [5]. By default, the header is five words long; the sixth word is optional. There is a field called *Internet Header Length (IHL)* in the header that indicates the header's length in words, since the header's length is variable. All the information needed to deliver the packet is contained in the header.

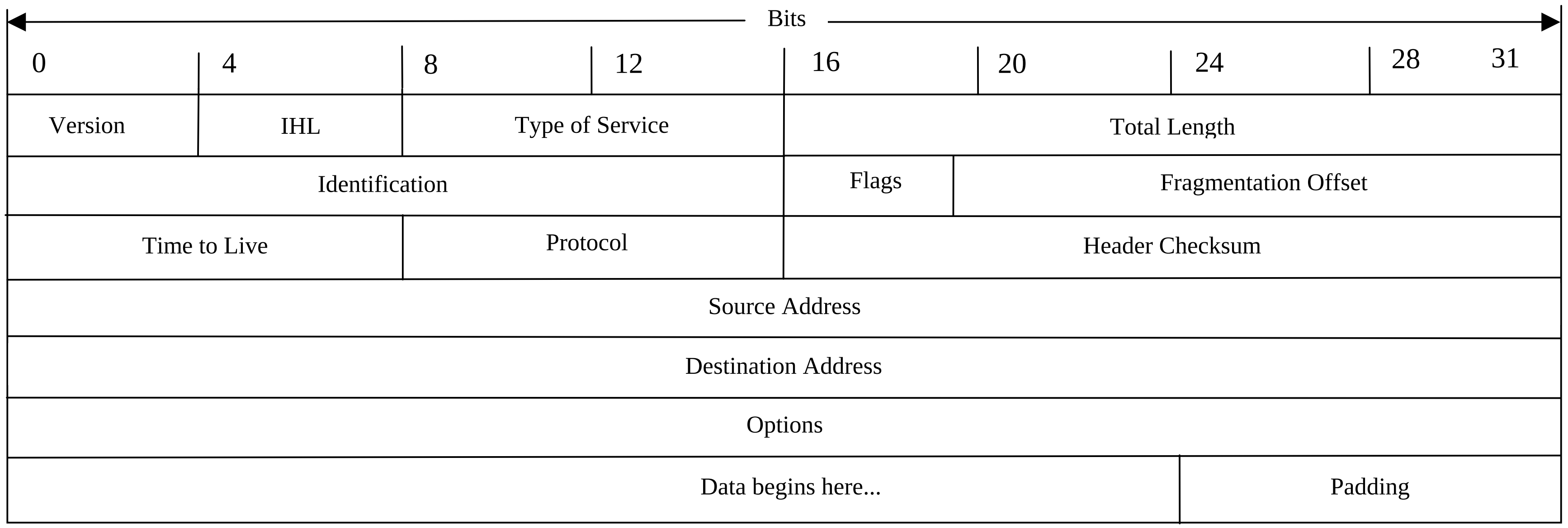


Figure 2.1 IP Datagram format (after [5])

The Internet Protocol transmits the datagram by checking the *Destination Address* in word 5 of the header. The Destination Address is a standard 32-bit IP address that identifies the destination network and the particular host on that network. The packet is transmitted directly to the destination if the Destination Address is the address of a host on the local network. Otherwise, the packet is passed to a gateway for delivery. *Gateways* are devices that switch packets between the different physical networks [5]. *Routing* is deciding which gateway to use [5]. The routing decision is made by IP for each individual packet. Figure 2.2 shows routing through gateways.

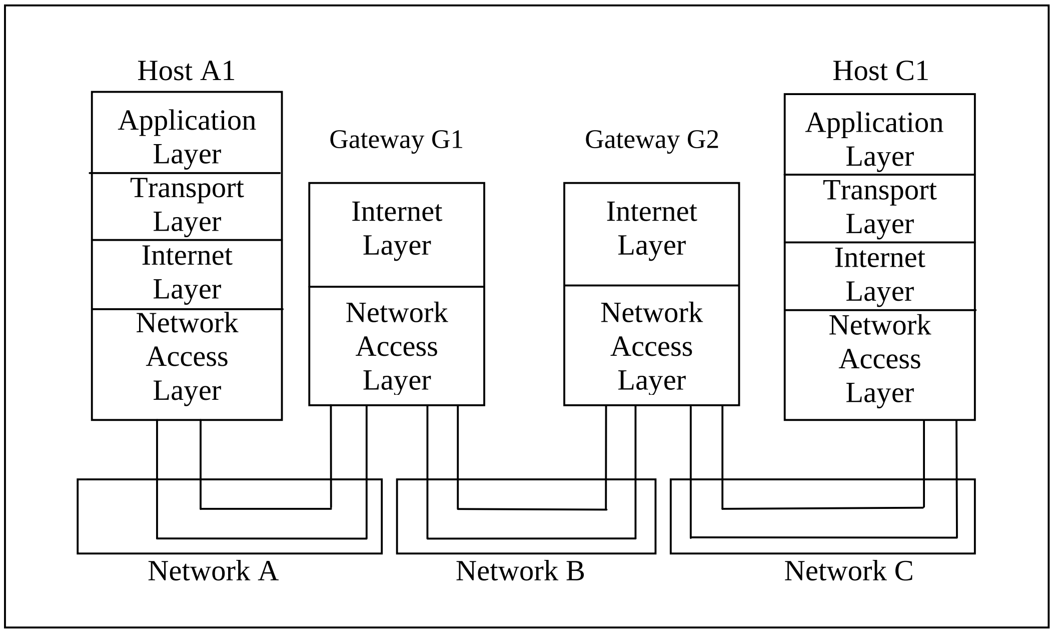


Figure 2.2 Routing Through Gateways (after [1])

The *Internet Control Message Protocol* (ICMP) is an integral part of IP. This protocol is part of the Internet Layer and uses the IP datagram transmission facility to send its messages. ICMP delivers messages that perform the following control, error reporting, and informational functions for TCP/IP (according to [5]):

- *Flow control*

  When datagrams arrive too fast for processing, the destination host or an intermediate gateway sends an ICMP Source Quench Message back to the sender. This tells the source to stop sending datagrams temporarily.

- *Detecting unreachable destinations*

  When a destination is unreachable, the system detecting the problem sends a Destination Unreachable Message to the datagram's source. If the unreachable destination is a network or host, the message is sent by an intermediate gateway. But if the destination is an unreachable port, the destination host sends the message.

- *Redirecting routes*

  A gateway sends the ICMP Redirect Message to tell a host to use another gateway, presumably because the other gateway is a better choice. This message can be used only when the source host is on the same network as both gateways.

- *Checking remote hosts*

  A host can send the ICMP Echo Message to see if a remote system's Internet Protocol is up and operational. When a system receives an echo message, it replies and sends the data from the packet back to the source host. The ping command uses this message [6].

3. Transport Layer

   The *Host-to-Host Transport Layer* is the protocol layer above the Internet Layer. This name is usually shortened to *Transport Layer*. The two most significant protocols in the Transport Layer are the *User Datagram Protocol* (UDP) and the *Transmission Control Protocol* (TCP). UDP provides low-overhead, connectionless datagram transmission service. Reliable data delivery service is provided by TCP with end-to-end error detection and correction [7]. Both protocols transmit data between the Application Layer and the Internet Layer. Applications programmers can choose whichever service is more suitable for a specific application.

- User Datagram Protocol

  Like the transmission service that IP provides the User Datagram Protocol gives application programs direct access to a datagram delivery service. This allows applications to exchange messages over the network with a minimum of protocol overhead.

  UDP is an untrustworthy, connectionless datagram protocol. “Untrustworthy” merely means that there are no facilities in the protocol for verifying that the data reached the other end of the network correctly. Within your computer, UDP will transmit data correctly. The 16-bit *Source Port* and *Destination Port* numbers in word 1 of the message header is used by UDP to deliver data to the correct application process. Figure 2.3 shows the UDP message format.

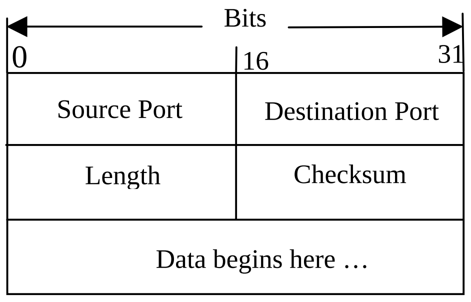


Figure 2.3 UDP Message Format (after [7])

There are a number of good reasons why UDP is chosen as a data transport service. If the amount of data being delivered is small, the overhead of creating connections and ensuring reliable delivery may be greater than the work of re-transmitting the entire data set. In this case, the most efficient choice for a Transport Layer protocol is UDP.

- Transmission Control Protocol

  TCP is used by applications that require the transport protocol to provide reliable data delivery because it verifies that data is delivered across the network accurately and in the proper sequence. TCP is such a protocol that is *reliable*, *connection-oriented*, and *byte-stream*.

  a. Reliable

     Reliability is provided by TCP with a mechanism called *Positive Acknowledgment with Re-transmission* (PAR). Simply stated, a system using PAR sends the data again until it hears from the remote system that the data arrived okay. A *segment* is the unit of data exchanged between cooperating TCP modules (see figure 2.4). A checksum that the recipient uses to verify that the data is undamaged is contained in each segment. The receiver sends a *positive acknowledgment* back to the sender if the data segment is received undamaged. Otherwise the receiver discards it. After an appropriate time-out period, the sending TCP module re-delivers any segment for which no positive acknowledgment has been received.

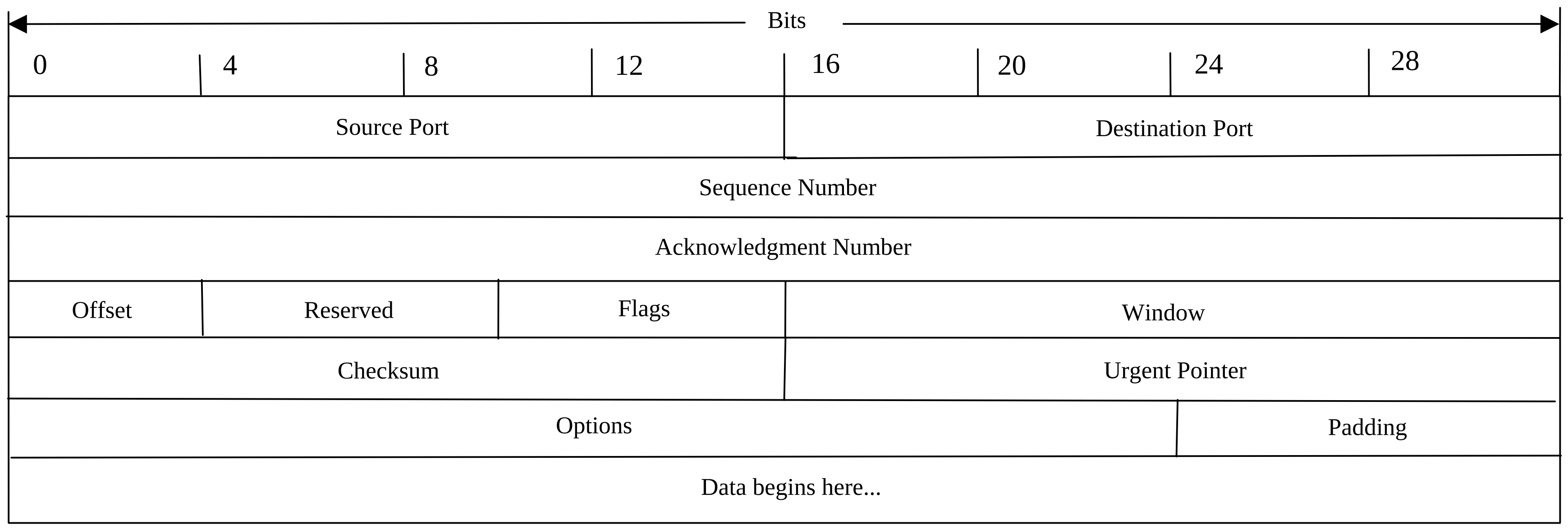


Figure 2.4 TCP Segment Format (after [7])

b. Connection-oriented

TCP is connection-oriented. A logical end-to-end connection between the two communicating hosts is established by TCP. Control information, called a *handshake*, is exchanged between the two endpoints to establish a dialogue before data is delivered. The control function of a segment is indicated by setting the appropriate bit in the Flags field in word 4 of the *segment header*.

c. Byte-stream

The data sent by TCP is viewed as a continuous stream of bytes, not as independent packets. Thus, the sequence in which bytes are sent and received is maintained by TCP. Two fields in the TCP segment header, the Sequence Number and Acknowledgment Number, keep track of the bytes.

4. Application Layer

The *Application Layer* is at the top of the TCP/IP protocol architecture. All processes that use the Transport Layer protocols to deliver data are included in this layer. Following are the most widely known and implemented applications protocols [8]:

Telnet: The Network Terminal Protocol that provides remote login over the network.

FTP: The File Transfer Protocol that is used for interactive file transfer.

SMTP: The Simple Mail Transfer Protocol that delivers electronic mail.

HTTP: The Hypertext Transfer Protocol that delivers Web pages over the network.

2.3 Packet Filtering

To understand packet filtering, packets and how they are handled at each layer of the TCP/IP protocol stack must be understood.

2.3.1 Packet Definition

To transfer information across a network, the information has to be broken up into small pieces, each of which is sent separately. Breaking the information into pieces allows many systems to share the network, each sending pieces in turn. In IP networking, those small pieces of data are called packets. All data transfer across IP network happens in the form of packets. So the packet is the fundamental unit of communication on the Internet.

2.3.2 Packet Structure

Packets are constructed in such a way that layers for each protocol used for a particularly connection are wrapped around the packets. At each layer, a packet has two parts: the header and the body. The header contains protocol information relevant to that layer, while the body contains the data for that layer which often consists of a whole packet from the next layer in the stack. Each layer treats the information it gets from the layer above it as data, and applies its own header to this data. At each layer, the packet contains all of the information passed from the higher layer; nothing is lost. The process of preserving the data while attaching a new header is known as encapsulation. Figure 2.5 shows how this works.

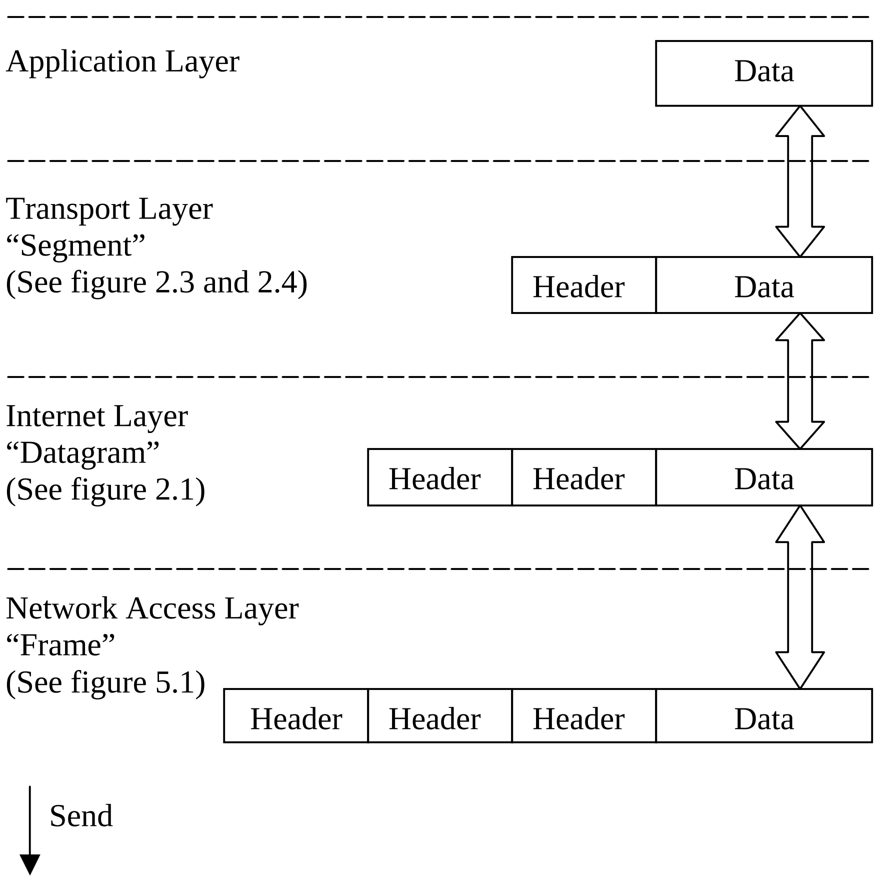


Figure 2.5 Data Encapsulation (after [3])

At the application layer, the packet consists simply of the data to be transferred. As it moves to the transport layer, the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP) preserves the data from the previous layer and attaches a header to it. At the next layer, IP considers the entire packet (composed now of the TCP or UDP header and the data) to be data, and now attaches its own IP header. Finally, at the network access layer, Ethernet or another network protocol considers the entire IP packet passed to it to be the data, and attaches its own header.

2.3.3 Packet Filtering Definition

Packet filtering is a network security mechanism that works by controlling what data can flow to and from a network. Packet filters allow or block packets, usually while routing them from one network to another (most often from the Internet to an internal network, and vice versa). To accomplish packet filtering, a set of rules that specify what types of packets are to be allowed and what types are to be blocked is set up.

2.3.4 Reasons for Packet Filtering

Packet filtering controls (allows or disallows) data transfer based on:

- The address the data is (supposedly) coming from
- The address the data is going to
- The session and application protocols being used to transfer the data

2.3.5 Advantages of Packet Filtering

Packet filtering has a number of advantages:

- One router can help protect an entire network

  One of the key advantages of packet filtering is that a single, strategically placed packet-filtering router can help protect an entire network. If there is only one router that connects the internal network to the Internet, tremendous leverage on network security will be gained, regardless of the size of the internal network, by doing packet filtering on that router.

- Packet filtering doesn’t require user knowledge or cooperation

  Packet filtering doesn’t require any custom software or configuration of client machines, nor does it require any special training or procedure for users.

- Packet filtering is widely available in many routers

  Packet filtering capabilities are available in many hardware and software routing products.

2.3.6 Types of Packet Filtering

Every packet has a set of headers containing certain information as mentioned in the previous section. The main information is:

- IP source address
- IP destination address
- Protocol (whether the packet is a TCP, UDP, or ICMP packet)
- TCP or UDP source port
- TCP or UDP destination port
- ICMP message type

The information above can be used to set up the specific policy, for example, filtering by service, filtering by source port and so on. In this project, the IP source address is used as the policy, that is to say filtering the packet by the source's IP address. Filtering in this way restricts the flow of packets based on the source IP address of the packets, without having to consider what protocols are involved. However, all the information in the header could be used to do packet filtering. It should noted that while the source IP address is a popular choice for packet filtering, any or all of the information in the packet header may be used. This makes the Silicon Firewall can handle IP source address masquerading. Also, in the future the dynamic CAM will handle this issue better.

Chapter 3 HARDWARE/SOFTWARE SYSTEM CODESIGN

The background of Internet security and Internet firewall security systems were presented in Chapter 1 and Chapter 2 respectively. Since the Silicon Firewall system is an embedded system, and the design of the Silicon Firewall is a hardware-software codesign, before introducing the design issues of the Silicon Firewall system, some background of embedded system and hardware-software codesign are given. Since the microprocessor is the common component between the hardware-software codesign processes, background information on embedded processor and embedded processor Programmable Logic Device (PLD) solutions are given. Finally, the Altera® Excalibur™ embedded processor programmable solutions are presented as the solution that was used in the Silicon Firewall system design.

3.1 Embedded System Overview

The Silicon Firewall system is an embedded real-time system. Embedded systems are quite diverse; no one statement applies to all cases. Basically, embedded systems are built to constantly respond to external events and to generate control outputs as a function of their current state and inputs from sensors, etc..

Embedded-system specification and design consists of two tasks, the first is describing a system's desired functionality; the second is mapping that functionality for implementation by a set of system components such as processors, FPGAs, memories, and buses [9]. Ever-increasing embedded-system design complexity combined with a very tight time-to-market window has revolutionized the embedded system design process. The concurrent design of hardware and software has displaced traditional sequential design. Further more, hardware and software design now begins before the system architecture is finalized. Requirement definitions and system specifications are developed by system architectures, customers, and marketing departments together [10]. Embedded computing is unique because it is a codesign problem where the hardware and software architecture should be designed simultaneously.

3.2 Hardware-Software Codesign

The practice of concurrent hardware-software design can significantly cut the cost and cycle to build digital systems for embedded real-time applications. It has been proved that the traditional hardware-first, software-last development process is difficult and costly [11]. The codesign approach speeds up the intuitively serial design process by developing hardware and software concurrently. It helps the embedded system designers meet the design and development deadlines.

According to [12]

> "In hardware-software codesign designers consider trade-offs in the way hardware and software components of a system work together to exhibit a specified behavior, given a set of performance goals and an implementation technology. Because of a wide range of possible system structures and design goals, the hardware-software codesign problem takes on many forms. One type of codesign seeks to accelerate application software by extracting portions for implementation in hardware. Programmable hardware may take this type of software acceleration common even in general-purpose computing. In this case, the codesign problem entails characterizing hardware and software performance, identifying a hardware-software partition, transforming the functional description into such a partition, and synthesizing the resulting hardware and software. High-level (or behavioral) synthesis can produce hardware implementations for functions described in a high-level software language such as C."

The Silicon Firewall system belongs to this type, the system hardware was written in behavioral and register-transfer-level Verilog HDL, while the software was written in C.

The processor is the only piece in common between the hardware-software codesign processes, whether it is a microprocessor/microcontroller or digital signal processor. That is to say the processor is the common link that bridges the gap between software and hardware design.

3.3 System-on-a-Chip Design

Today's submicron fabrication technologies enable designers to put large numbers of device on a single microchip. Along with this capability, a vast choice of hardware and software components makes system design and validation increasingly complex. As system design grows increasingly complex, the use of predesigned components, such as general-purpose microprocessors, can simplify synthesized hardware.

3.4 Embedded Processor PLD (Programmable Logic Device) Solutions

As digital systems progress towards higher levels of integration, system designers benefit from lower development costs, shorter design cycles, increased performance, and lower power consumption. Functions once performed by multiple, individual devices are now combined into more capable, higher density devices, achieving higher integration at the device level. Additionally, greater time-to-market pressures and frequent changes in system specifications require an increase in design flexibility. In particular, two types of programmable, off-the-shelf components maintain flexibility and increase system integration: microprocessors (software) and programmable logic (hardware).

Given the advantages above, the next logical step in system integration would be the combination of embedded processors and programmable logic. In addition to offering all of the traditional benefits of higher integration, embedded processor PLD solutions also provide unique advantages to the system designer because of the flexibility of programmable logic, it also give the system designer unprecedented freedom to determine which functions should be executed in software and which would benefit most from dedicated hardware. Table 3.1 compares the capabilities of embedded processor PLDs to other component-level solutions for system integration, such as application specific integrated circuits (ASICs) and application specific standard products (ASSPs).

Table 3.1 Component-Level Solutions for System Integration (after [13])

	ASICs	**ASSPs**	**Embedded Processor PLDs**
Level of System Integration	High	Moderate	High
Development Cost	High	Low	Low
Unit Cost	Low	Low	Moderate
Design Flexibility	Low	Low	High
Total Time to Market	Long	Moderate	Short

3.5 Altera® Excalibur™ Embedded Processor Programmable Solutions

Altera® Excalibur™ embedded processor solutions provided the tools needed to integrate an entire system on a single programmable logic device (PLD). The Excalibur solutions help speed the development and shorten time-to-market for the embedded processor applications by offering the ARM®-based hard core (fixed architecture) embedded processors and Nios™ soft core (configurable architecture) embedded processors. Combined with programmable and memory, the Excalibur solutions give all the programmable solutions needed with single-chip integration.

The advanced features of Excalibur Solutions include:

- 200-MHz high performance RISC (Reduce Instruction Set Computer) processor
- Support for a wide range of end applications
- Integrated hardware and software development workflow
- Optimized integration with Altera device architectures

3.6 Soft Core vs. Hard Core

The arguments for both soft and hard core processor implementation in PLDs are convincing [13]. Flexibility, scalability, and low absolute cost are offered by soft cores. Many applications that require moderate performance are fit by the soft cores, and they immediately benefit from process enhancements to their target hardware platform. On the other hand, maximum performance and cost effectiveness are offered by hard core processors, primarily for applications that require high performance. The Nios processor's parameterizability allows users to make the performance/cost tradeoff quickly, without needing to be a processor architect. Regardless of the configuration, the same instruction set allows Altera to deliver fully-verified cores and industry-standard software development tools such as C/C++ compilers [13]. The leading-edge, 32-bit RISC processor performance and substantial system RAM are offered by the ARM-and MIPS-based products, integrated with industry-leading programmable logic architectures. These products bring the performance, memory capacity and gate density normally associated with ASICs within the reach of every software or hardware designer. Table 3.2 outlines the difference between the soft core and hard core Excalibur product families.

Table 3.2 Excalibur Product Comparison (after [13])

	Nios (Soft Core)	**ARM-&MIPS-Based** (Hard Core)
Flexibility	High	Moderate
Performance	Moderate	High
Multiprocessor Implementation	Yes	No
Processor Enhancement Benefit	Immediately	Requires Mask Change

3.7 Excalibur Workflow

The Excalibur solutions are supported by a complete design workflow, which automates system design, incorporating familiar hardware and software (C/C++ code) methodologies [13].

All the tools necessary to develop Excalibur designs are provided by Altera, including an industry-standard C/C++ compiler and debugger, peripherals, and drivers, the Quartus™ II software for PLD design development, and download cables for device programming and verification. These tools provide a system-centric approach to development and allow hardware and software to be created concurrently. Figure 3.1 illustrates the Excalibur workflow.

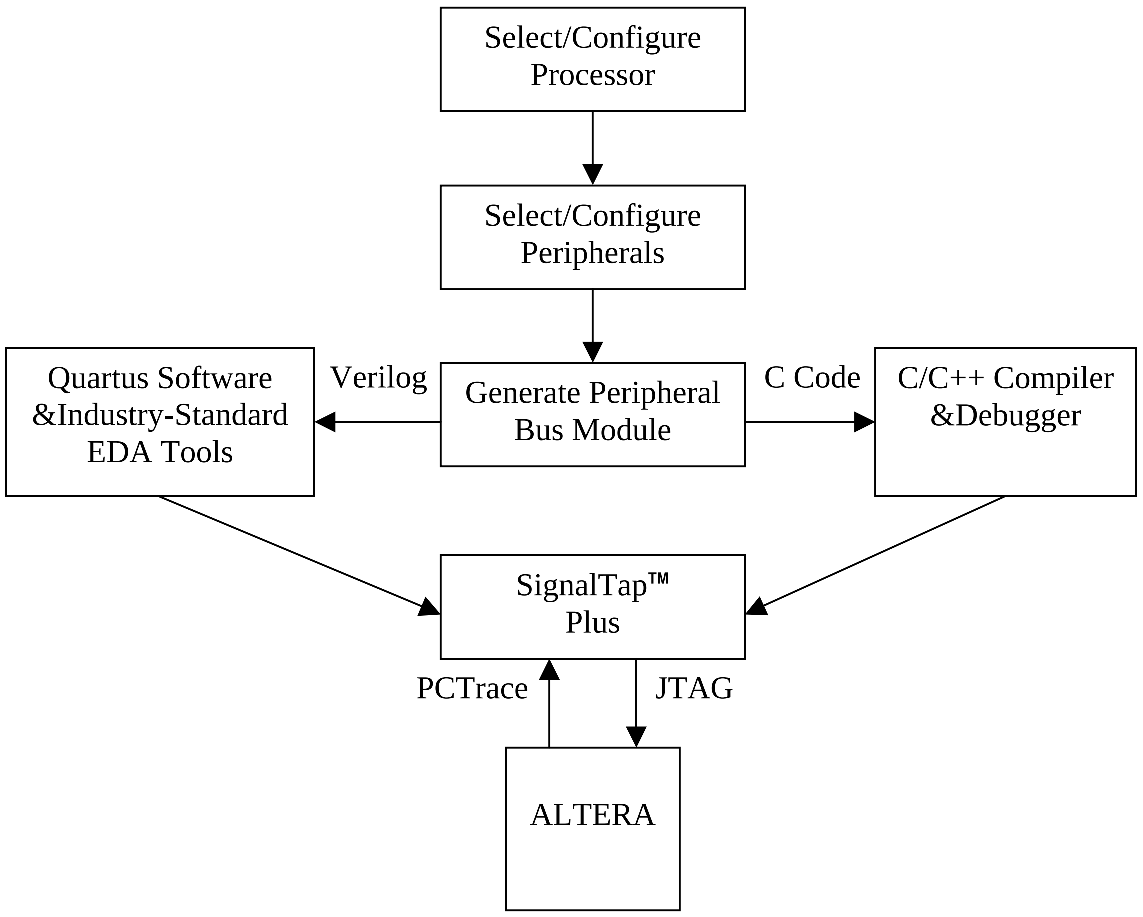

Select/Configure Processor
Select/Configure Peripherals
Quartus Software &Industry-Standard EDA Tools
Verilog
Generate Peripheral Bus Module
C Code
C/C++ Compiler &Debugger
SignalTap™ Plus
PCTrace
JTAG
ALTERA

Figure 3.1 The Excalibur Workflow (after [13])

3.8 Silicon Firewall System

Figure 3.2 is a simple diagram of the Silicon Firewall system. Basically in this hardware software codesign, the software and hardware are split as follows: the software does the initialization and configuration; the hardware does packet filtering. Using software to do configuration is easier, while using hardware to do packet filtering is faster, that is the reasons for splitting software and hardware in this way.

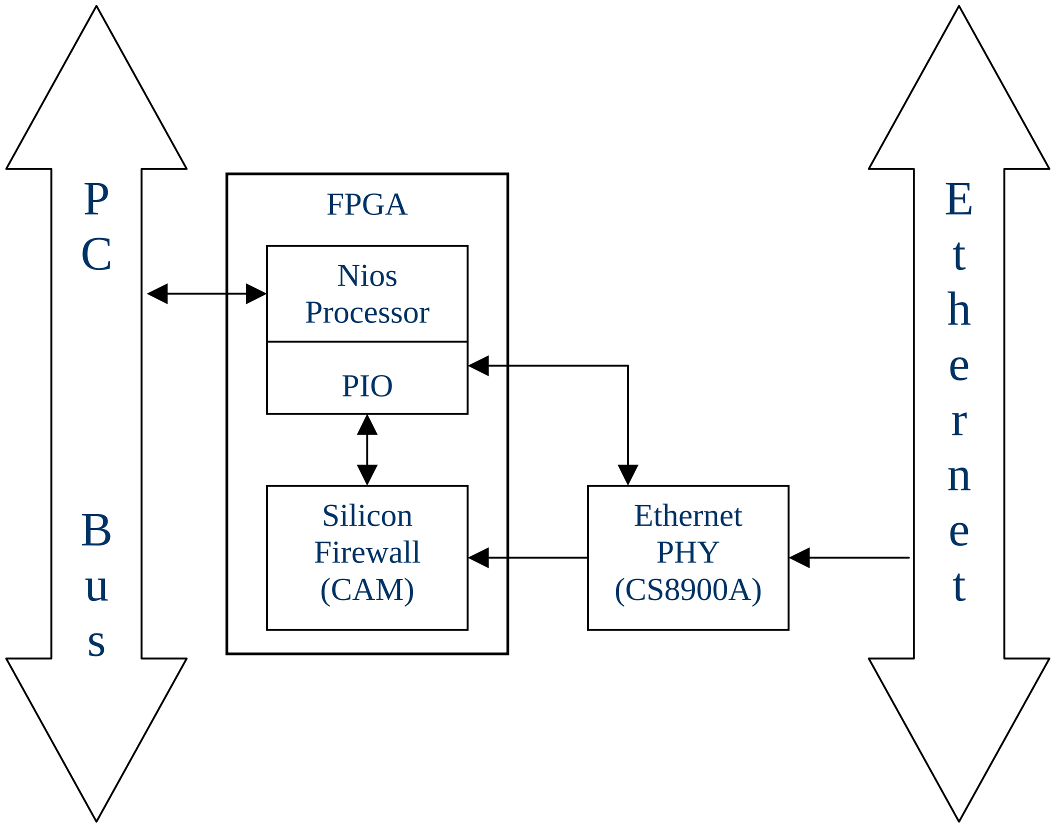


Figure 3.2 A Simple Diagram of The Silicon Firewall System

Chapter 4 ALTERA NIOS EMBEDDED PROCESSOR

The Altera Nios™ soft core embedded processor was selected in the Silicon Firewall system based on the advantages of soft core processors described in section 3.6. The Nios processor will be introduced in more detail in this chapter. The Nios Development Kit (NDK) and the Nios Ethernet Development Kit (NEDK) are also presented. The Nios development board and the daughter card are introduced.

4.1 Nios Embedded Processor Overview

The Nios® embedded processor is a user-configurable, general-purpose RISC embedded processor [14]. It was designed to be a flexible and powerful processor solution. The ease-of-use and flexibility make the Nios processor's one of the most popular embedded processors in the world. Custom processor-based systems can be created using the SOPC Builder system development tool. One or more configurable Nios CPUs can be integrated by the SOPC Builder into an FPGA with any number of standard peripherals, “gluing” the system together with the automatically generated Avalon™ switch fabric.

The Nios embedded processor is optimized for system-on-a-programmable-chip (SOPC) integration and Altera® programmable logic. The Nios processor and user logic can be combined together and programmed into an FPGA using SOPC Builder. The Nios embedded processor’s unique features such as custom instructions and the simultaneous multi-master Avalon switch fabric make it different from other soft core processor solutions. These features provide simple, yet non-traditional methods to accelerate and optimize the designs. Table 4.1 shows a comparison of the 32-bit and 16-bit Nios embedded processors in typical configurations.

Table 4.1 Comparisons of Typical Nios Processor Configurations (after [14])

Feature	**32-Bit Nios CPU**	**16-Bit Nios CPU**
Data bus size (bits)	32	16
Arithmetic logic unit (ALU) width (bits)	32	16
Internal register width (bits)	32	16
Address bus size (bits)	32	16
Instruction size (bits)	16	16
Logic elements (LEs) (typical)	Fewer than 1,500	Fewer than 1,000
f_{MAX}	Over 125 MHz	Over 125 MHz

4.2 The Nios Soft Core Embedded Processor

The Nios embedded processor is the first RISC processor soft core to be developed specially for programmable logic and can provide up to 50MIPS (Million Instructions Per Second) performance while being optimized for area in a PLD. Figure 4.1 shows a block diagram of the Nios embedded processor.

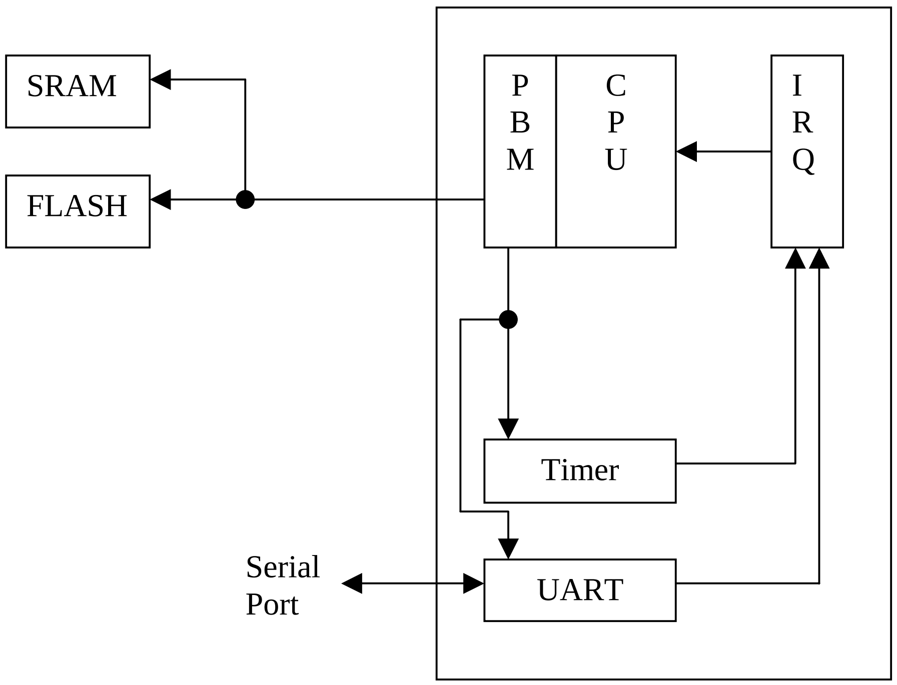


Figure 4.1 Block Diagram of the Nios Embedded Processor (after [13])

4.3 Nios Processor and Peripherals

An address map with different types, widths, and speeds of memory and peripherals can be designed using a simple interface. The interface logic that connects all Nios peripherals as defined by the user is generated by the interface. Figure 4.2 diagrams the communication between the Nios embedded processor and the peripherals. The interface to each is specified as peripherals are added. The interface creates a peripheral bus module (PBM) according to the configuration specified.

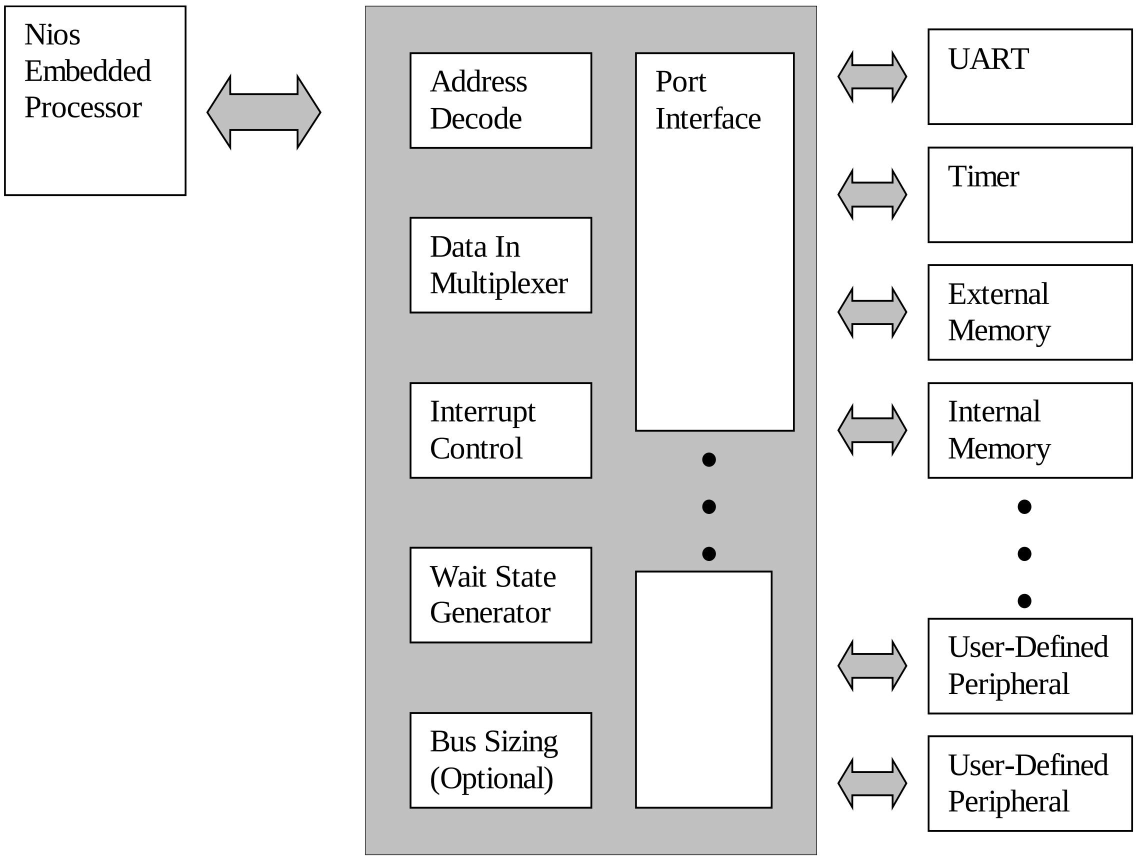


Figure 4.2 Nios Processor and Peripherals (after [15])

4.4 Nios Development Kit

Altera's Nios® development kit provides everything needed for system-on-a-programmable-chip (SOPC) development.

This kit includes:

- Nios embedded processor configurable CPU soft core
- Library of standard microprocessor peripherals
- SOPC Builder system development tool
- GNUPro compiler and debugger from Red Hat
- Quartus® II Limited Edition development software

- Nios development board populated by an APEX™ 20KE device (EP20K200EFC484)
- Multiple SOPC reference designs targeted to the Nios development board

4.4.1 Nios Peripheral Library

The Nios® embedded processor development kits include a library of standard peripherals that are available for use in Altera® programmable logic. These peripherals are provided to the user as Verilog HDL or VHDL source code and include all of the necessary software routines for easy system integration. The Nios peripheral library (Verilog or VHDL code) includes the peripherals listed in Table 4.2.

Table 4.2 Nios Peripheral Library (after [16])

Peripheral	Description
UART	Common serial interface; with variable baud rate, parity, stop and data bits, and optional flow control signals
Timer	32-bit timer; can be used as periodic pulse generator or system watchdog timer
Parallel I/O (PIO)	1- to 32-bit parallel I/O (input, output, and edge-capture)
Serial Peripheral Interface (SPI)	Serial peripheral interface, 3-wire, master/slave
Direct Memory Access (DMA)	The DMA peripheral allows for efficient bulk data transfer between peripherals and memory by removing the CPU from the data path
Memory Interfaces	• On-chip ROM and RAM • Off-chip SDRAM and SSRAM, SRAM, and flash • Off-chip Altera serial configuration device
Ethernet Port	• 10 Mbps Cirrus Logic CS8900A PHY/MAC chip • Interfaces supported by Plugs Ethernet Library
Interface to User Logic	Used to easily connect on-chip user logic or off-chip devices to an SOPC Builder-generated system

4.4.2 Development Tools

All the necessary tools for effective embedded system development are contained in the Nios development kit. The NDK comprises the two kits previously known as the HDK (Hardware Development Kit) and the SDK (Software Development Kit). The HDK is used to create Nios embedded processor systems in APEX devices. The HDK consists of the Altera's SOPC Builder tool and the Quartus II development software that combine to create a hardware development tool flow. On the software side, the SDK allows compile, run, debug C and assembly language programs on the Nios embedded processors. Altera has chosen the GNUPro toolkit, which is a popular and well-known suite of embedded software development tools.

The reader is referred to appendix D for more information about the three components of NDK: SOPC Builder system development tool, Quartus II development software and GNUPro Toolkit Compiler and Debugger.

4.5 Nios Development Board

The Altera Excalibur development board [14] is an advanced and integrated solution for creating embedded processor applications. It contains the following things:

- An APEX EP20K200E programmable device
- 8 Mbits (512K x 16) of on board Flash RAM
- Two 1 Mbit (64K x 16) on board SRAM devices
- An RS-232 communication port
- A JTAG port
- A parallel port
- Multiple expansion ports

- Two LEDs
- Two 7-Segment displays
- miscellaneous other switches and components

Figure 4.3 shows a diagram of the Excalibur board.

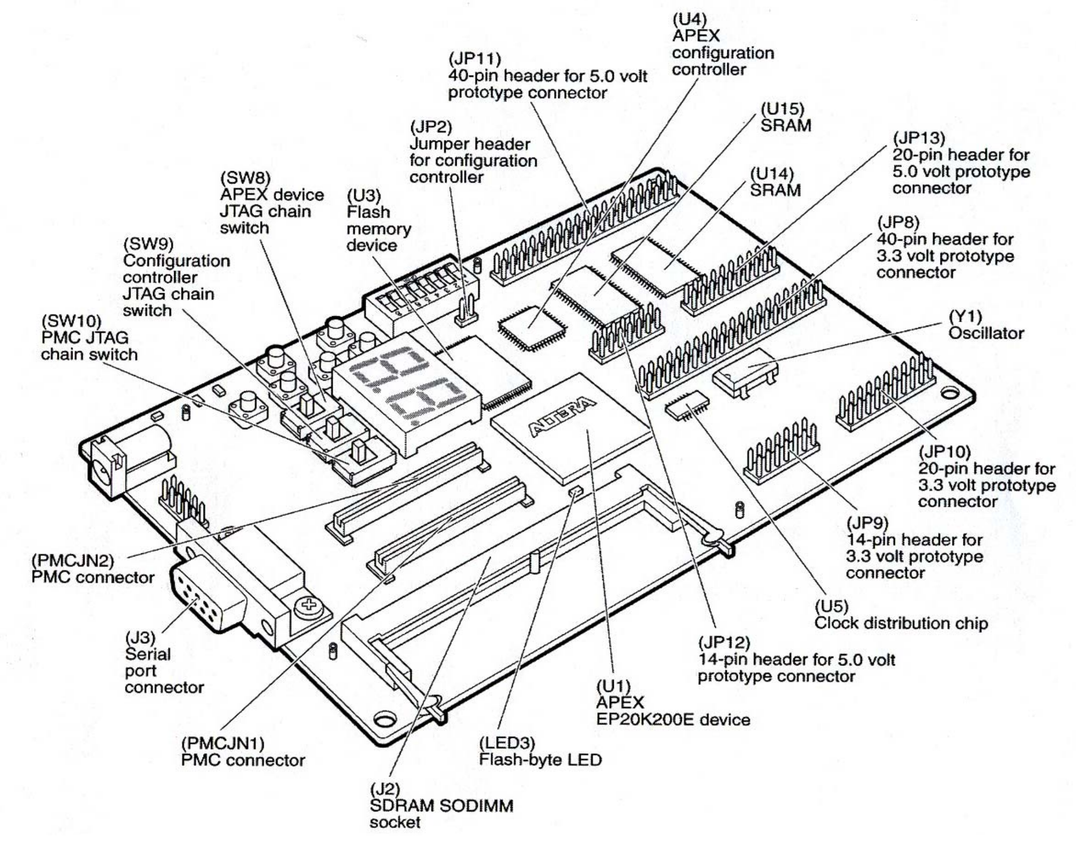

(U4) APEX configuration controller
(JP11) 40-pin header for 5.0 volt prototype connector
(U15) SRAM
(JP2) Jumper header for configuration controller
(JP13) 20-pin header for 5.0 volt prototype connector
(SW8) APEX device JTAG chain switch
(U3) Flash memory device
(U14) SRAM
(JP8) 40-pin header for 3.3 volt prototype connector
(SW9) Configuration controller JTAG chain switch
(SW10) PMC JTAG chain switch
(Y1) Oscillator
(JP10) 20-pin header for 3.3 volt prototype connector
(JP9) 14-pin header for 3.3 volt prototype connector
(PMCJN2) PMC connector
(U5) Clock distribution chip
(J3) Serial port connector
(JP12) 14-pin header for 5.0 volt prototype connector
(U1) APEX EP20K200E device
(PMCJN1) PMC connector
(LED3) Flash-byte LED
(J2) SDRAM SODIMM socket
ALTERA

Figure 4.3 Nios Development Board (from [14])

The default Nios soft core processor can be loaded into the Altera FPGA, which contains a 16 bit instruction set, and the data bus is capable of operating with a 16 or 32 bit. It can perform 50 million instructions per second with one instruction per clock cycle.

The two internal SRAM devices can be used with 16 or 32 bit applications, the 16 bit is smaller, while the 32 bit is faster, thus in this research project the 32 bit data path is selected. A 144-pin SODIMM memory expansion socket is also provided on the board if needed. Both the Nios processor and the APEX™ device share the flash memory.

The flash memory is organized as follows:

Table 4.3 Flash Memory Configurations (after [14])

Flash Address	Size	Comments
0x1C00000 – 0x1FFFFF	256Kbyte	Factory-default APEX configuration
0x180000 – 0x1BFFFF	256 Kbyte	User-defined APEX configuration data
0x100000 – 0x17FFFF	512 Kbyte	Nios instruction and nonvolatile data space

There is a factory programmed controller chip on the board, which is a MAX7054 device that loads data from the flash and clocks it into the APEX device. The beginning address for the user-defined configurations is 0x180000; for factory default the starting address is 0x1C0000.

The Excalibur board provides 5 volt and 3.3 volt daughter cards for the purposes of expansion. Three connectors are provided for 5-volt cards: a 40-pin connector (JP11), a 20-pin connector (JP13), and a 14-pin connector (JP12). The same applies to 3.3-volt cards, using JP8, JP10, and JP9, respectively.

There are three programmable devices on the Excalibur board: the APEX device, the configuration controller, and the PCI Mezzanine Card (PMC) (devices for JNC1 and JNC2). SW8, SW9, and SW10 determine the ability to program each respectively. The corresponding device is added to the JTAG chain if a switch is positioned to the left (marked connect on the board); the device will be removed from the chain if each switch positioned to bypass.

SW1-SW7 are the seven remaining switches on the board. SW1 is an eight-pin user defined DIPswitch. SW2 is a special button used for resetting the board. The configuration controller reloads the flash memory into the APEX device upon a reset. SW3 is defined by the configuration controller (a CPU reset by factory default), which is the clear function. SW4-SW7 are user defined and may perform any function necessary. When pressed, the logic zero signals are provided.

On the Excalibur board there are two clocks for use. The first one is provided by an onboard oscillator, which is a 33.3333 MHz signal. The second one can be used to allow the user to create the clock, utilizing the phase locked loop circuitry on the board. In this research project, the first one is used.

4.6 Nios Ethernet Development Kit

The Nios Ethernet Development Kit (NEDK)[17] is an add-on to the Nios Development Board. It includes a daughter card with an Ethernet interface chip, and make this peripheral available to Nios system built with Quartus.

The Nios Ethernet Development Kit includes hardware and software components that provide network connectivity for the Nios-based embedded systems. The components included in this kit are (according to [17]):

- A network-interface daughter card that can plug directly into the Nios development board.
- An SOPC Builder library component that defines the logic and interface signals necessary to use the daughter card in a Nios system.
- A C language library that provides a network-protocol stack. This library includes support for raw Ethernet, address resolution protocol (ARP), Internet

protocol (IP), Internet control message protocol (ICMP), user datagram protocol (UDP), and transmission control protocol (TCP) protocols and utility routines for controlling the daughter card hardware.

APEX™ device hardware reference designs and example software application programs are included in this kit. These reference designs and application examples can be used as starting points to be modified for the specific network-enabled application.

The following items are included in the Nios EDK:

- Nios EDK daughter card based on the Cirrus Logic CS8900A PHY/MAC chip
- Cabling
- Nios EDK CD-ROM

The Nios EDK CD-ROM contains the following files (according to [17]):

- SOPC library components
- PC-board schematic and layout files for the Nios EDK daughter card
- Example hardware reference design configurations:
    - Nios 32-bit CPU for a single daughter card
    - Nios 16-bit CPU for a single daughter card
    - Nios 32-bit for dual-stacked daughter cards
- Example software applications:
    - Library general demonstration and configuration programs
    - Example web server
    - Nios 32-bit CPU network-based GERMS monitor application example

4.7 Nios EDK Daughter Card

The Nios EDK daughter card is a circuit board with the following components (according to [17]):

- A Cirrus Logic CS8900A integrated Ethernet 10Mbit PHY/MAC chip
- A RJ-45 network connector with integrated-transformer magnetics and Link/LAN LEDs
- A 20 MHz crystal oscillator that is used by the CS8900A chip

Figure 4.4 shows the picture of the daughter card.

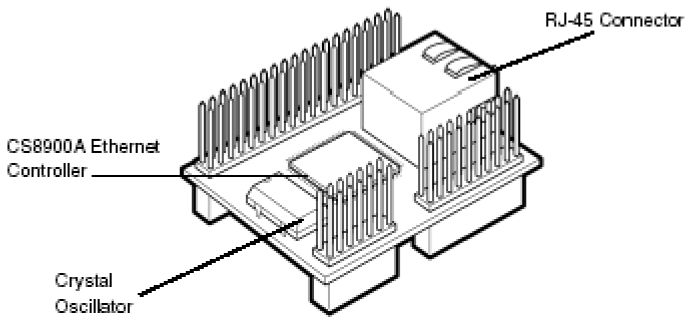


Figure 4.4 Nios EDK Daughter Card (from [17])

The CS8900A integrated PHY/MAC chip is a main functional component on the Nios EDK daughter card. An ISA-bus interface (not used in this research) is presented to the Nios CPU by the CS8900A. The set of female connectors provides the necessary electrical-interface signals. These connectors are compatible with the expansion connector groups on the Nios development board. The Nios EDK daughter card is compatible with either the 5-V (JP11, JP12, JP13) or the 3.3-V (JP8, JP9, JP10) expansion connector groups. The daughter card does not use any 5-V signals.

4.8 Plugs Library

The Plugs Library is a software library included in the Nios EDK. It allows the software to use standard network protocols for transmitting and receiving data. The features of the plugs library are as follows (according to [17]):

- Access to low-level packets
- Access to high level-packet payloads
- Conforms to RFCs
- Allows opening connections and sending data with only a few lines of code
- Is similar to the Unix-standard sockets routines
- Each plug can be set to print debug information for either transmit or receive data

The protocols supported by the plugs library are (according to [17]):

- Raw Ethernet
- Address resolution protocol (ARP)
- Internet protocol (IP)
- Internet control message protocol (ICMP)
- User datagram protocol (UDP)
- Transmission control protocol (TCP)

Figure 4.5 shows the relationships between the library-supported Nios EDK protocols.

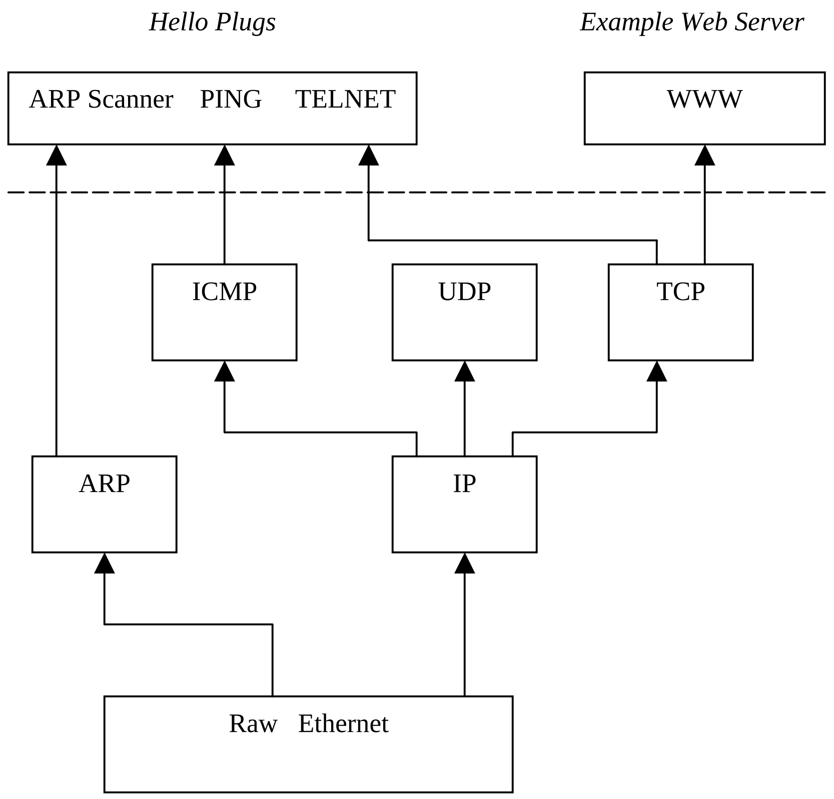


Figure 4.5 Plugs Library-Supported Nios EDK Protocols (after [17])

The Ethernet and 802.3 packets are supported by the Nios EDK. Ethernet packets are sent and received by the library routines to and from arbitrary 48-bit Ethernet media access control (MAC) address. Higher level protocols (such as ICMP, UDP, and TCP) use Ethernet transparently.

Chapter 5 SILICON FIREWALL DESIGN

In this chapter, the Silicon Firewall design is presented in detail. Details of the CAM and the CS8900A are discussed since they are the two major peripherals of the Nios CPU in the Silicon Firewall.

5.1 Introduction to CS8900A

The CS8900A [18] is a low-cost Ethernet Local Area Network (LAN) Controller that is optimized for Industry Standard Architecture (ISA) Personal Computers. It is a single chip, full-duplex, Ethernet solution, incorporating all of the analog and digital circuitry needed for a complete Ethernet circuit. The CS8900A, the most complicated peripheral in the Silicon Firewall system, will be introduced in this section.

5.1.1 General Description

Major functional blocks of the CS8900A [18] include: a direct ISA-bus interface, an 802.3 Media Access Control (MAC) engine, integrated buffer memory, a serial Electrically Erasable Programmable Read-Only Memory (EEPROM) interface, and a complete analog front end with both 10BASE-T and AUI (Attachment Unit Interface). The CS8900A must be configured before it can perform its two basic functions, Ethernet packet transmission and reception. Various parameters such as Memory Base Address, Ethernet Physical Address, frame types to receive, and which media interface to use must be written to its internal Configuration and Control registers. There are two methods to do the configuration, the first is using the host to write the configuration data to CS8900A across the ISA bus (or direct using register addressing) or to load the data automatically from an external EEPROM.

The Silicon Firewall uses direct register addressing (I/O) mode to configure the CS8900A.

5.1.2 Frame Encapsulation and Decapsulation

After configuration is complete, operation can begin. All aspects of Ethernet frame transmission and reception are handled by the CS8900A's MAC engine, which is fully compliant with the Institute of Electrical and Electronics Engineers (IEEE) 802.3 Ethernet standard (ISO/IEC 8802-3,1993) and supports full-duplex operation.

The main functions of the MAC are: frame encapsulation and decapsulation, error detection and handling, and, media access management. It assembles transmit packets and disassembles receive packets automatically.

For transmission, when the proper number of bytes has been transferred to the CS8900A's memory (either 5, 381, 1021 bytes, or 1518 bytes), and providing that access to the network is permitted, the MAC automatically transmits the 7-byte preamble (1010101b…), followed by the Start-of-Frame Delimiter (SFD, 10101011b), and then the serialized data.

For reception, the MAC receives the incoming packet as a serial stream of Non-Return to Zero (NRZ) data. Then it checks for the SFD. If the SFD is detected, the MAC assumes all subsequent bits are frame data. The Destination Address (DA) is read and compared to the criteria programmed into the address filter by the MAC. The frame is loaded into the CS8900A's memory if the DA passes the address filter. Figure 5.1 shows the format of Ethernet frame.

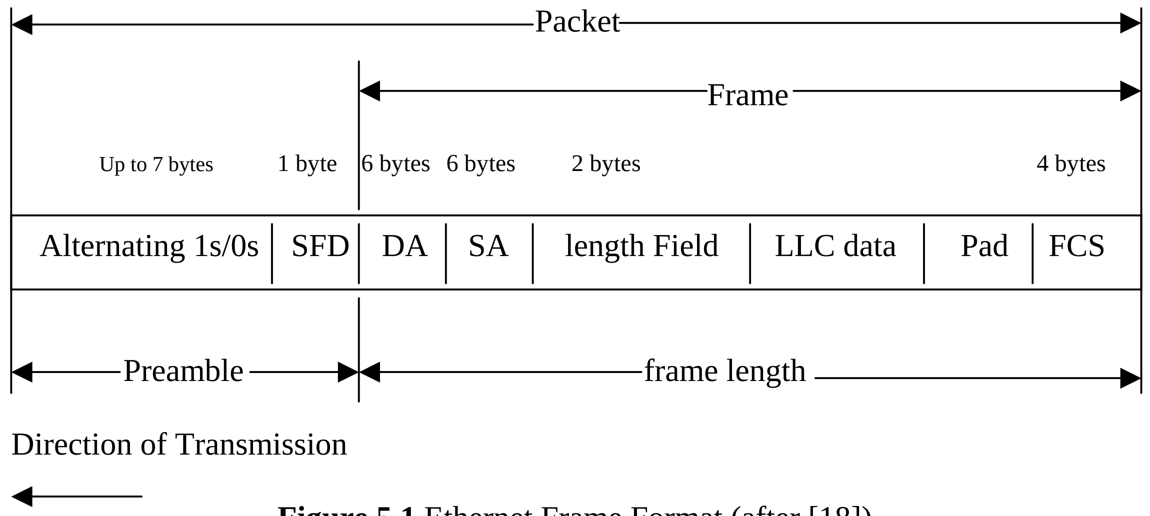


Figure 5.1 Ethernet Frame Format (after [18])

5.1.3 Two Basic Functions

As mentioned in section 5.1.1, the two basic functions of CS8900A are Ethernet packet transmission and reception. They will be briefly introduced in this section.

1. Packet Transmission

There are two phases in packet transmission. In the first phase, the Ethernet frame is moved into the CS8900A's buffer memory by the host. In the second phase of transmission, the frame is converted into an Ethernet packet then transmitted onto the network by the CS8900A.

2. Packet Reception

Like packet transmission, there are two phases in packet reception. In the first phase, an Ethernet packet is received and stored in on-chip memory by the CS8900A. In the second phase, the received frame is transferred across the ISA bus and into host memory by the host. An alternative to using ISA bus transfer is to use direct register addressing (I/O mode).

5.1.4 CS8900A Operation Modes

The receive frame can be transferred as Memory space operations, I/O space operations, or as DMA operations using host DMA (ISA bus).

1. DMA Mode Operation

   A direct interface to ISA buses running at clock rates from 8 to 11 MHz is provided by CS8900A. Its on-chip bus drivers are able to deliver 24 mA of drive current, which allows the CS8900A to drive the ISA bus directly, without added external “glue logic”.

   In order to minimize missed frames, the ISA-bus operation below 8 MHz should use the CS8900A’s Receive DMA mode.

2. Memory Mode Operation

   In Memory Mode operation, the CS8900A’s internal registers and frame buffers are mapped into a contiguous 4-Kbyte block of host memory, given that the host with direct access to the CS8900A’s internal registers and frame buffers.

3. I/O Mode Operation

   In I/O Mode operation, the CS8900A is accessed through eight, 16-bit I/O ports, which are mapped into sixteen contiguous I/O locations in the host system’s I/O space. Being the default configuration for the CS8900A, the I/O Mode is always enabled. Table 5.1 illustrates I/O Mode mapping.

Table 5.1 I/O Mode Mapping (after [18])

Offset	**Type**	**Description**
0000h	Read/Write	Receive/Transmit Data (port 0)
0002h	Read/Write	Receive/Transmit Data (port 1)
0004h	Write-only	TxCMD (Transmit Command)
0006h	Write-only	Txlength (Transmit Length)
0008h	Read-only	Interrupt Status Queue
000Ah	Read/Write	PacketPage Pointer
000Ch	Read/Write	PacketPage Data (Port 0)
000Eh	Read/Write	PacketPage Data (Port 1)

I/O Mode is 99.6% as fast as Memory Mode [19]. Cirrus Logic recommends the use of I/O Mode since the CS8900A defaults to I/O and no glue logic is needed in most systems in I/O mode. I/O mode is used in the Silicon Firewall.

5.1.5 PacketPage

PacketPage is a unique, highly-efficient method of accessing internal registers and buffer memory, which the CS8900A architecture is based on. A unified way of controlling the CS8900A in Memory mode or I/O mode that minimizes CPU overhead and simplifies software is provided by PacketPage.

Central to the CS8900A architecture is PacketPage memory, which is a 4-Kbyte page of integrated RAM. Transmit and receive frames are stored in PacketPage memory temporarily, also PacketPage memory is used for internal registers. Table 5.2 presents the user-accessible portion of the PacketPage memory.

Table 5.2 The User-Accessible Portion of The PacketPage Memory (after [18])

PacketPage Address	**Contents**
0000h-0045h	Bus Interface Registers
0100h-013Fh	Status and Control Registers
0140h-014Fh	Initiate Transmit Registers
0150h-015Dh	Address Filter Registers
0400h	Receive Frame Location
0A00h	Transmit Frame Location

The following tables illustrate the packet page memory address map.

Table 5.3 Bus Interface Registers (after [18])

PacketPage Address	**# of Bytes**	**Type**	**Description**
0000h	4	Read-only	Product Identification Code
0004h	28	-	Reserved
0020h	2	Read/Write	I/O Base Address
0022h	2	Read/Write	Interrupt Number (0,1,2,3)
0024h	2	Read/Write	DMA Channel Number (0,1,2)
0026h	2	Read/Only	DMA Start of Frame
0028h	2	Read-only	DMA Frame Count (12 Bits)
002Ah	2	Read-only	RxDMA Byte Count
002Ch	4	Read/Write	Memory Base Address Register
0030h	4	Read/Write	Boot PROM Base Address
0034h	4	Read/Write	Boot PROM Address Mask
0038h	8	-	Reserved
0040h	2	Read/Write	EEPROM Command
0042h	2	Read/Write	EEPROM Data
0044h	12	-	Reserved
0050h	2	Read only	Received Frame Byte Counter
0052h	174	-	Reserved

Table 5.4 Status and Control Registers (after [18])

PacketPage Address	**# of Bytes**	**Type**	**Description**
0100h	32	Read/Write	Configuration&Control Registers (2 bytes per register)
0120h	32	Read-only	Status& Event Registers (2 bytes per register)
0140h	4	-	Reserved

Table 5.5 Initiate Transmit Registers (after [18])

Packetpage Address	**# of Bytes**	**Type**	**Description**
0144h	2	Write-only	TxCMD (transmit command)
0146h	2	Write-only	TxLength (transmit length)
0148h	8	-	Reserved

Table 5.6 Address Filter Register (after [18])

PacketPage Address	# of Bytes	Type	Description
0150h	8	Read/Write	Logical Address Filter
0158h	6	Read/Write	Individule Address
015Eh	674	-	Reserved

Table 5.7 Frame Location (after [18])

PacketPage Address	# of Bytes	Type	Description
0400h	2	Read-only	RXStatus
0402h	2	Read-only	RxLength
0404h	-	Read-only	Receive Frame Location
0A00	-	Write-only	Transmit Frame Location

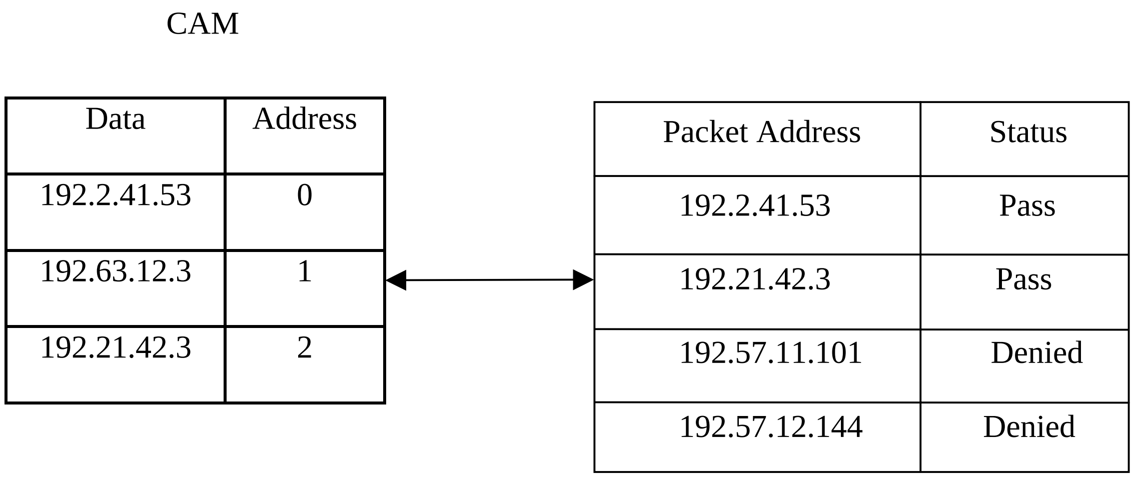
5.2 Using Content-Addressable Memory as an IP Packet Filter

An Internet Protocol (IP) packet filter is a security feature that prohibits unauthorized users from accessing local-area network (LAN) resources. IP traffic over a wide-area network (WAN) link can also be restricted by such a filter. LAN users can be restricted to specific applications on the Internet (such as e-mail) with an IP packet filter as well.

A Silicon Firewall serves to filter packet traffic by checking to determine if a packet is to be permitted or denied according to the desired polices and rules. Given the traffic rates now in place, for the Silicon Firewall to be effective, it must operate at high speed. The content-addressable memory (CAM) technology is incorporated to the Silicon Firewall in order to meet this goal.

5.3 Content-Addressable Memory

In the Silicon Firewall system, CAM is used as a filter to block all accesses except for packets that have permission. The source addresses that have permission are stored in CAM; when a source address is presented to the CAM, the CAM reports whether it contains the source address [20]. The source address(es) residing within CAM have permission for a particular activity. Figure 5.2 shows an example of an IP filter where the ultimate action is to “pass” or “deny” the packet depending on the source address.



Data	Address
192.2.41.53	0
192.63.12.3	1
192.21.42.3	2

Packet Address	Status
192.2.41.53	Pass
192.21.42.3	Pass
192.57.11.101	Denied
192.57.12.144	Denied

Figure 5.2 Using CAM as an IP Filter (after [20])

5.3.1 CAM and Traditional Memory Devices

Most memory devices address specific memory locations to store and retrieve data. For example, a system using RAM or ROM locates data by searching sequentially through memory. This technique can slow system performance because the search requires multiple clock cycles to complete. However, identifying stored data by content, rather than by its address can considerably reduce the time required finding an item stored in memory. CAM works in this way. CAM simultaneously compares the desired information against the entire list of pre-stored entries, so it offers a performance advantage over other memory search algorithms, such as binary-based searches, tree-based searches, or lookaside tag buffers.

RAM stores data at a particular address. Retrieving data from RAM, the system supplies the address and then receives the data. With CAM, the system supplies the data rather than the address, as shown in figure 5.3.

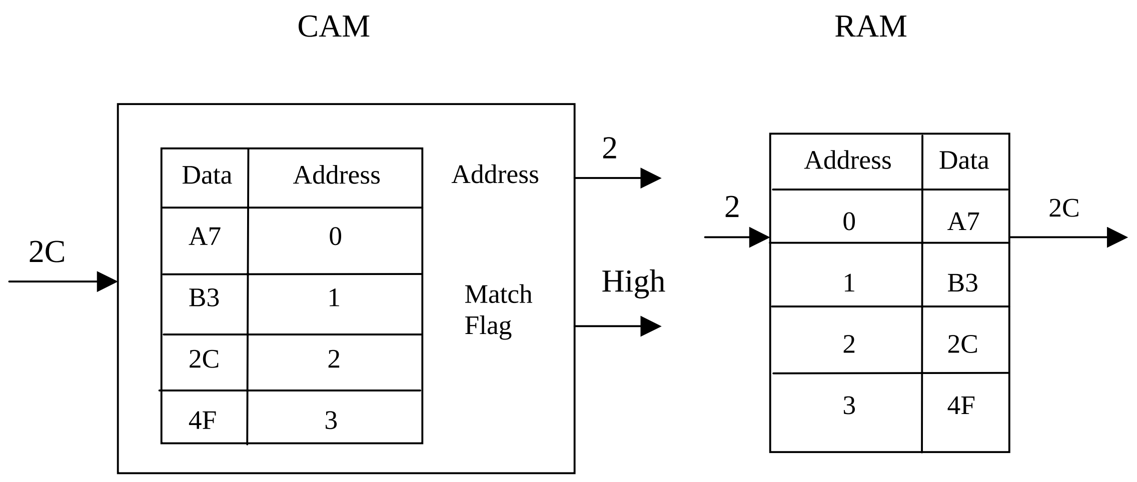


Figure 5.3 CAM vs. RAM (after [20])

CAM takes one clock cycle to search through all memory locations in parallel to locate stored data and returns the data's address. CAM provided by Altera drives a match flag high if the data is found, or low if the data is not found.

5.3.2 Advantages of CAM

A performance advantage is offered by CAM over other memory search algorithms, because it compares the desired information against the entire list of pre-stored entries simultaneously. CAM provides orders-of-magnitude reduction in the search time and helps much in data analysis and updating. Its typical match time is less than 10 ns for the Altera CAM megafunction.

To better understand the performance advantages of using CAM, the total time required to search for an item using both RAM and CAM can be compared. Locating an item in a 32-word, 32-bit RAM block running at 125 MHz requires up to 256 ns, as 32 clock cycles of 8 ns each may be needed to find a match. In contrast, the total time required to find an item in a similar-sized CAM block is only 4 ns, or 1 clock cycle of 8 ns. In this example, CAM is 32 times fast as RAM and has a latency of one clock cycle compared to a maximum of 32 clock cycles for RAM [21]. Note that regardless of the size of the CAM, the latency is always one clock cycle.

CAM is ideally suited for many applications, such as Ethernet address look-up, data compression, pattern recognition, cache tags, fast routing table look-up, high-bandwidth address filtering, user privileges, and security and encryption information.

5.3.3 Discrete CAM and Integrated CAM

Currently, discrete CAM devices are mostly used for applications that require fast searches. It increases design time and reduces the amount of usable PCB (printed circuit board) space because designers have to add a separate CAM device to their printed circuit board (PCB). Discrete CAM also reduces system performance because of the introduction of additional on-chip and off-chip delays. However, reconfigurable devices containing on-chip CAM built into their embedded system blocks (ESBs) eliminate the disadvantages of discrete CAM. Altera on-chip CAM has an access time of 4 ns, however, the access time for a typical discrete CAM is 20 ns [20]. Because CAM is integrated inside an FPGA device, it provides faster system performance than traditional discrete CAM. In this project, there are 52 embedded system blocks (ESBs) in the Altera EP20K200E device, allowing a maximum of 53,248 CAM bits. In this project, 94% ESB bits are used, not only by the CAM, some ESB bits are used by other functions, such as the SignalTap embedded logic analyzer and the on-chip read only memory (ROM). There are large external CAMs available, for example, the SiberCAM [22] Ultra-2M is a CAM with 2,359,296 ternary storage elements. Up to 16 Ultra-2M devices can be connected together to provide increased storage depth without performance degradation in search datapath throughput. Table 5.8 shows the comparison of discrete CAM and APEX CAM.

Table 5.8 Comparison of Discrete CAM & APEX CAM (after [20])

Feature	**Discrete CAM**	**APEX CAM**
Access time	20 ns	4 ns
System performance	Multi-device solution 28.2 ns	Single-device solution 4.9 ns

5.3.4 The Altcam Megafunction

CAM is implemented in the Quartus II software through the altcam megafunction. In order to describe how CAM works in the Silicon Firewall clearly, CAM is introduced in detail in this section.

5.3.4.1 Symbol

The symbol for the altcam megafunction is as below:

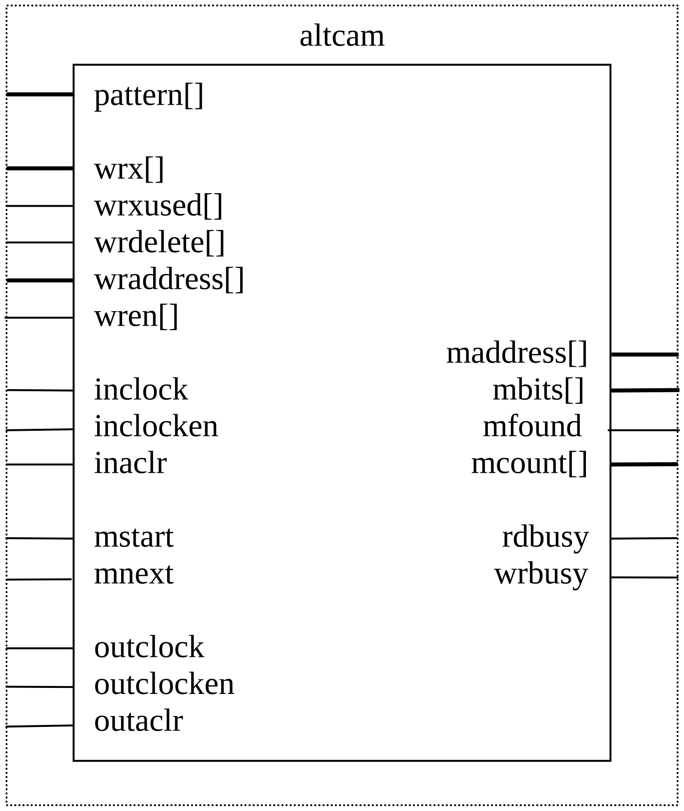


Figure 5.4 Symbol for The Altcam Megfunction (after [20])

5.3.4.2 Input Pins

Table 5.9 describes the input pins of the altcam megafunction.

Table 5.9 Input Pins of The Altcam Megafunction (after [20])

Port Name	Require	Description	Notes
pattern []	Yes	Input data pattern for searching or writing.	Input port WIDTH wide.
wrx []	No	Pattern “don’t care” bits (indicated with 1s), for writing only.	Input port WIDTH wide.
wrxused	No	Indicates whether wrx[] should be used.	If false, writing takes two clock cycles to complete; if true, writing takes three clock cycles. If asserted during a write cycle, the value of the wrx [] port is used. Otherwise, the value of the wrx[] port has no effect.
wrdelete	No	Indicates that the pattern at wraddress []	Deleting a pattern takes two clock cycles; pattern [], wrx[], and

		should be deleted.	wrxused are ignored during delete cycles.
wraddress[]	No	Address for writing.	Input port WIDTHAD wide.
wren	No	Write enable.	Assert wren to start to a write or delete operation. De-assert wren for a read (match) operation.
inclock	Yes	Clock for most inputs.	
inclocken	No	Clock enable for inclock.	
inaclr	No	Asynchronous clear for registers that use inclock.	
mstart	No	Multi-match mode only: indicates that a new CAM read is starting and forces maddress [] to first match.	This port is not available for single-match mode but reauired for multiple-match modes. In fast multiple-match mode, this port is required if the mnext port is used.
mnext	No	Multi-match mode only: advances maddress [] to next match.	This port is not available for single-match mode.
outclock	No	Clock for mstart, mnext, and outputs.	Used only if "OUTPUT_REG=OUTCLOCK". If "OUTPUT_REG=UNREGISTERED" or "INCLOCK" this port must remain unconnected.
outclocken	No	Clock enable for outclock.	Used only if "OUTPUT_REG=OUTCLOCK". If "OUTPUT_REG=UNREGISTERED" or "INCLOCK" this port must remain unconnected.
outaclr	No	Asynchronous clear for registers that use outclock.	

5.3.4.3 Output Pins

Table 5.10 describes the output pins of the altcam megafunction.

Table 5.10 Output Pins of The Altcam Megafunction (after [20])

Port	Required	Description	Comments
maddress[]	No	Encoded address of current match.	Output port WIDTHAD wide. One of the output ports must be used. Altera recommends using ether a combination of the maddress[] and mfound output ports, or the mbits [] output port.
mbits[]	No	Address of the found match.	Output port with width [NUMWORDS-1..0]. One of the output ports must be present. Altera recommends using either a combination of the maddress [] and mfound output ports, or the mbits [] output port.
mfound	No	Indicates at least one match.	Used with the maddress[] port. One of the output ports must be present. Altera recommends using either a combination of the maddress [] and mfound output ports, or the mbits [] output port.
mcount[]	No	Total number of matches.	Output port WIDTHAD wide. One of the output ports must be present. Altera recommends using either a output ports, or the mbits [] output port.
rdbusy	No	Indicates that read input ports must hold their current value.	One of the output ports must be present.
wrbusy	No	Indicates that write input ports must hold their current value.	One of the output ports must be present.

As mentioned previously, to accomplish packet filtering, a set of rules has to set up to specify what types of packets (e.g., those to or from a particular IP address or port) are to be allowed and what types are to be blocked. Since CAM is used to do the packet filtering in this project, we need to write the reference source address patterns into and then read the match results from CAM.

5.3.4.4 Writing Patterns into CAM

CAM can be pre-loaded with data either during configuration, or during systemoperation. In most cases, writing each word into CAM takes two clock cycles [20]. The “don’t care” bits can be written into CAM words and bits set to “don’t care” do not affect matching. A third clock cycle is required if “don’t care” bits are used [20].

5.3.4.5 Reading from CAM

Altera CAM operates in one of three different modes: single-match mode, multiple-match mode, and fast multiple-match mode. In each mode, the matched data’s location is outputted by an ESB as an encoded or unencoded address. In an encoded output, the address of the matched data is indicated. In an unencoded output, each output represents one word of the CAM block. The corresponding address is a match if an output goes high (e.g., if the data is located in address 14, the fourteenth output line goes high).

Single match mode is more suited for designs without duplicate data in the memory. If multiple locations in the memory contain the same data, CAM should be used in multiple-match or fast multiple-match mode. In these two modes, CAM supports multiple-match data and the ESB outputs the locations of the matched data as an encoded or unencoded addresses. Also, the CAM only takes one clock cycle to acquire outputs in single match mode, while in multiple-match mode two clock cycles are needed and fast multiple match while taking one clock cycle need twice as much ESB memory. Since there are no duplicate patterns in the Silicon Firewall, and speed and size are important, the single match mode is used in this project.

5.4 Silicon Firewall Hardware Mode

The hardware design is based on the Nios version 1.1 reference design. Basically, the hardware mode extracts the source IP address and uses a CAM to perform an address match.

5.4.1 Source IP Address Extraction

As mentioned previously, the source IP address is the policy of packet filtering in this research project, so source IP address extraction is the important part in this design. In this section, extraction of the source IP address will be discussed.

5.4.1.1 CS8900A

In software mode, the CS8900A is configured to interrupt mode. The interrupt signal from the CS8900A is very important in the hardware design, since all the actions are based on this signal. In the reference design, this signal is connected to the Nios CPU. In the Silicon Firewall system, it is connected to one of the Verilog modules. The function of this Verilog module is to read the incoming packet and extract the source IP address. It outputs this IP address to the CAM to perform an address match. This module communicates with both the CS8900A and the CAM megafunction.

The CS8900A is one of the peripherals for the Nios CPU in the reference design, so all the pins are set up already, based on that usage. The Verilog module uses all these pins to communicate with CS8900A, and no more pins are necessary.

When there is an interrupt, the hardware first reads a special register (the ISQ register) to cause the interrupt pin to go low. However, the interrupt pin will remain low until the null word (0000h) is read from the ISQ register, or for 1.6 us, whichever is longer. In order to make sure the Silicon Firewall will not miss any packets, the first method (null word from ISQ) must be used.

5.4.1.2 Source IP Address Generation

Three things are considered in the source IP address generation part of the design:

1. The source IP address is the only thing needed to make a decision, so reading the whole packet is not necessary.
2. The packet data is made of header and data, and the format of the header is fixed (see Chapter 2). The method used in this design is to count the bytes.
3. The two sets of 16-bit data are combined together to form a 32-bit IP address.

5.4.2 CAM

The function of CAM in this design is to do address matching, so both writing and matching CAM operations are needed. In software mode, the Nios CPU stores IP addresses in CAM (write); in hardware mode, the previously described Verilog module feeds the extracted IP address to CAM (matching).

Since the IP address is 32-bit, a 32 bit*32 words CAM is built through the Megawizard tool in Quartus II. It is then added to the system as a peripheral of the Nios CPU. In this design, since there are no don't care bits written into CAM, only four inputs are needed to write the IP address: inclock, pattern, wren, wraddress. Three outputs pins are used: outclock, maddress and mfound. The inclock and the outclock are same. Both of them are from the Altera APEX board (33.3333 MHz).

5.4.2.1 mfound

mfound is one of the outputs from CAM. It is used to indicate if there is a match. In the Silicon Firewall system, if mfound goes high, it means a packet with the authorized source IP address has arrived. In this case, the whole packet's data is wanted. The mfound signal is used to interrupt the Nios CPU to process the packet data.

In order to make sure mfound interrupts the Nios CPU properly (only goes high when there is a desired packet) two more input signals are added to CAM: inclocken and outclocken. When the Nios CPU initializes CAM, only the inclocken is high (activated). After the initialization, both of these two clock enable signals are set to high. Thus enables the CAM to perform source IP address matching.

5.4.3 Packet Data Transfer

In this design, both the Nios CPU and the hardware communicate with the CS8900A, and both the Nios CPU and the hardware communicate with the CAM. However, two signals cannot be connected to an output pin or input pin directly. Therefore there are two situations to deal with in the data transfer in this system.

1. For the data bus of the CS8900A, two bi-directional, tri-state buffers are used, one is for the Nios CPU and the other one is for the hardware.
2. For the other pins of the CS8900A and the "pattern" input pin of CAM, a bus multiplexer is used.

5.5 Silicon Firewall Software Mode

In the Silicon Firewall system, there are two operating modes: software mode and hardware mode. In this section, the software mode is introduced.

As mentioned in section 3.8, basically the software mode does initialization and configuration. Other than that, since there are two modes in this system, the task of switching between the software mode and hardware mode is done by the software.

5.5.1 Initialization

In order to make the Silicon Firewall system work properly, various parameters have to be written to the peripherals. In this specific system, there are two peripherals that need to be initialized, the CS8900A and the CAM.

5.5.1.1 CS8900A Initialization

The Silicon Firewall system design uses the example hardware Nios 32-bit CPU reference design for a single daughter card in the Nios EDK CD-ROM as the starting point. In the reference design, the CS8900A has been connected to the Nios CPU and configured already. After loading the reference design to the Altera FPGA, the Nios CPU can communicate with the CS8900A by running the example source file. In the SDK of the reference design, the "lib" folder contains the C program that initialized the CS8900A.

In the Silicon Firewall design, the configuration of the CS8900A was changed. There are two differences between the reference design configuration and Silicon Firewall system configuration.

1. In the reference design, the interrupt pin is not activated in the configuration program. However the interrupt from the CS8900A is a very important signal for the hardware in the Silicon Firewall system, so additional configuration data are written into the internal registers of the CS8900A to enable the interrupt pin.
2. In the reference design, the CS8900A is configured for promiscuous mode, in which case it will accept all receive frames, irrespective of DA (Destination Address). In the Silicon Firewall system, the CS8900A is configured for only broadcast mode and individual address mode. When the individual address

mode is set, frames with a DA that matches the individual address are accepted, while in broadcast mode, all broadcast frames are accepted. This last mode is necessary since broadcast frame are required in a number of protocols (i.e. Ping command using the ICMP protocol).

5.5.1.2 CAM Initialization

The CAM is another important peripheral in the Silicon Firewall system (see figure 3.2). CAM can be used to accelerate a variety of applications such as local-area networks (LANs), database management, file-storage management, table look up, pattern recognition, artificial intelligence, fully associative and processor-specific cache memories, and disk cache memories [20]. In the Silicon Firewall system, CAM is used to do address matching. In order to do that, the CAM has to be initialized, that is to say some data has to be stored in the CAM in advance. In this research project design, the Nios CPU writes a set of data (Source IP addresses) in the CAM before the operation of the system.

A CAM megafunction is built through the MegaWizard in Quartus II, and then it is added to the system. Since the CAM is a complex peripheral, 48 parallel I/O (input/output) pins are used to make the CAM work properly.

5.5.2 Polling vs. Interrupt

This design used Nios version 1.1 to begin with. The reference design is also 1.1-based. However the 1.1-based designs have a problem with the interrupt circuitry. The interrupt arrives at the Nios core asynchronous to the main CLK (clock) signal, causing occasional spurious interrupts to other values than the assigned interrupt (irq) number.

However, in the Silicon Firewall design, the Nios CPU is not intended to read all the packet data, only the packet data with the permitted source IP address is read. In that case, a signal from CAM will interrupt the Nios CPU, thus the interrupt feature of the Nios CPU is very important for this design. Fortunately, the Nios 2.0 core fixes this with a D-Flip-Flop between the CS8900A and the Nios, clocked by CLK and implements this fix internally.

Accordingly, there is a major change in the plugs library in Nios 2.0 CPU. That is the plugs library may now be run with interrupts enabled, polling is no longer necessary. From an CPU efficiency stand point this is very desirable.

5.5.3 Mode Switching

As mentioned previously, this design is a hardware and software codesign, so another concern is how to make the hardware and software work together properly and efficiently. After splitting the function of hardware and software in this system, a method to switch between hardware mode and software mode has to be provided. In this design, two PIOs (parallel input/output) are used to do this. One of the PIOs is defined as output by the Nios CPU and is connected to related hardware to make the hardware work correctly. The Nios CPU uses this signal as a control signal to switch between hardware mode and software mode. The other PIO is defined as input by the Nios CPU. In reality this input PIO is connected to the output PIO. The reason for doing this is as follows: the output PIO is used to control the hardware, however that is not enough since the Nios CPU will do different things in different modes. That is to say the Nios CPU has to "know" which mode it is in before it does something as requested.

After the interrupt is enabled in the Nios 2.0 CPU, there is an interrupt handler in the "lib" folder in the reference design. The interrupt service routine will check for events and dispatch packet when there is an interrupt. However, there are two modes in the Silicon Firewall design, so before the interrupt service routine does something, the current mode has to be determined. The input PIO is used to do this.

5.5.4 Flow Chart

The following is the flow chart of the software mode. In this figure, "Hw" means hardware mode, "Sw" means software mode.

The Silicon Firewall system uses software mode to begin with. After the system initialization, the Silicon Firewall is ready to function. In the flow chart, the interrupt could come from two places, the first one is directly from the CS8900A; the second one is from the Silicon Firewall system hardware when there is an authorized packet coming in.

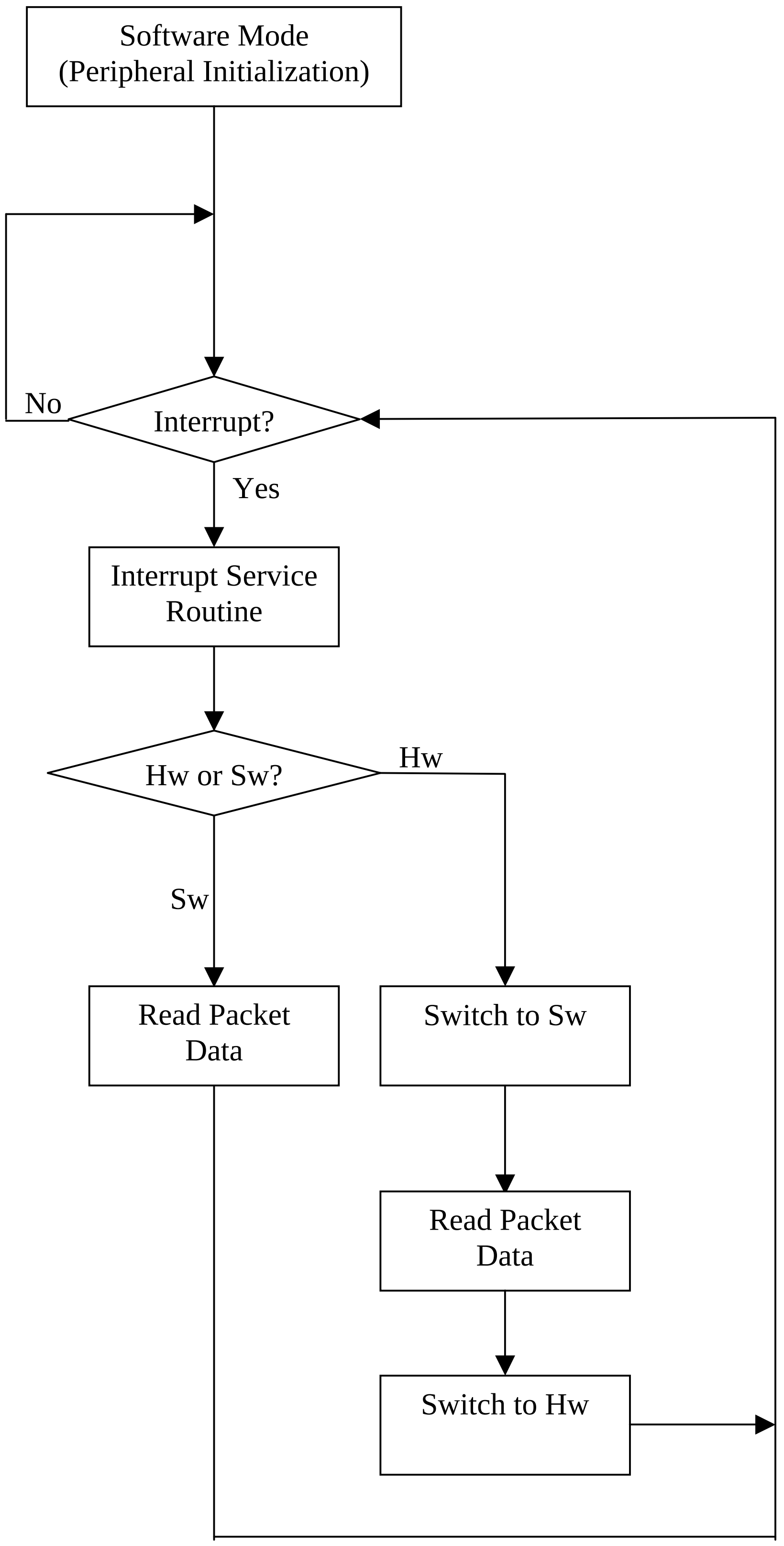


Figure 5.5 Software Mode Flow Chart

5.6 Silicon Firewall Design

Based on the introduction to the CS8900A and the CAM, the implementation in the Silicon Firewall system will now be introduced in detail.

5.6.1 Nios System to Daughter Card Pin Map

Each Ethernet Interface (CS8900A) peripheral in the Nios system will have an associated set of I/O pins on the system module. This section describes how to connect the daughter card to the system-module I/O pins. In general, these connections are established by making pin-assignments in the PLD design. The Silicon Firewall design just uses the pin-assignment included in the reference designs that it is based on.

The names given to the system-module I/O ports will depend on the name for the Ethernet Interface (CS8900A) peripheral. In Tables 5.11, <*your_name*> indicates the name assigned to this component, in the Silicon Firewall system it is “enet”. The name for some system-module I/O ports will also depend on the tri-state bus selected for this peripheral. In Tables 5.11, <*your_bus_name*> indicates the name of the bus assigned to this Ethernet Interface (CS8900A) peripheral, in the Silicon Firewall it is “nedk_card_bus”.

Table 5.11 Nios 32-bit CPU System Module I/O Port Name and Daughter Card Pin Name (after [17])

32-bit CPU System Module I/O Port Name	**Daughter Card Pin Name (Lower of Two Stacked Cards)**
<*your_bus_name*>_data	SD [15..0]
<*your_bus_name*>_address[4]	SA [3]
<*your_bus_name*>_address[3]	SA [2]
<*your_bus_name*>_address[2]	SA [1]
<*your_bus_name*>_byteenablen[1]	SHBE_n
ior_n_to_the <*your*_name>	IOR_n (lower)
iow_n_to_the <*your*_name>	IOW_n (lower)
irq_to_the_<*your_name*>	INTRQ0 (lower)
~(system module reset_n)	RESET
Constant Logic-1	MEMW_n
Constant Logic-1	MEMR_n
Constant Logic-1	SA [9..8]
Constant Logic-0	SA [11..10]
Constant Logic-0	SA [7..4]
Constant Logic-0	SA [0]
Constant Logic-0	CHIPSEL_n (lower)
Constant Logic-0	CHIPSEL_n (upper)

The descriptions for these pins are as follows (according to [18]):

SD [0:15]: System Data Bus, Bi-Directional with 3-State Output pins 65-68, 71-74, 27-24, 21-28. Bi-directional 16-bit System Data Bus used to transfer data between the CS8900A and the host.

SA [0:19]: System Address Bus, Input pins 37-48, 50-54, 58-60. Lower 20 bits of the 24-bit System Address Bus used to decode accesses to CS8900A I/O and Memory space, and attached Boot PROM. SA0-SA15 are used for I/O Read and Write operations. SA0-SA19 are used in conjunction with external decode logic for Memory Read and Write operations.

SBHE_n: System Bus High Enable, Input pin 36. Active-low input indicates a data transfer on the high byte of the System Data Bus (SD8-SD15). After a hardware or a software reset, provide a HIGH to LOW and then LOW to HIGH transition on SBHE signal before any I/O or memory access is done to the CS8900A.

IOR_n: I/O Read, Input pin 61. When IOR_n is low and a valid address is detected, the CS8900A outputs the contents of the selected 16-bit I/O register onto the System Data Bus. IOR_n is ignored if REFRESH_n is low.

IOW_n: I/O Write, Input pin 62. When IOW_n is low and a valid address is detected, the CS8900A writes the data on the system Data Bus into the selected 16-bit I/O register. IOW_n is ignored if REFRESH_n is low.

INTRQ [0:3]: Interrupt Request, 3-State pins 30-32, 35. Active-high output indicates the presence of an interrupt event. Interrupt Request goes low once the Interrupt Status Queue (ISQ) is read as all 0's. Only one Interrupt Request output is used (one is selected during configuration). All non-selected Interrupt Request outputs are placed in a high-impedance state.

RESET: Reset, Input pin 75. Active-high asynchronous input used to reset the CS8900A. Must be stable for at least 400 ns before the CS8900A recognizes the signal as a valid reset.

MEMW_n: Memory_Read, Input pin 29. Active-low input indicates that the host is excecuting a Memory Read operation.

MEMR_n: Memory Write, Input pin 28. Active-low input indicates that the host is excecuting a Memory Write operation.

CHIPSEL_n: Chip Select, Input pin 7. Active-low input generated by external Latchable Address bus decode logic when a valid memory address is present on the ISA bus. If Memory Mode operation is not needed, CHIPSEL_n should be tied low. The CHIPSEL_n is ignored for I/O and DMA mode of the CS8900A.

In the Silicon Firewall system, the pins used are presented in table 5.12.

Table 5.12 The Pins of CS8900A Used in The Silicon Firewall System

CS8900A Pin Name	**Silicon Firewall Pin Name**
SD [0:15]	NEDK_data [15..0]
SA [0:19]	NEDK_reg_address [2..0]
IOR_n	NEDK_L_IOR_n
IOW_n	NEDK_L_IOW_n
SBHE_n	/NEDK_SBHE_n
INTRQ [0:3]	NEDK_L_IRQ
RESET	NEDK_RESET

In I/O mode all the register numbers are even numbers. That is to say the least significant bits are all “0”. Thus in the Silicon Firewall design the least significant bit is connected to ground, and only three bits are needed. The Silicon Firewall system uses “INTRQ0” as the interrupt pin.

5.6.2 System Operation

After compiling the Silicon Firewall design successfully, one programming file is generated by the Quartus II compiler, which can be used to program or configure the APEX device. The configuration data is downloaded into a flash memory device on the Nios development board over a serial port. Then the APEX device is configured using the data stored in flash memory.

First start the “bash” shell, which is an UNIX command shell that allows running the “nios-build” and “nios-run” utility on the Nios development board provided with the GNUPro Nios software development tools. The “inc” and “lib” directories are subdirectories in the “SDK” folder, which contain a memory map and peripheral structures based on the memory layout and particular peripherals of the Silicon Firewall system. Since the plugs library is changed, so it has to be rebuilt using the “make –s all” command in the “lib” directory. A typical session of library rebuild is shown below.

```
[nios]$ make -s all
2003.10.19.00:03:38 --- Deleting libnios32.a libnios32_debug.a
2003.10.19.00:03:38 --- Removing objects
2003.10.19.00:03:41 --- Compiling cs8900.c
2003.10.19.00:03:43 --- Compiling flash_AMD29LV800.c
2003.10.19.00:03:44 --- Assembling nios_atexit.s
2003.10.19.00:03:44 --- Assembling nios_copyrange.s
2003.10.19.00:03:45 --- Assembling nios_cstubs.s
2003.10.19.00:03:45 --- Assembling nios_cwpmanager.s
2003.10.19.00:03:45 --- Compiling nios_debug.c
2003.10.19.00:03:46 --- Assembling nios_delay.s
2003.10.19.00:03:46 --- Assembling nios_emulator.s
2003.10.19.00:03:46 --- Compiling nios_gdb_stub.c
2003.10.19.00:03:47 --- Compiling nios_gdb_stub_io.c
2003.10.19.00:03:48 --- Assembling nios_gdb_stub_isr.s
2003.10.19.00:03:48 --- Assembling nios_getctlreg.s
2003.10.19.00:03:48 --- Compiling nios_gprof.c
2003.10.19.00:03:49 --- Assembling nios_isrmanager.s
2003.10.19.00:03:49 --- Assembling nios_jumptoreset.s
2003.10.19.00:03:49 --- Assembling nios_jumptostart.s
2003.10.19.00:03:50 --- Assembling nios_math1.s
2003.10.19.00:03:50 --- Compiling nios_printf.c
2003.10.19.00:03:50 --- Assembling nios_setjmp.s
2003.10.19.00:03:50 --- Assembling nios_setup.s
2003.10.19.00:03:51 --- Compiling nios_sprintf.c
2003.10.19.00:03:51 --- Assembling nios_zerorange.s
2003.10.19.00:03:52 --- Compiling pio_lcd16207.c
2003.10.19.00:03:56 --- Assembling pio_showhex.s
2003.10.19.00:03:56 --- Compiling plugs.c
2003.10.19.00:03:59 --- Compiling plugs_print.c
2003.10.19.00:04:00 --- Assembling timer_milliseconds.s
2003.10.19.00:04:01 --- Assembling uart_rxchar.s
2003.10.19.00:04:01 --- Assembling uart_txchar.s
2003.10.19.00:04:01 --- Assembling uart_txcr.s
2003.10.19.00:04:01 --- Assembling uart_txhex.s
2003.10.19.00:04:01 --- Assembling uart_txhex16.s
2003.10.19.00:04:01 --- Assembling uart_txhex32.s
2003.10.19.00:04:02 --- Assembling uart_txstring.s
```

```
2003.10.19.00:04:02 --- Building libnios32.a
2003.10.19.00:04:02 --- Compiling cs8900.c
```

Then in the “src” directory, the “nios-build” utility compiles the C code in the “sfw.c” file. Finally an executable file “sfw.srec” is generated. A typical session of “nios-build sfw.c” is shown below.

```
[nios]$ nios-build sfw.c

--------------------
Beginning Build
--------------------

Sources:
    sfw_menu.c
    sfw.c

# 2003.10.24 21:24:52 (*) nios-elf-gcc  -I ../inc -I ../../inc -I ../../../inc -
I ../../../../inc -I ../../../../../inc -g -O2 -m32 sfw_menu.c -o sfw_menu.c.o -
c

# 2003.10.24 21:24:52 (*) nios-elf-gcc  -I ../inc -I ../../inc -I ../../../inc -
I ../../../../inc -I ../../../../../inc -g -O2 -m32 sfw.c -o sfw.c.o -c

# 2003.10.24 21:24:53 (*) nios-elf-ld -e _start -u _start -g -T /cygdrive/c/alte
ra/excalibur/sopc_builder_2_5/bin/nios.ld  ../lib/obj32/nios_jumptostart.s.o sfw
_menu.c.o sfw.c.o --start-group -l nios32 -l c -l m -l gcc -l c -l nios32 --end-
group -L/cygdrive/c/altera/excalibur/sopc_builder_2_5/bin/nios-gnupro/nios-elf/l
ib/m32 -L/cygdrive/c/altera/excalibur/sopc_builder_2_5/bin/nios-gnupro/lib/gcc-l
ib/nios-elf/2.9-nios-010801-20020103/m32 -L../lib -L../../lib -L../../../lib -L.
./../../../lib -L../../../../../lib -L../inc -L../../inc -L../../../inc -L../../
../../inc -L../../../../../inc -L. -o sfw.out

# 2003.10.24 21:24:54 (*) nios-elf-objcopy -O srec sfw.out sfw.srec

# 2003.10.24 21:24:54 (*) nios-elf-nm sfw.out | sort > sfw.nm
# 2003.10.24 21:24:54 (*) nios-elf-objdump -D --source sfw.out > sfw.objdump

--------------------
Finishing Build
--------------------
```

The “nios-run” utility download the “sfw.srec” program over the serial port and runs it in the Nios_based Silicon Firewall system module. For a second download, the “clear” button (SW2) has to be pressed first. In the Silicon Firewall design, the “sfw.srec” program is used to initialize the whole system and control the hardware-software mode switching. When downloading the “sfw.srec” program is complete, the Silicon Firewall system is ready to receive the packet data. By default the system is in software mode; it can be switched to hardware mode by press the “q” button of the keyboard. The program keeps running to receive packet data unless “ctrl+c” is pressed to stop it. Figure 5.6 shows the complete system; the Nios development board and the computer that the software (NDK, NEDK) is installed on. The big window on the screen is Quartus II window; the small black window is the “bash” shell window. Figure 5.7 shows the picture of the Nios Development Board with the daughter card attached. Figure 5.8 illustrates the sequence of events of packet reception.

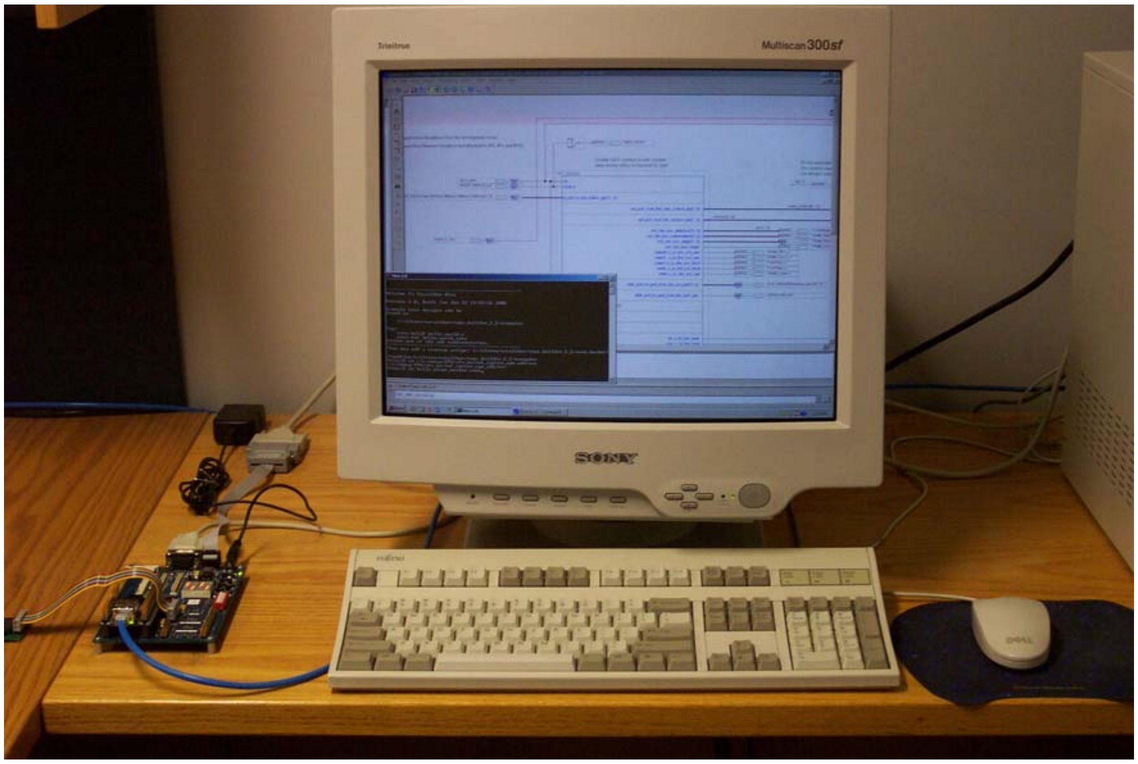


Figure 5.6 Silicon Firewall Design System

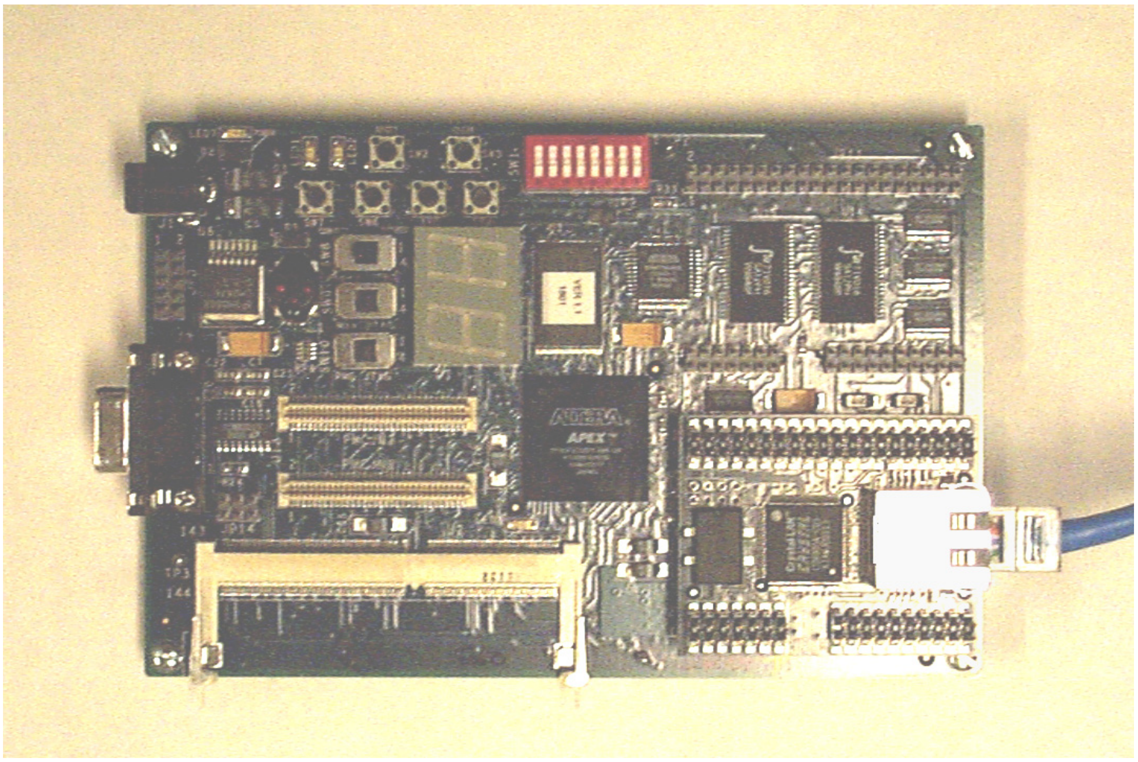


Figure 5.7 Nios Development Board With The Attached Daughter Card

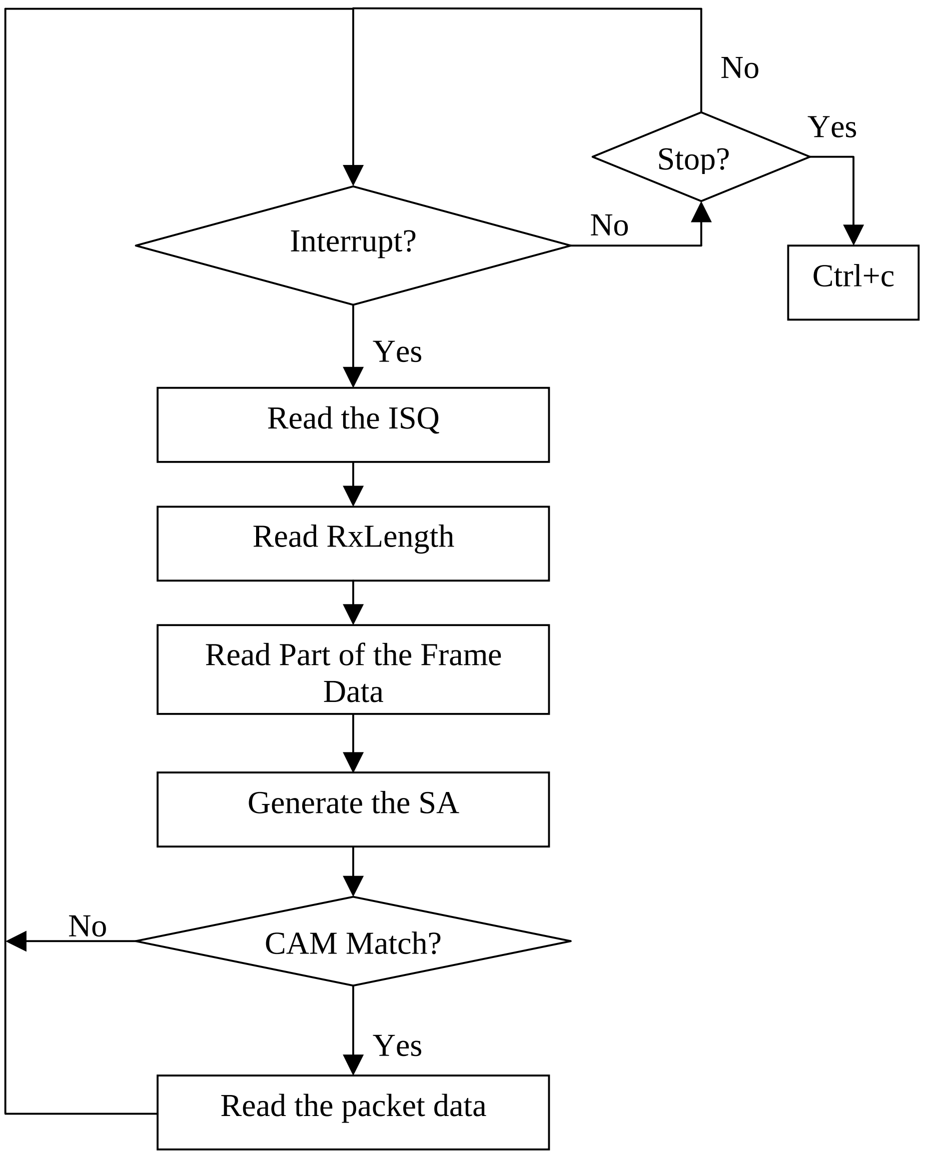


Figure 5.8 Sequence of Events of Packet Reception

5.6.3 Pin and Register Manipulation In The Software Configurations

In the following only those bits in registers that are used are shown and discussed. The interested reader is referred to [18].

The “sfw.c” implements the low-level routines for an adapter that the "plugs" embedded TCP/IP stack can use. This file resets the chip to a usable state. This involves some pin manipulation, and then some register manipulation.

First, the reset pin is raised and lowered, and then the "byte high enable (bhe)" is forced to undergo multiple-transitions. Multiple logic-transitions on the SBHE_n (byte-enable) pin are used to set the CS8900A in the appropriate communications-mode. This is the first thing that needs to be done. This sequence (a group of byte-writes) results in multiple edge-transitions on the CS8900's byte-enable input. After this, some internal registers are written.

1. RxCTL: Receiver Control (Read/Write)

7	6	5	4	3	2	1	0
F	E	D	C	B	A	9	8
				BroadcastA	IndividualA		RxOKA

The value is “0000_1101_0000_0000”. The three bits are set are as follows:

- RxOKA

  When set, the CS8900A accepts frames with correct CRC and valid length (valid length is: 64 bytes<=length<= 1518 bytes).

- BroadcastA

  When set, receive frames are accepted if the Destination Address is FFFF FFFF FFFFh.

- IndividualA

  When set, receive frames are accepted if the Destination Address matches the Individual address found at PacketPage base + 0158h to PacketPage base + 015Dh.

In the reference design, promiscuous mode is used. In the Silicon Firewall design, broadcast and individual address are used, because only the packet going to the Nios CPU will be examined, there is no concern about any other packets in the network.

2. RxCFG: Receiver Configuration (Read/Write)

7	6	5	4	3	2	1	0
F	E	D	C	B	A	9	8
	ExtradataiE	RuntiE	CRCerroriE				RxOKiE

The Value is" 0111_0001_0000_0000". The four bits are set are as follows:

- ExtradataiE

  When set, there is an Extradata Interrupt if a frame is longer than1518 bytes. The operation of this bit is independent of the received packet integrity (good or bad CRC).

- RuntiE

  When set, there is a Runt Interrupt if a frame is received that is shorter than 64 bytes. The CS8900A always discards any frame that is shorter than 8 bytes.

- CRCerror iE

  When set, there is a CRC error Interupt if a frame is received with a bad CRC.

- RxOKiE

  When set, there is an RxOK Interrupt if a frame is received without errors. RxOK interrupt is not generated when DMA mode is used for frame reception.

3. LineCTL (Read/Write)

7	6	5	4	3	2	1	0
SerTxON	SerRxON						
F	E	D	C	B	A	9	8

The value is “0000_0000_1100_0000”. The two bits are set are as follows:

- SerRxON

  When set, the receiver is enabled. When clear, no incoming packets pass through the receiver.

- SerTxON

  When set, the transmitter is enabled. When clear, no transmissions are allowed.

4. BusCTL (Read/Write)

7	6	5	4	3	2	1	0
F	E	D	C	B	A	9	8
EnableIRQ							

The value is “1000_0000_0000_0000”. The bit to be set is:

- EnableIRQ

  When set, the CS8900A will generate an interrupt in response to an interrupt event. When clear, the CS8900A will not generate any interrupt.

5.6.4 CS8900A Interrupt

The interrupt request signal is a very important signal for use in the Silicon Firewall. It will be discussed in this section.

5.6.4.1 Interrupt Activation

The CS8900A has four interrupt request output pins that can be connected directly to any four of the ISA bus Interrupt Request signals. Only one interrupt output is used at a time. It is selected by writing the interrupt number (0 to 3) into PacketPage Memorybase + 0022h. Unused interrupt request pins are placed in a high-impedance state. The selected interrupt request pin goes high when an enabled interrupt is triggered. The pin goes low after the first read of the Interrupt Status Queue (ISQ). In this research project design, INTRQ0 is used, so “0” is written into PacketPage Memorybase + 0022h to activate it.

5.6.4.2 Interrupt Re-enable

Enabling the interrupt is not enough, the interrupt only goes high once after that. In order to make the Silicon Firewall handle all the interrupts, the interrupt pin must be re-enabled after each event. This is done when the Interrupt Status Queue (ISQ) is read as all 0’s.

The Interrupt Status Queue (ISQ) is used by the CS8900A to communicate Event reports to the host processor. Whenever an event occurs that triggers an enabled interrupt, the CS8900A sets the appropriate bit(s) in one of the five registers, maps the contents of that register to the ISQ, and drives the selected interrupt request pin high. When the host services the interrupt, it must first read the ISQ to learn the nature of the interrupt. It can then process the interrupt.

Three of the registers mapped into the ISQ are event registers: RxEvent, TxEvent, and BufEvent. The other two registers are counter-overflow reports: RxMISS and TxCOL. There may be more than one RxEvent report and/or more than one TxEvent report in the ISQ at a time. However, there may be only one BufEvent report, one RxMISS report and one TxCOL report in the ISQ at any one time.

Event reports stored in the ISQ are read out in the order of priority, with RxEvent first, followed by TxEvent, BufEvent, RxMiss, and the TxCOL. The host needs to read from the ISQ to get the interrupt currently at the front of the queue. In I/O mode, the ISQ is located at I/O base + 0008h. Each time the host reads the ISQ, the bits in the corresponding register are cleared and the next report in the queue moves to the front.

When the host starts reading the ISQ, it must read and process all the event reports in the queue. A read out of a null word (0000h) indicates that all the interrupts have been read. After this null readout the interrupt is re-enabled.

In this research project, RxEvent is the only event. There are different ways to reading out of a null word, for example, (a). read ISQ, read RxEvent, read ISQ; (b). read ISQ, read RxLength, read the packet based on the RxLength, read ISQ; (c) read ISQ, read RxLength or RxStatus, Set SKIP_1 bit in RxCFG, read ISQ.

The condition in case (a) is called an "implied skip", which is not recommended by Cirrus Logic. In the Silicon Firewall design, the hardware only extracts the IP address and the task of reading the whole packet is done by the Nios CPU. Therefore case (b) is not proper for this design. Case (c) is adopted in the hardware design. Note that the "Set SKIP_1 bit" will cause the last committed received frame to be deleted from the receive buffer, since the Nios CPU has not read the whole packet, the hardware ignores this part. As well, until the Nios CPU reads the whole packet, the RxEvent can not be cleared by reading the ISQ. Therefore the last "Read ISQ" is ignored by the hardware part since the Nios CPU will perform this function.

5.6.5 CS8900A and Silicon Firewall Hardware

The reader is referred to appendix A and appendix C.

When there is an interrupt, the hardware first reads the ISQ register, then it reads the RxLength Register, and then it reads the packet data 16 times continuously. The CS8900A's internal architecture is based on a 16-bit data bus. Since the packet format is fixed and the source IP address is in words 15 and 16, a counter is used to count the 16-bit data words, and the 15^{th} and 16^{th} 16-bit data will be combined together to form the 32-bit IP Source Address (SA).

Reading the RxLength register means accessing the internal register. To access any of the CS8900A's internal registers in I/O mode, the host must first setup the PacketPage Pointer [23]. It does this by writing the PacketPage address of the target register to the PacketPage Pointer Port (I/O base + 000A). The contents of the target register is then mapped into the PacketPage Data Port (I/O base + 000Ch). This has to be done according to the switching characteristics of the CS8900A. Table 5.13 and Figure 5.9 show the 16-bit I/O read and write switching characteristics. The system clock is 33.3333 MHz, so another counter is used to count the clock cycles, so that individual timing requirements can be met.

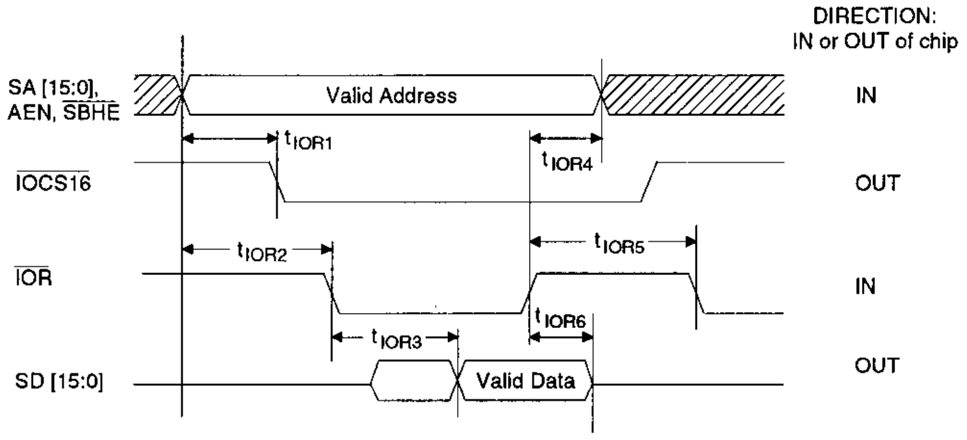
Table 5.13 Switching Characteristics of CS8900A (from [18])

(a). 16-Bit I/O Read

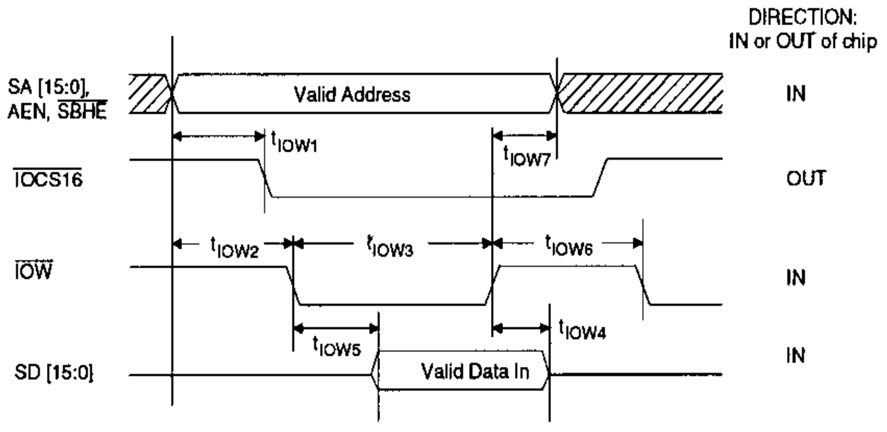
Parameter	Symbol	Min	Typ	Max	Unit
16-Bit I/O Read, IOCHRDY Not Used					
Address, AEN, $\overline{\text{SBHE}}$ active to $\overline{\text{IOCS16}}$ low	t_{IOR1}	-	-	35	ns
Address, AEN, $\overline{\text{SBHE}}$ active to $\overline{\text{IOR}}$ active	t_{IOR2}	10	-	-	ns
$\overline{\text{IOR}}$ low to SD valid	t_{IOR3}	-	-	135	ns
Address, AEN, $\overline{\text{SBHE}}$ hold after $\overline{\text{IOR}}$ inactive	t_{IOR4}	0	-	-	ns
$\overline{\text{IOR}}$ inactive to active	t_{IOR5}	35	-	-	ns
$\overline{\text{IOR}}$ inactive to SD 3-state	t_{IOR6}	-	30	-	ns

(b). 16-Bit I/O Write

16-Bit I/O Write					
Address, AEN, $\overline{\text{SBHE}}$ valid to $\overline{\text{IOCS16}}$ low	t_{IOW1}	-	-	35	ns
Address, AEN, $\overline{\text{SBHE}}$ valid to $\overline{\text{IOW}}$ low	t_{IOW2}	20	-	-	ns
$\overline{\text{IOW}}$ pulse width	t_{IOW3}	110	-	-	ns
SD hold after $\overline{\text{IOW}}$ high	t_{IOW4}	0	-	-	ns
$\overline{\text{IOW}}$ low to SD valid	t_{IOW5}	-	-	10	ns
$\overline{\text{IOW}}$ inactive to active	t_{IOW6}	35	-	-	ns
Address hold after $\overline{\text{IOW}}$ high	t_{IOW7}	0	-	-	ns



(a) 16-Bit I/O Read



(b) 16-Bit I/O Write

Figure 5.9 Switching Characteristics of CS8900A (from [18])

5.7 Nios System Components

Table 5.14 lists the system components.

Table 5.14 System Components of Silicon Firewall System

Module Name	Description	Bus Type	Base	Width of PIO	IRQ
Ref_system_cpu	Altera Nios 2.0 CPU	Avalon	0x000		
Boot_monitor_rom	On-Chip memory (RAM or ROM)	Avalon	0x400		
UART_1	UART (RS-232 serial port)	Avalon	0x420		26
Seven_seg_pio	PIO (Parallel I/O)	Avalon	0x440	16	
Timer1	Interval timer	Avalon	0x460		25
Led1_pio	PIO (Parallel I/O)	Avalon	0x470	1	
Button_pio	PIO (Parallel I/O)	Avalon	0x480	12	27
Lcd_pio	PIO (Parallel I/O)	Avalon	0x40000	11	
Ext_ram	SRAM (one or two IDT71V016 chips)	Avalon_tristate	0x100000		
Ext_flash	Flash memory	Avalon_tristate	0x500		
enet	Ethernet Interface (CS8900)	Avalon_tristate	0x520		30
Pattern_pio	PIO (Parallel I/O)	Avalon	0x530	32	
Cam_control_pio	PIO (Parallel I/O)	Avalon	0x540	6	
Mfound_pio	PIO (Parallel I/O)	Avalon	0x550	1	
Maddress_pio	PIO (Parallel I/O)	Avalon	0x560	5	
Sel_control_pio	PIO (Parallel I/O)	Avalon		1	
Ext_ram_bus	Avalon Tri-state bus	Avalon_tristate|avalon			
Nedk_card_bus	Avalon Tri-state bus	Avalon_tristate|avalon			
Clocken_pio	PIO (Parallel I/O)	Avalon	0x430	2	
Mode_pio	PIO (Parallel I/O)	Avalon	0x570	1	

In this table, Mfound_pio is used to interrupt the Nios CPU when there is a match. Sel_control_pio is used to control the mode of the Silicon Firewall in hardware. The Nios CPU also read the Sel_control_pio back when there is an interrupt. In this case it is called Mode_pio.

5.8 Software-Hardware Switching Mechanism

During the Silicon Firewall operation, both the Nios CPU and the "sfw" module need to communicate with the CS8900A. Both the Nios CPU and the "sfw" read packet data from CS8900A. Since only one signal can be connected to an output or bidirectional pin, three 2-input bus multiplexers are used to connect the three pins (NEDK_reg_address[2..0], NEDK_L_IOR_n, NEDK_L_IOW_n) to the Nios CPU and the CS8900A. This is also true for the pattern input signal of CAM. The two inputs of the bus multiplexer are from Nios CPU and “sfw” respectively.

Since the NEDK_data [15..0] pins are bi-directional pins, two tri-state bus megafunction were used for the connection. One is used for the Nios CPU, the other is used for the “sfw” module. There are four control signals: sw_r_enable, sw_w_enable; hw_r_enable, hw_w_enable. Four more 2-input bus multiplexers are used to generate these four control signals. For the two multiplexers output sw_r_enable and sw_w_enable, one of the inputs is from Nios, the other input is connected to “Vcc”; For the two multiplexers output hw_r_enable and hw_w_enable, one of the inputs is from “sfw”, the other input is connected to “Vcc”. In total eight 2-input bus-multiplexers are used in the Software-Hardware switching mechanism. All of them use the same control signal, which comes from Nios CPU through the “sel_control_pio” peripheral. In the Silicon Firewall design, “1” means hardware mode; “0” means software mode.

Chapter 6 TESTING AND RESULTS

The SignalTap logic analyzer and Ethereal network analyzer are used to analyze the results of this project. The Ping command is used to test the Silicon Firewall system. All of them will be introduced before the further discussion of the testing and results.

6.1 SignalTap Overview

The SignalTap® logic analyzer [23] is a megafunction that captures signals from any internal node or I/O pin of an APEX II or APEX 20K device in real-time at system speed. Also, the SignalTap analysis eliminates the need for external probes and design file changes to capture signals from an internal node and works with all existing EDA synthesis tool design flows. Both the logic analyzer controls and signal capture display are accessible from the Quartus II design software. Data transfer between the APEX II or APEX 20K device and the Quartus II software for waveform display of signals captured by SignalTap logic analysis is supported by the MasterBlaster™ or ByteBlasterMV™ communications cables. Figure 6.1 shows the SignalTap logic analyzer.

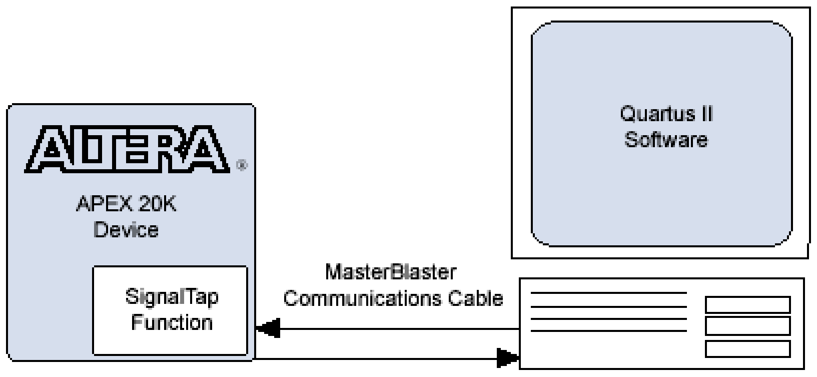


Figure 6.1 SignalTap Logic Analyzer (from [23])

6.1.1 Functional Description

Generally, the SignalTap megafunction is an embedded logic analyzer that provides access to signals inside an APEX II or APEX 20K device. The embedded logic analyzer function can be parameterized to capture up to 128 signals from internal nodes or I/O pins in-system and at system speed. From within the Quartus II software, the following items can be selected: which signals will be captured, when signal capture starts, and how many samples of data are captured. Also, the captured data can be stored in APEX II or APEX 20K embedded system block (ESB) RAM, or be sent to I/O pins for capture by external analysis equipment. Two things will be done to the data stored in ESB RAM, first it is transferred to a host computer by using the MasterBlaster or ByteBlasterMV communication cable. then it can be displayed in the SignalTap waveform viewer. The SignalTap logic analyzer can be automatically instantiated by the Quartus II software without making changes to user design files.

1. Assigning a Signal to the SignalTap File

   As mentioned previously, signals can be captured from any internal device node or I/O pins by the SignalTap analyzer. However, before signals capturing, the internal nodes or I/O pins must be assigned to SignalTap analyzer input channel. The SignalTap analyzer can capture from 1 to 128 internal nodes or I/O signals. The SignalTap analyzer uses more LEs as more signals are captured.

2. Filter Control

   The Filter Control dialog box allows selecting signals from a specific instance in the design to be displayed in the SignalTap window. The signals that are displayed can be managed by the Filter Control option.

3. Selecting an Acquisition Clock Signal

   All input channels are sampled on the rising edge of the acquisition clock signal, which must be a device signal. Using a global clock signal as the acquisition signal is recommended by Altera.

4. Setting the Sample Buffer Depth

   The sample buffer depth controls the amount of data the SignalTap analyzer captures when using the internal RAM configuration. More ESBs are used as more signals are captured and the sample buffer depth is increased.

5. Setting the Triggering Position

   A Trigger Position setting allows specifying the amount of data captured by the SignalTap logic analyzer that should be acquired before the trigger and the amount that should be acquired after the trigger. Figure 6.2 shows the circular buffer where the acquired data is placed in. The SignalTap logic analyzer continues sampling the input signals to capture post-trigger data when triggered. The settings shown in Table 6.1 allows setting the ratio of pre-trigger to post-trigger data saved in the sample buffer.

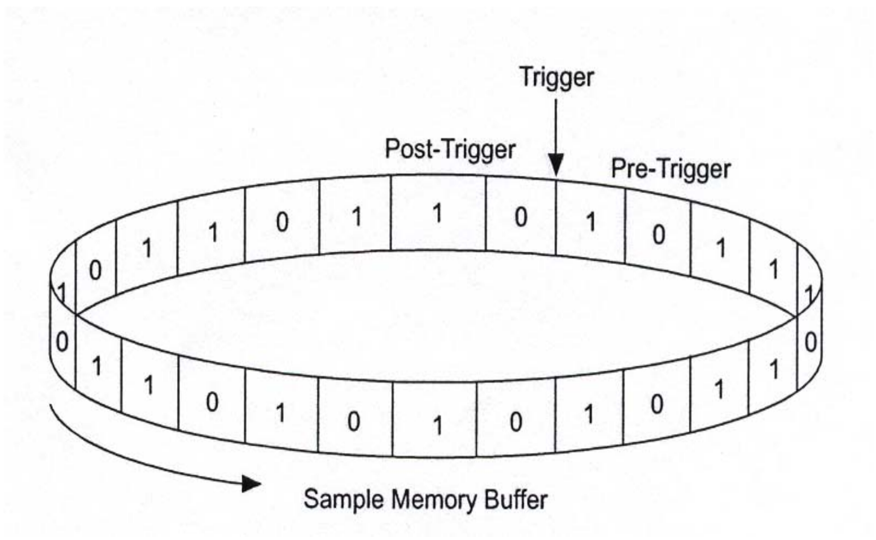


Figure 6.2 Circular Signal Capture Buffer (from [23])

Table 6.1 Trigger Position (from [24])

Name	**Description**
Pre-trigger	Captures signals immediately after triggering (12% pre-trigger, 88% post-trigger)
Center	Captures signals before and after triggering (50% pre-trigger, 50% post-trigger)
Post-trigger	Captures signals that occur immediately before triggering (88% pre-trigger, 12% post-trigger)
Continuous trigger	Captures signals indefinitely until stopped manually

2. Setting the Trigger Pattern

Signal pattern recognition is used for triggering by the SignalTap analyzer. Within the Quartus II software, the logic condition for each input signal to specify the trigger pattern is set. The SignalTap analyzer is triggered if the input signal matches the trigger pattern. Table 6.2 lists possible trigger patterns for each channel.

Table 6.2 Channel Trigger Patterns (from [23])

Trigger Pattern	Description
Don't Care	Default trigger condition. The channel is not used to determine the trigger event.
Low	The analyzer triggers when the channel is low.
High	The analyzer triggers when the channel is high.
Falling	The analyzer triggers when the channel is falling.
Rising	The analyzer triggers when the channel is rising.
Rising or Falling Edge	The analyzer triggers when the channel is rising or falling.

6.2 The Ethereal Network Analyzer

Ethereal [25] is a network protocol analyzer for use on Unix and Windows operating systems. It allows examining data from a live network or from a capture file on disk. With Ethereal, the capture data can be browsed interactively by viewing summary and detail information for each packet. Also, Ethereal has several features, including the ability to view the reconstructed stream of a TCP session.

6.3 Ping Command

"Ping" is one of the most useful network debugging tools. It takes its name from a submarine sonar search - if a short sound burst is sent and an echo is listened- a *ping* - coming back.

In an IP network, "ping" sends a short data burst (a single packet) and listens for a single packet in reply. The most basic function of an IP network (delivery of single packet) can be tested by a "ping", which is implemented using the required ICMP Echo function. The first two packets are broadcast packets, which are important for the following four ICMP packets going through.

6.4 Testing Methodology

In order to test the Silicon Firewall system easily, a small network that consists of three computers are made up: Mordor, Nios1 and Athlon2. The “Ethereal” was installed in the Mordor computer, and the “ping” command is issued from there. Nios1 is the name for the soft core Nios CPU in the Silicon Firewall system. The NDK and the NEDK are installed in Athlon2, also the testing result is displayed in the Quartus II software in this computer.

Basically, the results will be discussed in two sections, in section 6.4.1, the Silicon Firewall results will be demonstrated; the results of software firewall will be demonstrated in section 6.4.2, and some comparison will be given.

Before any detailed result is given, some configurations of SignalTap will be discussed. In this project, the “clock” for the SignalTap is the clock from the APEX board, which is “33.33333 MHz”. For the Silicon Firewall, the sample depth is “1K”samples due to the limitation of the ESBs included in the APEX EP20K200E device and the number of nodes. For the software firewall, the sample depth is “16K” because of fewer signals being traced. The “Pre” trigger position is used. In CAM initialization, the trigger pattern is the rising edge of the “wren” signal. For both the Silicon Firewall and software firewall, the rising edge of interrupt signal from “NEDK_L_IRQ” is used as the trigger pattern.

6.4.1 Silicon Firewall Results

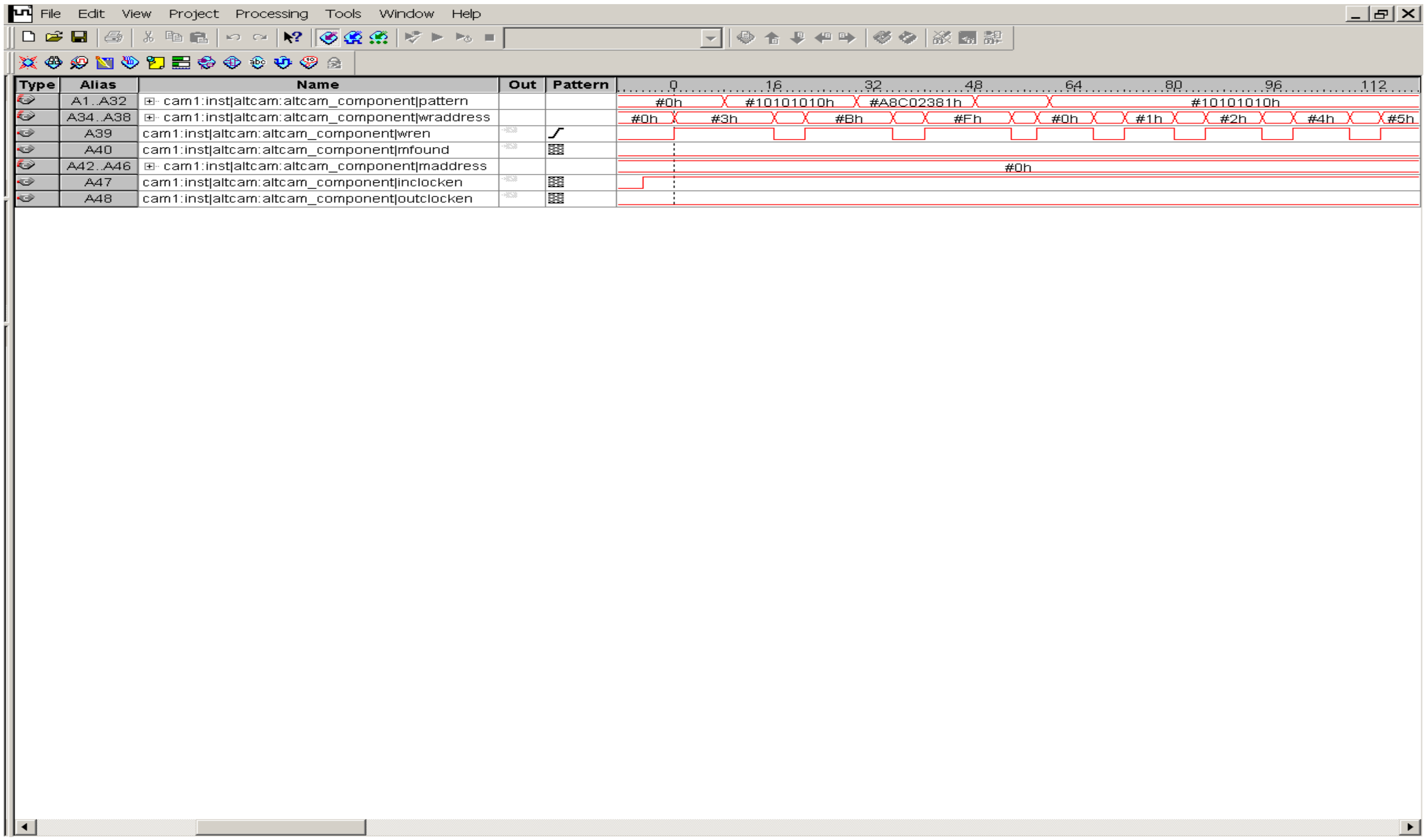
In this section, the results will be discussed in the following sequence. In section 6.4.1.1, the IP address of Mordor is stored in CAM when CAM is initialized, after that, a "ping" command is issued from Mordor. In section 6.4.1.2, the CAM is initialized without the IP address of Morder.

6.4.1.1 Results for an Authorized Packet

Figure 6.3 shows the waveform of CAM initialization, the IP address (192.168.128.216) of Mordor is written into CAM as a pattern, the value of which is "a8c02381" in hexadecimal.

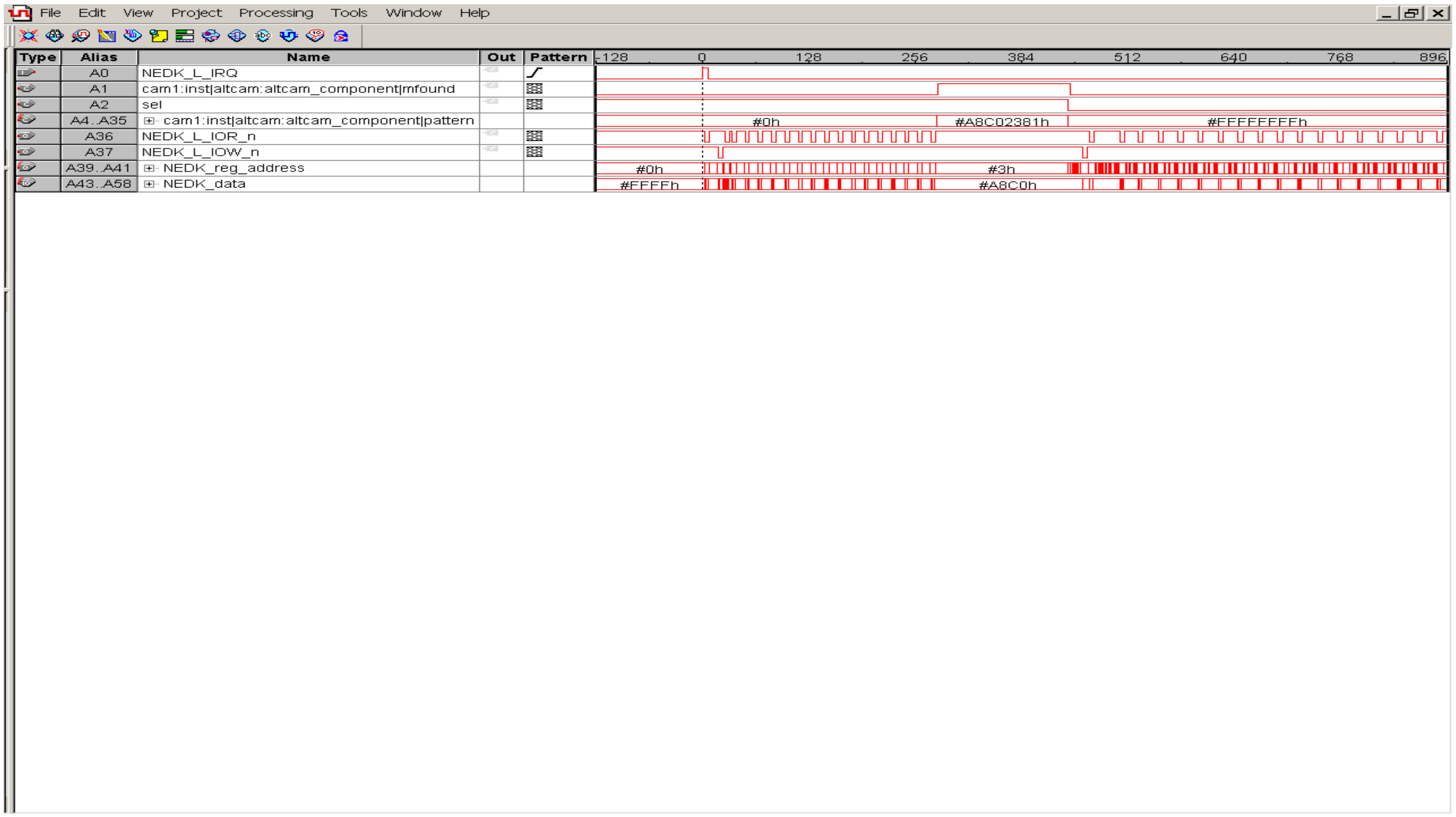
Figure 6.4 demonstrates the result when the "ping" command is issued from Mordor. First of all, an interrupt is generated from the CS8900A, the "NEDK_L_IRQ" goes high, and the first read of "ISQ" make this signal goes low. After that, the interrupt is re-enabled by reading the "RxLength" and part of the packet data. Then the IP address of Morder is reconstructed and fed into CAM. Since there is such a pattern stored in CAM, "mfound" goes high corresponding to a match. Before this point, the system is running in "Hw" mode. The "mfound" signal interrupts the Nios CPU to switch to "Sw" mode, and the Nios reads the complete packet data. From figure 6.4, the time between the interrupt that comes from the CS8900A IRQ line going high and "mfound" going high is 284 clock cycles.

Figure 6.5 shows the packet data captured by Ethereal. When the "ping" command is issued, four packets are sent out one after another. Figure 6.4 demonstrates what happens to one of them due to the buffer space limitation.



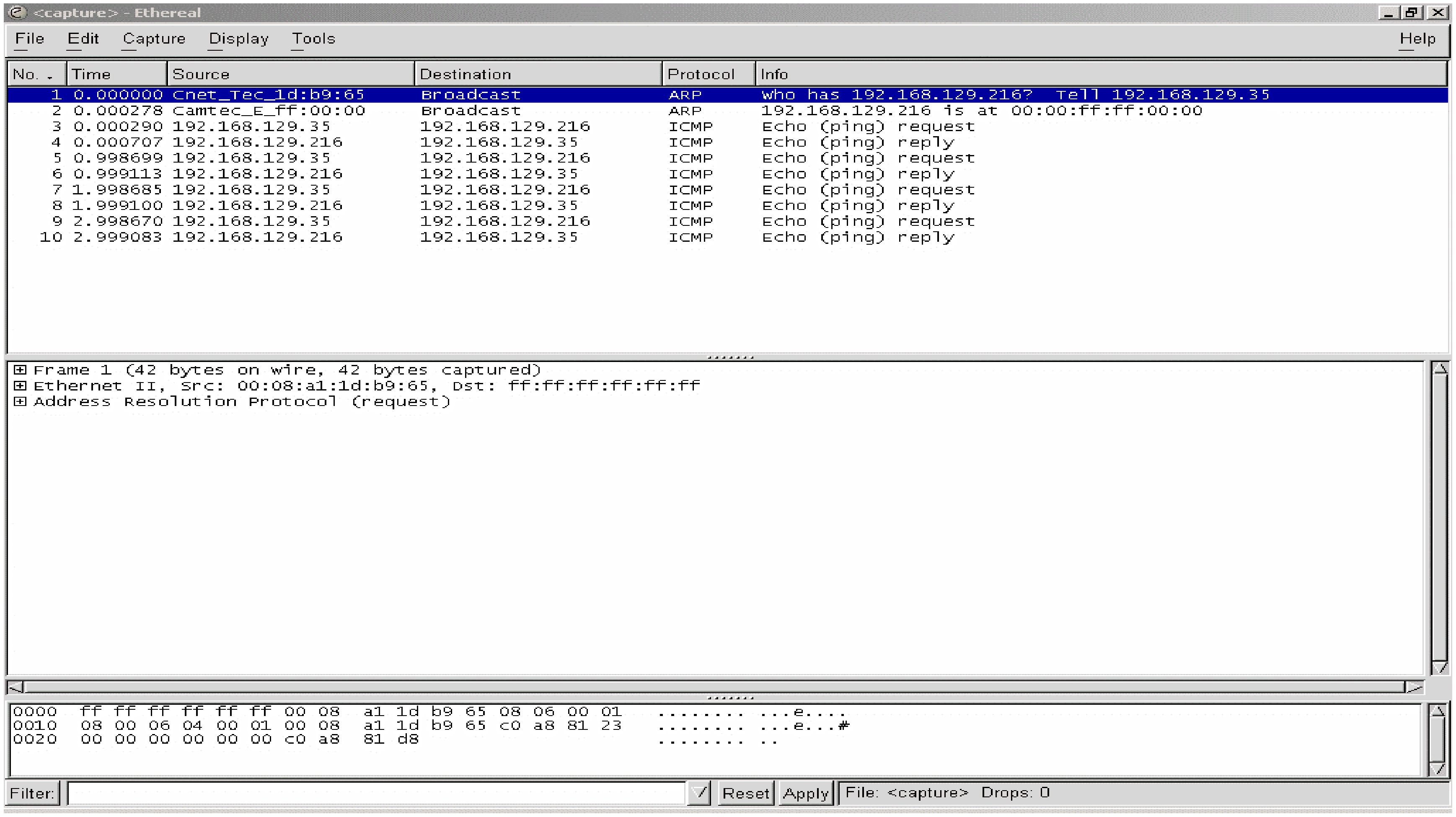
Type	Alias	Name	Out	Pattern
	A1..A32	⊞ cam1:inst|altcam:altcam_component|pattern		
	A34..A38	⊞ cam1:inst|altcam:altcam_component|wraddress		
	A39	cam1:inst|altcam:altcam_component|wren		
	A40	cam1:inst|altcam:altcam_component|mfound		▨
	A42..A46	⊞ cam1:inst|altcam:altcam_component|maddress		
	A47	cam1:inst|altcam:altcam_component|inclocken		▨
	A48	cam1:inst|altcam:altcam_component|outclocken		▨

Figure 6.3 CAM Initialization-1



Type	Alias	Name	Out	Pattern
	A0	NEDK_L_IRQ		
	A1	cam1:inst|altcam:altcam_component|mfound		
	A2	sel		
	A4..A35	⊞ cam1:inst|altcam:altcam_component|pattern		
	A36	NEDK_L_IOR_n		
	A37	NEDK_L_IOW_n		
	A39..A41	⊞ NEDK_reg_address		
	A43..A58	⊞ NEDK_data		

Figure 6.4 Results For an Authorized Packet In Hardware Mode



```
No.  Time     Source              Destination       Protocol Info
  1 0.000000 Cnet_Tec_1d:b9:65    Broadcast          ARP     Who has 192.168.129.216?  Tell 192.168.129.35
  2 0.000278 Camtec_E_ff:00:00    Broadcast          ARP     192.168.129.216 is at 00:00:ff:ff:00:00
  3 0.000290 192.168.129.35       192.168.129.216    ICMP    Echo (ping) request
  4 0.000707 192.168.129.216      192.168.129.35     ICMP    Echo (ping) reply
  5 0.998699 192.168.129.35       192.168.129.216    ICMP    Echo (ping) request
  6 0.999113 192.168.129.216      192.168.129.35     ICMP    Echo (ping) reply
  7 1.998685 192.168.129.35       192.168.129.216    ICMP    Echo (ping) request
  8 1.999100 192.168.129.216      192.168.129.35     ICMP    Echo (ping) reply
  9 2.998670 192.168.129.35       192.168.129.216    ICMP    Echo (ping) request
 10 2.999083 192.168.129.216      192.168.129.35     ICMP    Echo (ping) reply

⊞ Frame 1 (42 bytes on wire, 42 bytes captured)
⊞ Ethernet II, Src: 00:08:a1:1d:b9:65, Dst: ff:ff:ff:ff:ff:ff
⊞ Address Resolution Protocol (request)

0000  ff ff ff ff ff ff 00 08  a1 1d b9 65 08 06 00 01   ........ ...e....
0010  08 00 06 04 00 01 00 08  a1 1d b9 65 c0 a8 81 23   ........ ...e...#
0020  00 00 00 00 00 00 c0 a8  81 d8                     ........ ..

File: <capture>  Drops: 0
```

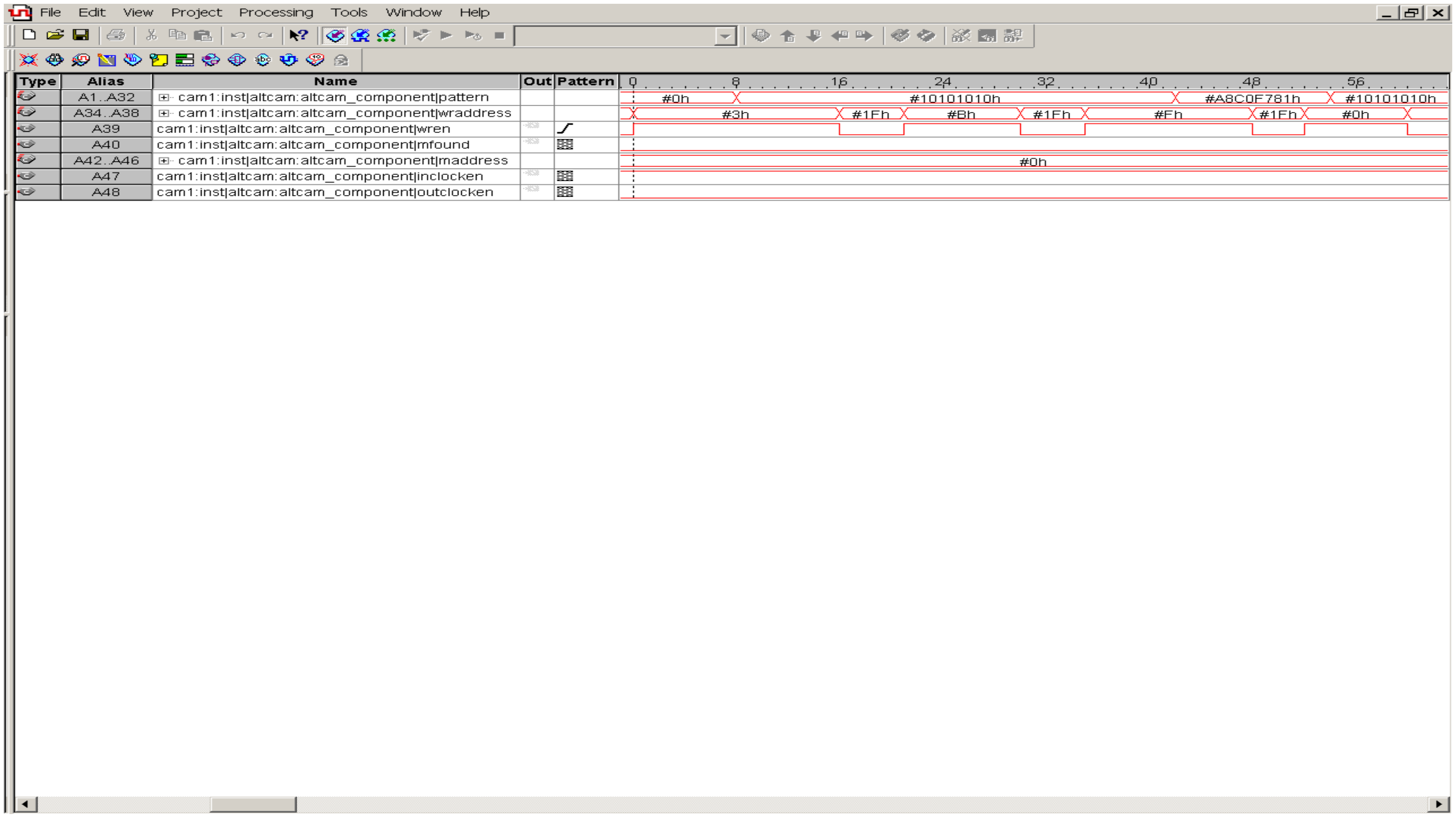
Figure 6.5 Data Captured By Ethereal For Authorized Packets

6.4.1.2 Results for Un-authorized Packet

Figure 6.6 shows the CAM initialization without the pattern for Mordor’s IP address. In this case, when the “ping” command is issued from Mordor, the interrupt still goes high, however, “mfound” never goes high since there is not a match any more (figure 6.7 demonstrates this), and the complete packet data will not be read by the Nios CPU. The “ping” command get timed out since there is no reply (figure 6.8 shows the data captured by Ethereal).

6.4.2 Software vs. Hardware Implementations

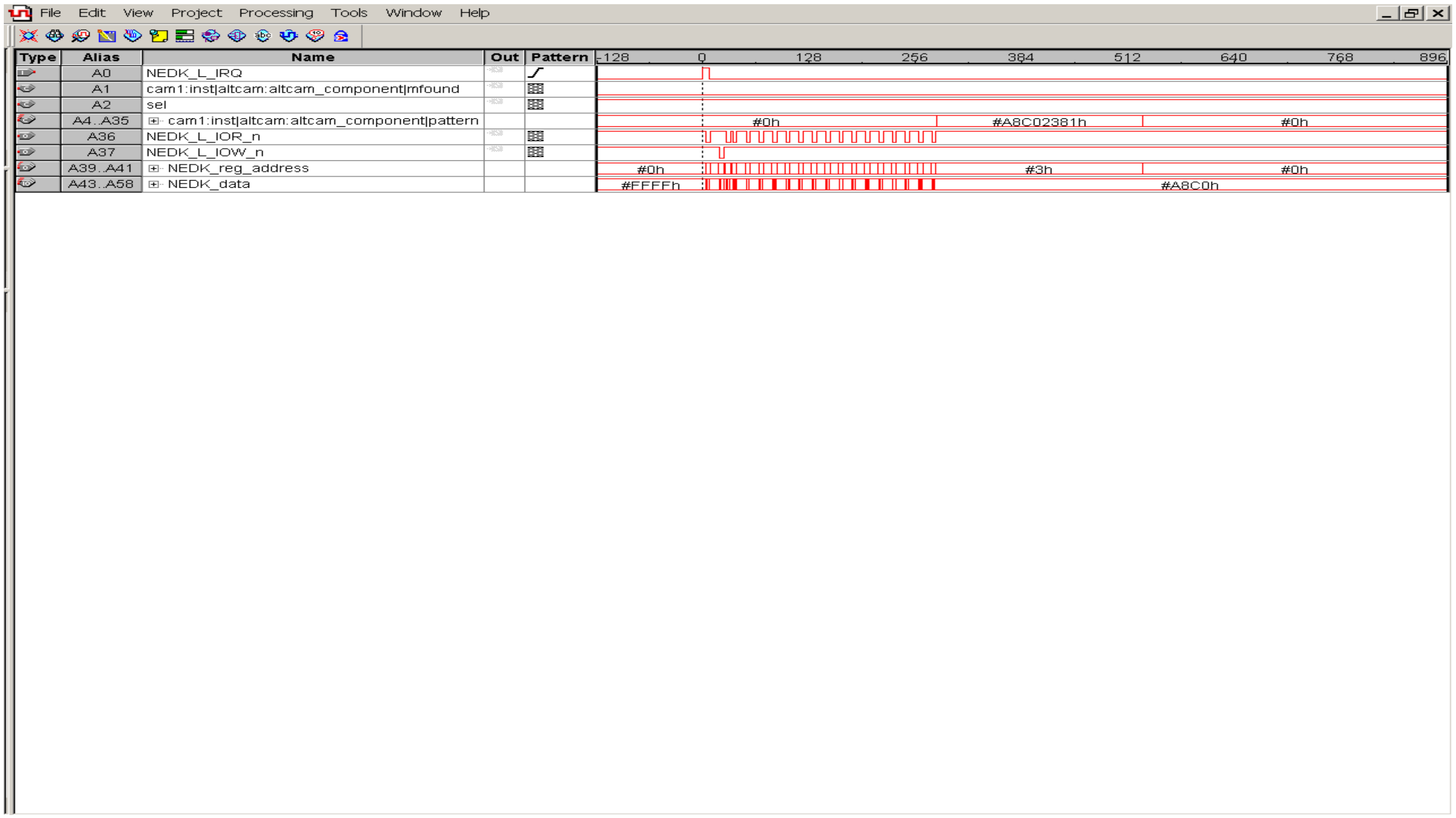
For the purpose of comparison, a software firewall is created using the same structure as the Silicon Firewall. For the software firewall, the CAM is initialized with the pattern of Athlon2’s IP address. Since the software firewall does table lookup sequentially, there are best case and worst case. For the best case, there is a match at the first address of the lookup table, the time interval is 9359 clock cycles (see figure 6.9); for the worst case, there is a match at the end of the lookup table, the time interval is 9598 clock cycles (see figure 6.10). Therefore, the Silicon Firewall is much faster than the software firewall (284 clock cycles vs. 9359 clock cycles).



Type	Alias	Name	Out	Pattern
	A1..A32	cam1:inst|altcam:altcam_component|pattern		
	A34..A38	cam1:inst|altcam:altcam_component|wraddress		
	A39	cam1:inst|altcam:altcam_component|wren		
	A40	cam1:inst|altcam:altcam_component|mfound		
	A42..A46	cam1:inst|altcam:altcam_component|maddress		
	A47	cam1:inst|altcam:altcam_component|inclocken		
	A48	cam1:inst|altcam:altcam_component|outclocken		

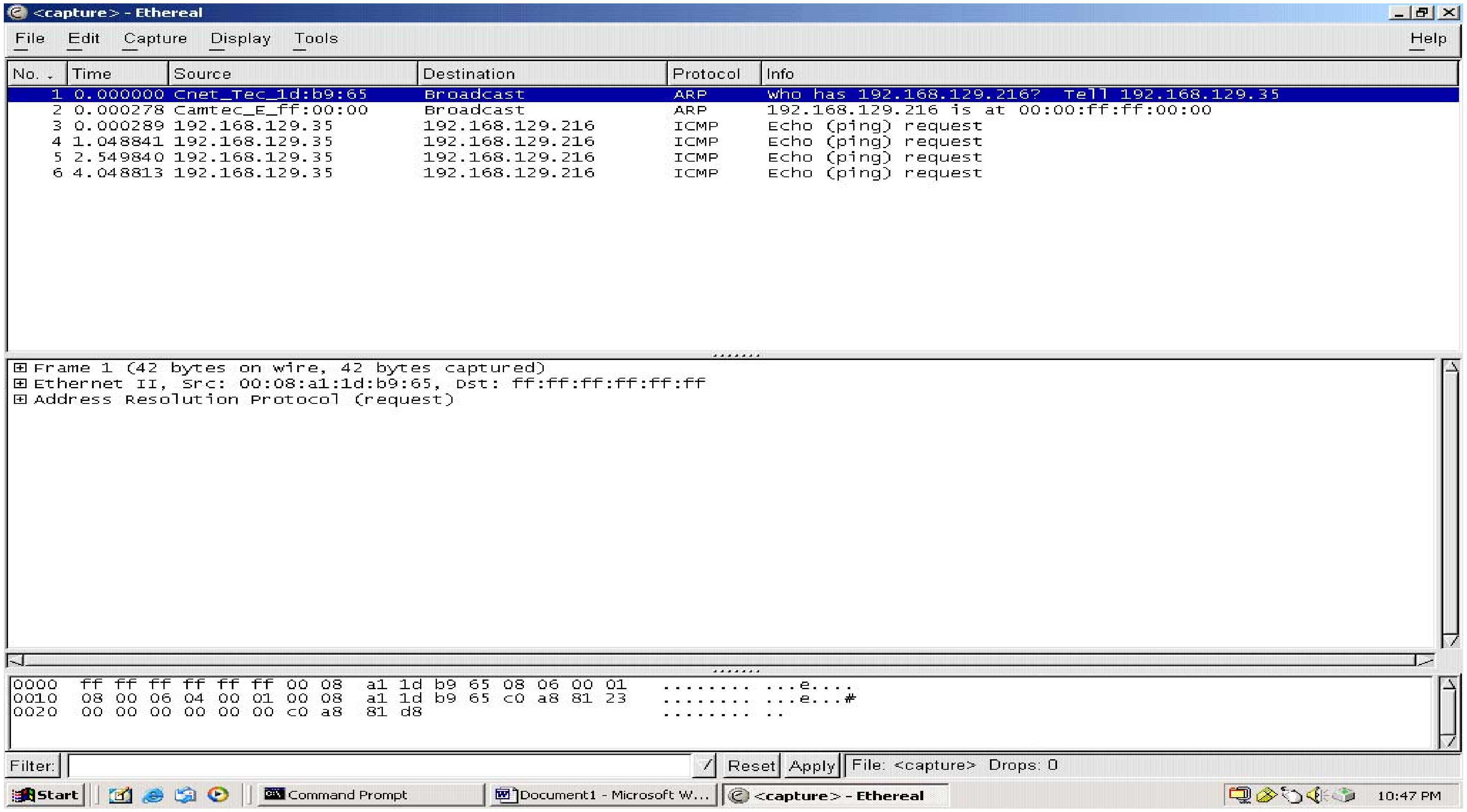
```
pattern:    #0h  #10101010h  #A8C0F781h  #10101010h
wraddress:  #3h  #1Fh  #Bh  #1Fh  #Fh  #1Fh  #0h
maddress:   #0h
```

Figure 6.6 CAM Initialization-2



Type	Alias	Name	Out	Pattern	
	A0	NEDK_L_IRQ			
	A1	cam1:inst|altcam:altcam_component|mfound			
	A2	sel			
	A4..A35	⊞ cam1:inst|altcam:altcam_component|pattern			#0h #A8C02381h #0h
	A36	NEDK_L_IOR_n			
	A37	NEDK_L_IOW_n			
	A39..A41	⊞ NEDK_reg_address			#0h #3h #0h
	A43..A58	⊞ NEDK_data			#FFFFh #A8C0h

Figure 6.7 Results For an Un-authorized Packet


<capture> - Ethereal
File Edit Capture Display Tools Help
Filter: Reset Apply
Start Command Prompt Document1 - Microsoft W... <capture> - Ethereal 10:47 PM

```
No.  Time      Source                Destination      Protocol  Info
  1 0.000000 Cnet_Tec_1d:b9:65     Broadcast        ARP       Who has 192.168.129.216?  Tell 192.168.129.35
  2 0.000278 Camtec_E_ff:00:00     Broadcast        ARP       192.168.129.216 is at 00:00:ff:ff:00:00
  3 0.000289 192.168.129.35        192.168.129.216  ICMP      Echo (ping) request
  4 1.048841 192.168.129.35        192.168.129.216  ICMP      Echo (ping) request
  5 2.549840 192.168.129.35        192.168.129.216  ICMP      Echo (ping) request
  6 4.048813 192.168.129.35        192.168.129.216  ICMP      Echo (ping) request

⊞ Frame 1 (42 bytes on wire, 42 bytes captured)
⊞ Ethernet II, Src: 00:08:a1:1d:b9:65, Dst: ff:ff:ff:ff:ff:ff
⊞ Address Resolution Protocol (request)

0000  ff ff ff ff ff ff 00 08  a1 1d b9 65 08 06 00 01   ........ ...e....
0010  08 00 06 04 00 01 00 08  a1 1d b9 65 c0 a8 81 23   ........ ...e...#
0020  00 00 00 00 00 00 c0 a8  81 d8                     ........ ..

File: <capture>  Drops: 0
```

Figure 6.8 Data Captured By Ethereal For Un-authorized Packets

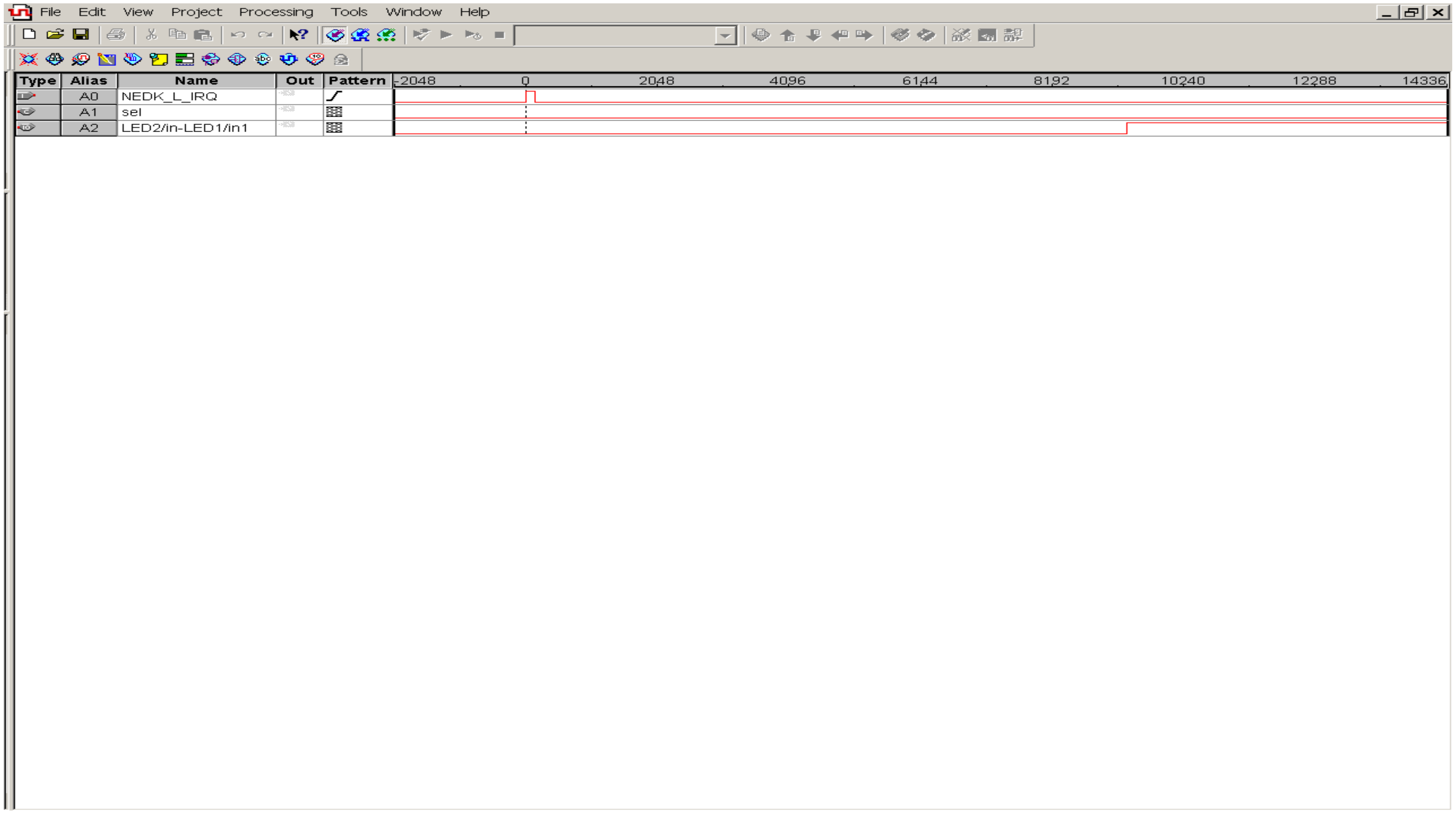


Figure 6.9 Results of Software Firewall For an Authorized Packet (Best Case)

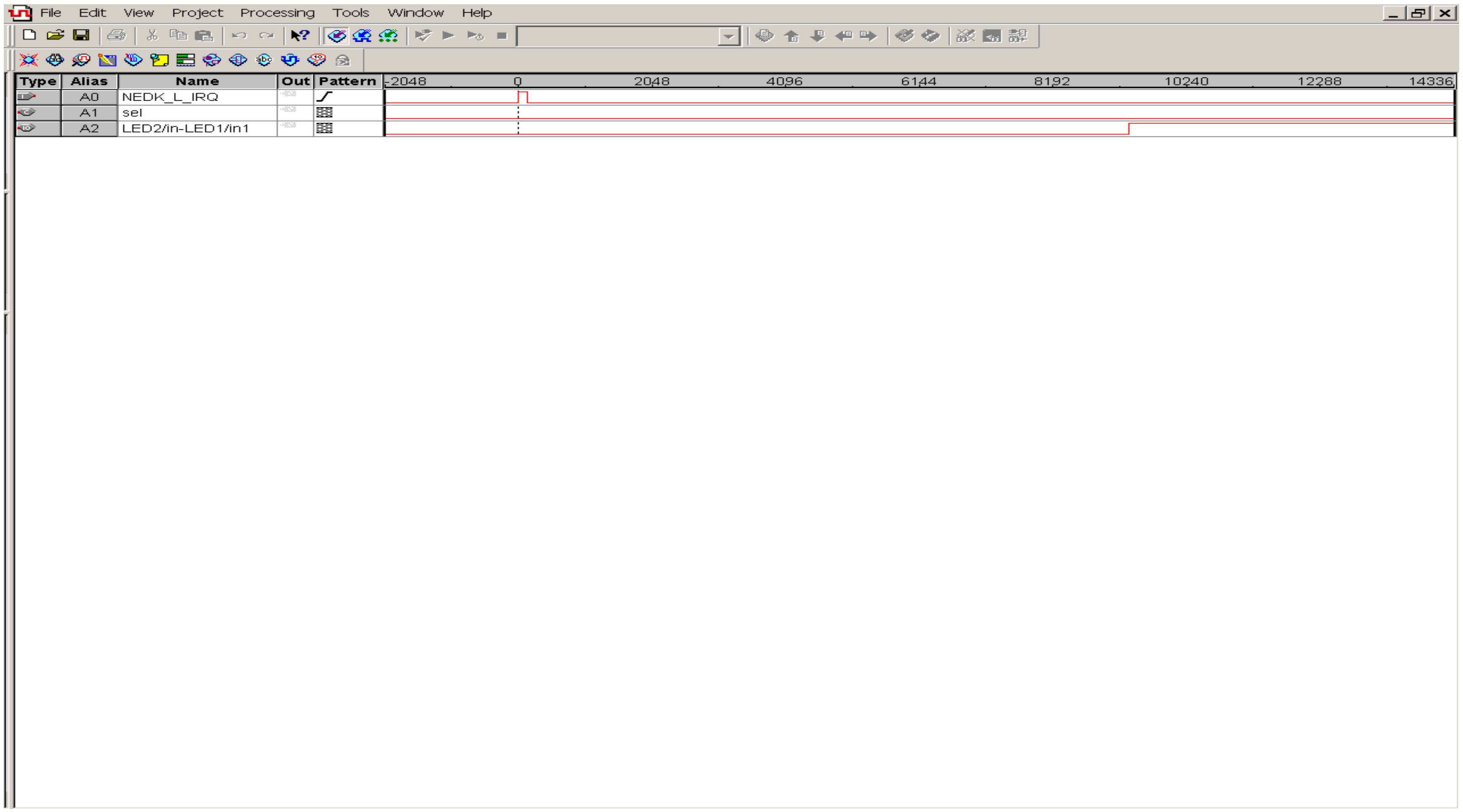


Figure 6.10 Results of Software Firewall For an Authorized Packet (Worst Case)

Chapter 7 SUMMARY, CONCLUSION AND FUTURE WORK

This thesis addresses issues related to a Silicon Firewall design and implementation in an Altera FPGA. The performance of this hardware firewall is tested using a real network, and compared with a software firewall design with silmilar architecture. This chapter summarizes the work that was done and presents future areas of research.

7.1 Summary

The Internet security problem was presented, and different security models were discussed. The firewall was identified as an effective type of network security. The possible inefficiency of the traditional software firewall technology was introduced, and the hardware firewall feasibility was reviewed. The research objective, an investigation of if and how existing software firewall technology could be improved by replacing software functionality with hardware (silicon) was then presented.

The Internet security system was reviewed. Different security strategies were discussed, and the TCP/IP fundamentals were introduced as background. Packet filtering was also presented.

Since the Silicon Firewall system is an embedded real-time system, and the design is a hardware-software codesign, some background on embedded system and hardware-software codesign was discussed. The Silicon Firewall system design was then presented in detail. SOPC design and embedded processor PLD solutions were introduced as the background needed to understand the Silicon Firewall system design. The Altera Excalibur embedded PLD solutions were presented, and based on a discussion of soft core and hard core implementations, the rationale for using the Nios soft core embedded processor in this research project was justified.

The details of the Nios embedded processor and two development kits (NDK and NEDK) were introduced. The Nios development board and the Ethernet daughter card as well as the two libraries (Nios peripheral library and Ethernet plugs library) were introduced.

The two most important components of the Silicon Firewall, the CS8900A Ethernet controller and the CAM implementation were reviewed. The Silicon Firewall design was described in detail, following the discussion of the CS8900A and CAM.

The SignalTap embedded logic analyer and the Ethernet network analyzer were introduced as the two tools used to capture data in the Silicon Firewall system testing. Background of the “Ping” command was also given. The testing methodology used was introduced and the Silicon Firewall system test results were presented.

7.2 Conclusions

The objective of this research was to “investigate if and how existing desktop computer software firewall technology could be improved by replacing software functionality with hardware (silicon).” The results show that this research objective has been successfully achieved:

1. Test results confirm that the Silicon Firewall system functions as an Internet firewall.
2. Comparison of the hardware firewall and software firewall show that the Silicon Firewall system is much faster than the traditional software firewall (284 clock sysles vs. 9359 clock cycles).

It should be noted that since the research was started 3Com Corporation has announced [26] development of an embedded firewall with similar capabilities to the Silicon Firewall. It was not yet appeared as a consumer product at this time.

7.3 Future Work

While the project has been successful in reaching the goal of design of a Silicon Firewall, further improvements must be done for a viable system in the future:

- Other CAMs

  Since the Silicon Firewall system in this research project does packet filtering only by the source IP address, only one CAM was included. However, the final Silicon Firewall may do other kinds of packet filtering, for example, by service, source port and so forth. Other CAMs may need to be added to the Silicon Firewall system. This may necessitate the need for external CAM.

- Dynamic CAM

  This is another issue related to CAM in the future. The CAM is pre-loaded with a number of patterns in advance for now, but dynamically changing the patterns in the CAM is desirable.

- Logging of IP addresses

  For the current Silicon Firewall system, an unauthorized packet will disappear after being blocked. In the future, logging of source IP address can be done for the purpose of determining who is attempting the attack.

- Remote Administration

  Currently, the Silicon Firewall only can be administrated in the local computer. A remote administration feature may be desirable in the future. This will allow system administrators to update individual user machines from a central site.

- PC Integration

  The final purpose is integrating the Silicon Firewall system with a PC. This would include PC bus implementation and appropriate drivers for the Silicon Firewall system.

Bibliography

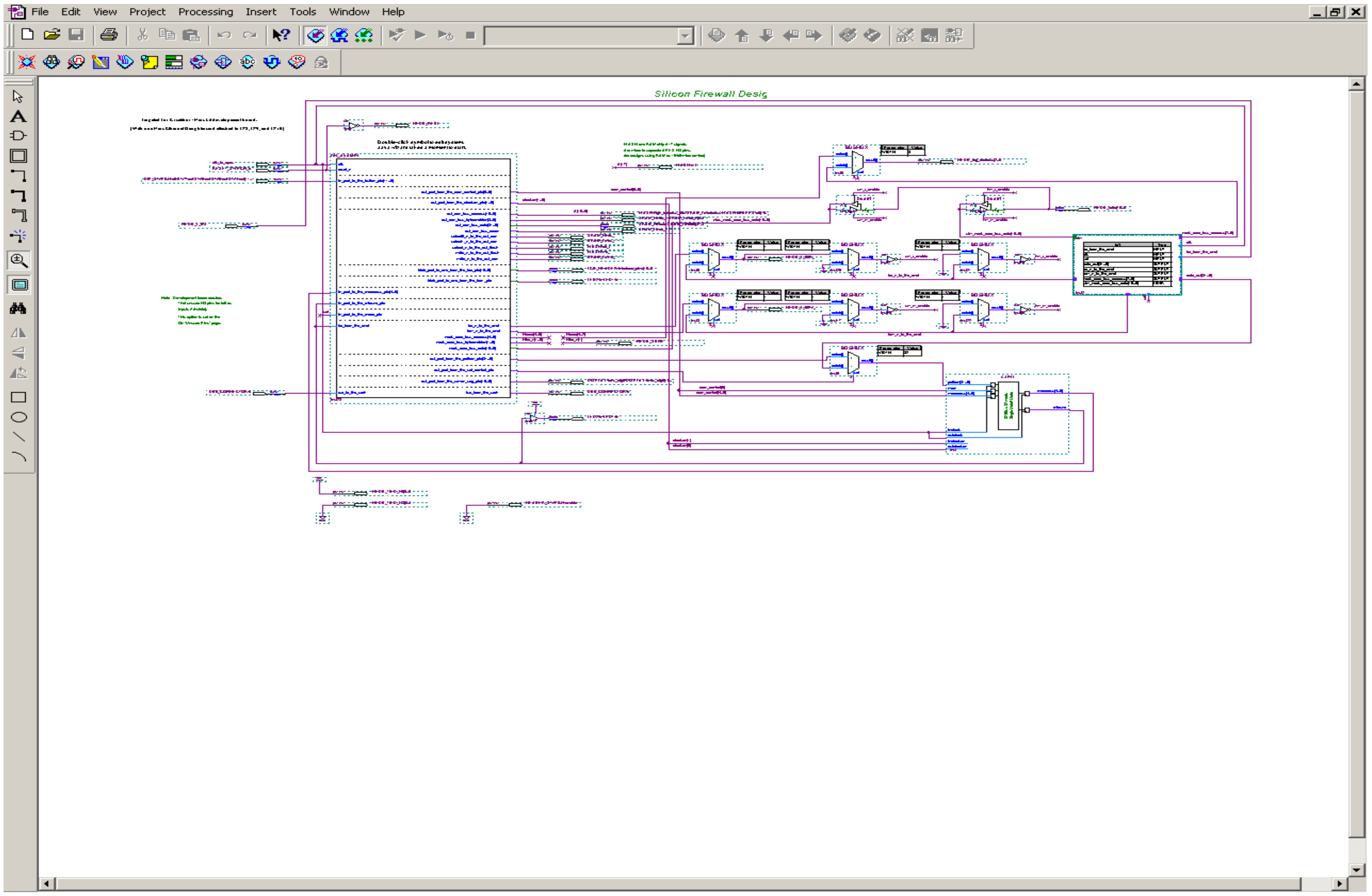
(See CD for World Wide Web HTM files)

1. D. B. Chapman and Elizabeth D. Zwicky, *Building Internet Firewalls*, O'Reilly & Associates, Inc., 1995.
2. *OSI Seven Layer Model*, at http://www.freesoft.org/CIE/Topics/15.htm
3. *TCP/IP Protocol Architecture*, at http://www.soldierx.com/books/networking/tcpip/ch01_03.htm
4. *Network Access Layer*, at http://www.soldierx.com/books/networking/tcpip/ch01_04.htm
5. *Internet Layer, at* http://www.soldierx.com/books/networking/tcpip/ch01_05.htm
6. *Ping*, at http://www.freesoft.org/CIE/Topics/53.htm
7. *Transport Layer*, at http://www.soldierx.com/books/networking/tcpip/ch01_06.htm
8. *Application Layer, at* http://www.soldierx.com/books/networking/tcpip/ch01_07.htm
9. D. D. Gajski and F. Vahid, "*Specification and Design of Embedded Hardware-software Systems*", IEEE Design and Test of Computers, Vol. 12, No. 1. Spring 1995, pp. 53-67.
10. R. Ernst, "*Codesign of Embedded Systems: Status and Trends*", IEEE Design and Test of Computers, Vol. 15, No. 2. April-June 1998, pp. 45-54.
11. C. Kuttner, TRW Space and Electronics Group, "*Hardware-Software Codesign Using Processor Synthesis*", IEEE Design and Test of Computers, Vol. 13, No. 3. Fall 1996, pp. 43-53.
12. D. E. Thomas, J. K. Adams and H. Schmit, "*A Model and Methodology for Hardware-Software Codesign*", IEEE Design and Test of Computers, Vol. 10, No. 3. July-September 1993, pp. 6-15.
13. Altera Corporation. *Excalibur Backgrounder White Paper*.
14. Altera Corporation. *Nios Embedded Processor Development Board Data Sheet*, Ver. 2.1, April 2002.
15. Altera Corporation. *Excalibur Development Kit with the Nios Embedded Processor Data Sheet*, Ver. 1.0, June 2000.
16. *Nios Peripheral Library*, at http://www.altera.com/products/devices/nios/features/nio-peripherals.html
17. Altera Corporation. *Nios Ethernet Development Kit User Guide*, Ver. 1.0, July 2001.
18. Cirrus Logic, *CS8900A Product Data Sheet*.

19. Cirrus Logic, Application Note 83: *Crystal LAN™ CS8900A Ethernet Controller Technical Reference Manual*.
20. Altera Corporation. Application Note 119: *Implementing High-Speed Search Applications with Altera CAM*, Ver. 2.1, July 2001.
21. Altera Corporation. *Using APEX 20KE CAM for Fast Search Applications*, Ver. 1.0, August 1999.
22. SiberCore Technologies. Cascading The SiberCAM Ulter-2M SCT2000B Application Note SCAN204, 3-00034-006.1-November 2000.
23. Altera Corporation. *SignalTap Embedded Logic Analyzer Megafunction Data*, Ver. 2.0, April 2001.
24. Altera Corporation. *SignalTap Embedded Logic Analyzer Megafunction Data*, Ver. 2.1, September 2002.
25. *The Ethereal Network Analyzer*, at http://www.ethereal.com/
26. 3Com. Corporation, *3Com® Embedded Firewall Architecture for E-business*: Stronger Security for Open Networks, Technical Brief, 100969-001, 04/01.
27. Altera Corporation. *SOPC Builder User Guide*, Ver. 1.0, June 2003.

APPENDIX A BLOCK DIAGRAM OF SILICON FIREWALL DESIGN

(See CD)


File
Edit
View
Project
Processing
Insert
Tools
Window
Help
Silicon Firewall Desig

APPENDIX B CAM INITIALIZATION

```
void cam_initialize(void)

//define three CAM initialization-related pios
np_pio *pattern_in=na_pattern_pio;
np_pio *cam_control=na_cam_control_pio;
np_pio *clocken=na_clocken_pio;

//define “input” or “output” of each pio
clocken->np_piodirection=1;          //set "clocken" as output
pattern_in->np_piodirection=1;       //set "pattern_in" as output
cam_control->np_piodirection=1;    //set "cam_control" as output
cam_control->np_piodata=0x00;     //initialize the cam_control pio data to be "0"
clocken->np_piodata=2;               //enable inclocken and disable outclocken

//Initialize CAM with 32 patterns, two of them are IP address of Mordor and Athlon2,
//the rest patterns can be any 32-bit numbers, in this file, all of them are “0x10101010”.
//For each pattern, set up wren and wraddress first, and then pattern. The 32 addresses
//can be written in any sequence.

cam_control->np_piodata=0x23;    //set "wen=1","waddress=3"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;         //set “wen=0” after each write
cam_control->np_piodata=0x2b;          //set "wen=1","waddress=b"
pattern_in->np_piodata=0xa8c02381;    //ip address of Mordor

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x2f;         //set "wen=1","waddress=f"
pattern_in->np_piodata=0x10101010;


cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x20;         //set "wen=1","waddress=0"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x21;         //set "wen=1","waddress=1"
pattern_in->np_piodata=0x10101010;
```

```
cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x22;        //set "wen=1","waddress=2"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x24;       //set "wen=1","waddress=4"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x25;      //set "wen=1","waddress=5"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x26;      //set "wen=1","waddress=6"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x27;      //set "wen=1","waddress=7"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x28;     //set "wen=1","waddress=8"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x29;    //set "wen=1","waddress=9"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x2a;   //set "wen=1","waddress=a"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x2c;   //set "wen=1","waddress=c"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x2d;  //set "wen=1","waddress=d"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x2e;  //set "wen=1","waddress=e"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x30;       //set "wen=1","waddress=16"
```

```
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x31;      //set "wen=1","waddress=17"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x32;      //set "wen=1","waddress=18"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x33;      //set "wen=1","waddress=19"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x34;      //set "wen=1","waddress=20"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x35;     //set "wen=1","waddress=21"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x36;    //set "wen=1","waddress=22"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x37;    //set "wen=1","waddress=23"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x38;   //set "wen=1","waddress=24"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x39;   //set "wen=1","waddress=25"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x3a;  //set "wen=1","waddress=26"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x3b;  //set "wen=1","waddress=27"
pattern_in->np_piodata=0x10101010;
```

```
cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x3c;   //set "wen=1","waddress=28"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x3d;   //set "wen=1","waddress=29"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x3e;   //set "wen=1","waddress=30"
pattern_in->np_piodata=0xa8c0d981;   // ip address of Athlon2

cam_control->np_piodata&=0xdf;
cam_control->np_piodata=0x3f;   //set "wen=1","waddress=31"
pattern_in->np_piodata=0x10101010;

cam_control->np_piodata&=0xdf;
clocken->np_piodata=3;   //enable both input and output clock

}
```

APPENDIX C IP SOURCE ADDRESS RECONSTRUCTION MODULE

```
//This Verilog module functions according to the interrupt from the CS8900A, for each
//interrupt, the ISQ register is read first to make the interrupt go low, then the Rx_length
//and part of the frame date are read to re-enable the interrupt. The first 14 words of
//frame data are skipped, the 15th word and the 16th word are two parts of the source IP
//address, they are reconstructed and form the 32 bits IP address.

module sfw( clk, irq_from_the_enet, nedk_card_bus_address, ior_n_to_the_enet,
            iow_n_to_the_enet, sfw_nedk_card_bus_data, data_out, sel);

input clk, irq_from_the_enet, sel;
output [2:0] nedk_card_bus_address;
output ior_n_to_the_enet;
output iow_n_to_the_enet;
output [31:0] data_out;
inout [15:0] sfw_nedk_card_bus_data;
reg IRQ;
reg ior_n_to_the_enet, iow_n_to_the_enet;
reg [2:0] nedk_card_bus_address;
reg [3:0]timing_cycle;  // "timing_cycle" is the system clock cycle, which is counted to
                        // meet the switching characteristics of the CS8900A
reg [5:0]command_cycle;  //every read of the register or the frame data is called a
                         //"command_cycle"
reg [15:0] data_from_cs8900;
reg [15:0] data_to_cs8900;
reg [15:0] data1;
reg [15:0] data2;
reg [31:0] data_out;

parameter Rx_Tx_0=3'b000;
parameter Rx_Tx_1=3'b001;
parameter TxCMD=3'b010;
parameter Txlength=3'b011;
parameter ISQ=3'b100;
parameter PPP=3'b101;
parameter PPD_0=3'b110;parameter PPD_1=3'b111;
parameter Rx_status=16'b0000_0100_0000_0000;
parameter Rx_length=16'b0000_0100_0000_0010;
parameter Rx_config=16'b0000_0001_0000_0010;
parameter Rx_event=16'b0000_0001_0010_0100;

always @ (posedge clk)

begin
        if ((irq_from_the_enet==1) && (IRQ==0))
            begin
                    command_cycle<=0;
                    timing_cycle<=0;
                    IRQ<=1;
            end

        else
             begin
                     case(command_cycle)
                           6'b000000:
                               begin  //read ISQ
                                        timing_cycle<=timing_cycle+1;
                                        case(timing_cycle)
                                              4'b0001: nedk_card_bus_address<=ISQ;
                                              4'b0010: ior_n_to_the_enet<=0;
                                              4'b0111: begin
                                                                   ior_n_to_the_enet<=1;
                                                                   nedk_card_bus_address<=3'b011;
                                                           end
                                              4'b1111: begin
                                                                   command_cycle<=command_cycle+1;
                                                                   timing_cycle<=0;
                                                           end
                                        endcase
                               end

                           6'b000001:
                               begin  //read Rx_length
                                        timing_cycle<=timing_cycle+1;
                                        case(timing_cycle)
                                              4'b0001:nedk_card_bus_address<=PPP; //set up
                                                            //packetpage pointer first
                                              4'b0010: iow_n_to_the_enet<=0;
                                              4'b0011: data_to_cs8900<=Rx_length;
                                              4'b0111: begin
                                                                   iow_n_to_the_enet<=1;
                                                                   nedk_card_bus_address<=3'b011;
                                                                   data_to_cs8900<=0;

                                                            end
                    4'b1001:nedk_card_bus_address<=PPD_0;  // read
                                 //data  from PPD_0
                    4'b1010: ior_n_to_the_enet<=0;
                    4'b1111: begin
                                ior_n_to_the_enet<=1;
                                nedk_card_bus_address<=3'b011;
                                data_to_cs8900<=0;
                                command_cycle<=command_cycle+1;
                                timing_cycle<=0;
                             end
                endcase
        end

      6'b000010:  //the frame data of the following 14 reads are skipped
        begin //read data from Rx_Tx_0
                timing_cycle<=timing_cycle+1;
                case(timing_cycle)
                    4'b0001: nedk_card_bus_address<=Rx_Tx_0;
                    4'b0010: ior_n_to_the_enet<=0;
                    4'b0111: begin
                                ior_n_to_the_enet<=1;
                                nedk_card_bus_address<=3'b011;
                             end
                    4'b1111: begin
                                command_cycle<=command_cycle+1;
                                 timing_cycle<=0;
                             end
                endcase
        end

      6'b000011:
        begin  //read data from Rx_Tx_0
                timing_cycle<=timing_cycle+1;
                case(timing_cycle)
                    4'b0001: nedk_card_bus_address<=Rx_Tx_0;
                    4'b0010: ior_n_to_the_enet<=0;
                    4'b0111: begin
                                 ior_n_to_the_enet<=1;
                                 nedk_card_bus_address<=3'b011;
                             end
                    4'b1111: begin
                                 command_cycle<=command_cycle+1;
                                timing_cycle<=0;
                            end
                endcase
        end
```

```
6'b000100:
    begin  //read data from Rx_Tx_0
           timing_cycle<=timing_cycle+1;
           case(timing_cycle)
                 4'b0001: nedk_card_bus_address<=Rx_Tx_0;
                 4'b0010: ior_n_to_the_enet<=0;
                 4'b0111: begin
                                   ior_n_to_the_enet<=1;
                                   nedk_card_bus_address<=3'b011;
                            end
                 4'b1111:begin
                                   command_cycle<=command_cycle+1;
                                   timing_cycle<=0;
                            end
           endcase
    end

6'b000101:
    begin  //read data from Rx_Tx_0
           timing_cycle<=timing_cycle+1;
           case(timing_cycle)
                 4'b0001: nedk_card_bus_address<=Rx_Tx_0;
                 4'b0010: ior_n_to_the_enet<=0;
                 4'b0111: begin
                                   ior_n_to_the_enet<=1;
                                   nedk_card_bus_address<=3'b011;
                            end

                 4'b1111: begin
                                   command_cycle<=command_cycle+1;
                                   timing_cycle<=0;
                           end
            endcase
    end

6'b000110:
    begin  //read data from Rx_Tx_0
           timing_cycle<=timing_cycle+1;
           case(timing_cycle)
                 4'b0001: nedk_card_bus_address<=Rx_Tx_0;
                 4'b0010: ior_n_to_the_enet<=0;
                 4'b0111: begin
                                   ior_n_to_the_enet<=1;
                                   nedk_card_bus_address<=3'b011;
                            end
```

```
                  4'b1111: begin
                                  command_cycle<=command_cycle+1;
                                  timing_cycle<=0;
                              end
              endcase
         end

     6'b000111:
         begin  //read data from Rx_Tx_0
              timing_cycle<=timing_cycle+1;
              case(timing_cycle)
                  4'b0001: nedk_card_bus_address<=Rx_Tx_0;
                  4'b0010: ior_n_to_the_enet<=0;
                  4'b0111: begin
                                  ior_n_to_the_enet<=1;
                                  nedk_card_bus_address<=3'b011;
                              end
                  4'b1111: begin
                                  command_cycle<=command_cycle+1;
                                  timing_cycle<=0;
                              end
              endcase
         end

     6'b001000:
         begin  //read data from Rx_Tx_0
              timing_cycle<=timing_cycle+1;
              case(timing_cycle)
                  4'b0001: nedk_card_bus_address<=Rx_Tx_0;
                  4'b0010: ior_n_to_the_enet<=0;
                  4'b0111: begin
                                  ior_n_to_the_enet<=1;
                                  nedk_card_bus_address<=3'b011;
                              end
                  4'b1111: begin
                                  command_cycle<=command_cycle+1;
                                  timing_cycle<=0;
                              end
              endcase
         end

     6'b001001:
         begin  //read data from Rx_Tx_0
              timing_cycle<=timing_cycle+1;
              case(timing_cycle)
                  4'b0001: nedk_card_bus_address<=Rx_Tx_0;
```

```
                4'b0010: ior_n_to_the_enet<=0;
                4'b0111: begin
                                ior_n_to_the_enet<=1;
                                nedk_card_bus_address<=3'b011;
                         end
                4'b1111: begin
                                command_cycle<=command_cycle+1;
                                timing_cycle<=0;
                         end
            endcase
    end

6'b001010:
    begin  //read data from Rx_Tx_0
            timing_cycle<=timing_cycle+1;
            case(timing_cycle)
                4'b0001: nedk_card_bus_address<=Rx_Tx_0;
                4'b0010: ior_n_to_the_enet<=0;
                4'b0111: begin
                                ior_n_to_the_enet<=1;
                                nedk_card_bus_address<=3'b011;
                         end
                 4'b1111: begin
                                command_cycle<=command_cycle+1;
                                timing_cycle<=0;
                          end
            endcase
    end

6'b001011:
    begin  //read data from Rx_Tx_0
            timing_cycle<=timing_cycle+1;
            case(timing_cycle)
                4'b0001: nedk_card_bus_address<=Rx_Tx_0;
                4'b0010: ior_n_to_the_enet<=0;
                4'b0111: begin
                                ior_n_to_the_enet<=1;
                                nedk_card_bus_address<=3'b011;
                        end
                4'b1111: begin
                                command_cycle<=command_cycle+1;
                                timing_cycle<=0;
                        end
            endcase
    end
```

```
6'b001100:
    begin  //read data from Rx_Tx_0
            timing_cycle<=timing_cycle+1;
            case(timing_cycle)
                  4'b0001: nedk_card_bus_address<=Rx_Tx_0;
                  4'b0010: ior_n_to_the_enet<=0;
                  4'b0111: begin
                                     ior_n_to_the_enet<=1;
                                     nedk_card_bus_address<=3'b011;
                               end
                  4'b1111: begin
                                    command_cycle<=command_cycle+1;
                                    timing_cycle<=0;
                               end
             endcase
    end

6'b001101:
    begin  //read data from Rx_Tx_0
            timing_cycle<=timing_cycle+1;
            case(timing_cycle)
                  4'b0001: nedk_card_bus_address<=Rx_Tx_0;
                  4'b0010: ior_n_to_the_enet<=0;
                  4'b0111: begin
                                    ior_n_to_the_enet<=1;
                                    nedk_card_bus_address<=3'b011;
                               end
                  4'b1111: begin
                                     command_cycle<=command_cycle+1;
                                     timing_cycle<=0;
                               end
             endcase
    end

6'b001110:
    begin  //read data from Rx_Tx_0
            timing_cycle<=timing_cycle+1;
            case(timing_cycle)
                  4'b0001: nedk_card_bus_address<=Rx_Tx_0;
                  4'b0010: ior_n_to_the_enet<=0;
                  4'b0111: begin
                                    ior_n_to_the_enet<=1;
                                    nedk_card_bus_address<=3'b011;
                               end

                  4'b1111: begin
```

```
                                        command_cycle<=command_cycle+1;
                                        timing_cycle<=0;
                               end
                  endcase
          end

      6'b001111:
          begin  //read data from Rx_Tx_0
                 timing_cycle<=timing_cycle+1;
                 case(timing_cycle)
                        4'b0001: nedk_card_bus_address<=Rx_Tx_0;
                        4'b0010: ior_n_to_the_enet<=0;
                        4'b0111: begin
                                        ior_n_to_the_enet<=1;
                                        nedk_card_bus_address<=3'b011;
                               end
                        4'b1111: begin
                                        command_cycle<=command_cycle+1;
                                        timing_cycle<=0;
                              end
                 endcase
          end

      6'b010000:
          begin  //read data from Rx_Tx_0, which is the first 16 bits of the
                  //source IP address
                 timing_cycle<=timing_cycle+1;
                 case(timing_cycle)
                        4'b0001: nedk_card_bus_address<=Rx_Tx_0;
                        4'b0010: ior_n_to_the_enet<=0;
                        4'b0111: begin
                                       data1<=data_from_cs8900;
                                       ior_n_to_the_enet<=1;
                                       nedk_card_bus_address<=3'b011;
                               end
                        4'b1111: begin
                                       command_cycle<=command_cycle+1;
                                       timing_cycle<=0;
                              end
                 endcase
          end

      6'b010001:
          begin  //read data from Rx_Tx_0, which is the second 16 bits of
                   //the source IP address
                 timing_cycle<=timing_cycle+1;
```

```
                                        case(timing_cycle)
                                            4'b0001: nedk_card_bus_address<=Rx_Tx_0;
                                            4'b0010: ior_n_to_the_enet<=0;
                                            4'b0111: begin
                                                          data2<=data_from_cs8900;
                                                          ior_n_to_the_enet<=1;
                                                          nedk_card_bus_address<=3'b011;
                                                     end
                                            4'b1000: data_out<=(data1<<16)|data2;
                                            4'b1111:begin
                                                          command_cycle<=command_cycle+1;
                                                          timing_cycle<=0;
                                                    end
                                        endcase
                                   end

                              default:
                                       begin
                                            nedk_card_bus_address<=3'b000;
                                            ior_n_to_the_enet<=1;
                                            iow_n_to_the_enet<=1;
                                            IRQ<=0;
                                            data_out<=32'b0000_0000_0000_0000_0000_0000_00
                                            00_0000;
                                       end

                 endcase
              end
           end


           bustri  bustri_inst1 (
                               .data ( data_to_cs8900),
                               .enabledt ( ~iow_n_to_the_enet ),
                               .enabletr ( ~ior_n_to_the_enet ),
                               .tridata ( sfw_nedk_card_bus_data ),
                               .result ( data_from_cs8900 )
                               );

           endmodule
```

APPENDIX D NDK COMPONENTS

(See CD for Figure D.1 and Figure D.2)

In this appendix, the three components of NDK: SOPC Builder system development tool, Quartus II development software and GNUPro Toolkit Compiler and Debugger will be discussed.

D.1 SOPC Builder System Development Tool

The SOPC Builder [27] system development tool simplifies the task of creating high-performance system-on-a-programmable-chip (SOPC) designs by accelerating system definition and integration. Using SOPC Builder, a complete system can be defined and implemented, from hardware to software, within one tool and in a fraction of the time of traditional system-on-a-chip (SOC) design. SOPC Builder is integrated within the Altera Quartus II software to give Altera FPGA designers access to this development tool.

SOPC Builder is a platform for composing bus-based systems from common system components placed inside or outside the FPGA. The SOPC Builder library components supplied by Altera or other third party developers range from simple blocks of fixed logic, to complex, parameterized, and dynamically generated subsystems. SOPC Builder library components include (according to [27]):

- Processors
- Microcontroller peripherals
- Digital signal processing (DSP) cores

- Intellectual property (IP) cores
- Communications peripherals
- Interfaces
    - Memory (on-chip or off-chip)
    - Buses and bridges
    - ASSPs
    - ASICs
- Software components
    - Header files
    - Generic C drivers
    - Operating system (OS) kernels
    - Middleware libraries

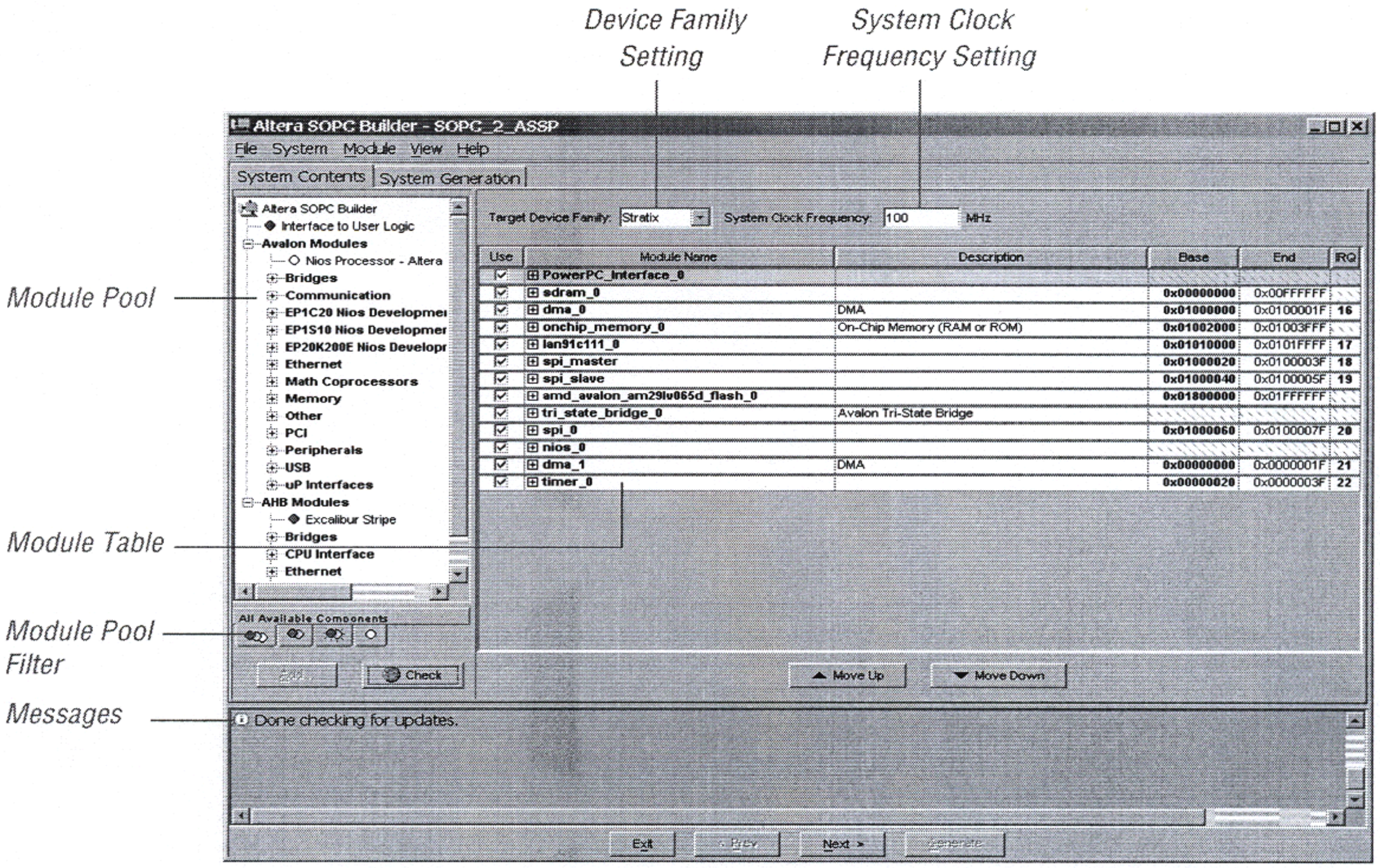
D.1.1 SOPC Builder Interface

After a Quartus II project is opened, the SOPC Builder user interface can be launched by choosing SOPC Builder (Tools menu) in the Quartus II software. The SOPC Builder user interface contains the following pages:

- System Contents page
- System dependency page(s)
- System Generation page

1. System Contents Page

This page is where system is defined. A listing of all available library components is included in the *module pool* and all of the components that have been added to a system are displayed in the *module table*. A single system module that includes components and specified interfaces is created when a system is generated with SOPC Builder. Additionally, automatically generated bus (interconnection) logic is contained in this single system module. Figure D.1 shows the System Contents page.



Target Device Family: Stratix System Clock Frequency: 100 MHz

Use	Module Name	Description	Base	End	IRQ
✓	PowerPC_Interface_0				
✓	sdram_0		0x00000000	0x00FFFFFF	
✓	dma_0	DMA	0x01000000	0x0100001F	16
✓	onchip_memory_0	On-Chip Memory (RAM or ROM)	0x01002000	0x01003FFF	
✓	lan91c111_0		0x01010000	0x0101FFFF	17
✓	spi_master		0x01000020	0x0100003F	18
✓	spi_slave		0x01000040	0x0100005F	19
✓	amd_avalon_am29lv065d_flash_0		0x01800000	0x01FFFFFF	
✓	tri_state_bridge_0	Avalon Tri-State Bridge			
✓	spi_0		0x01000060	0x0100007F	20
✓	nios_0				
✓	dma_1	DMA	0x00000000	0x0000001F	21
✓	timer_0		0x00000020	0x0000003F	22

Done checking for updates.

Figure D.1 The System Contents Page (from [27])

1) Module Pool

All available library components organized according to bus type and category is showed in the module pool. A colored dot is used to indicate each component appearing next to its name.

2) Module Table

The module table is where components are added to the system, including bridges, bus interfaces, CPUs, memory interfaces, peripherals, etc.

Additionally, the following elements are described using the module table.

- Master and slave connectivity
- System address map
- System IRQ assignments
- Arbitration priorities for shared slaves

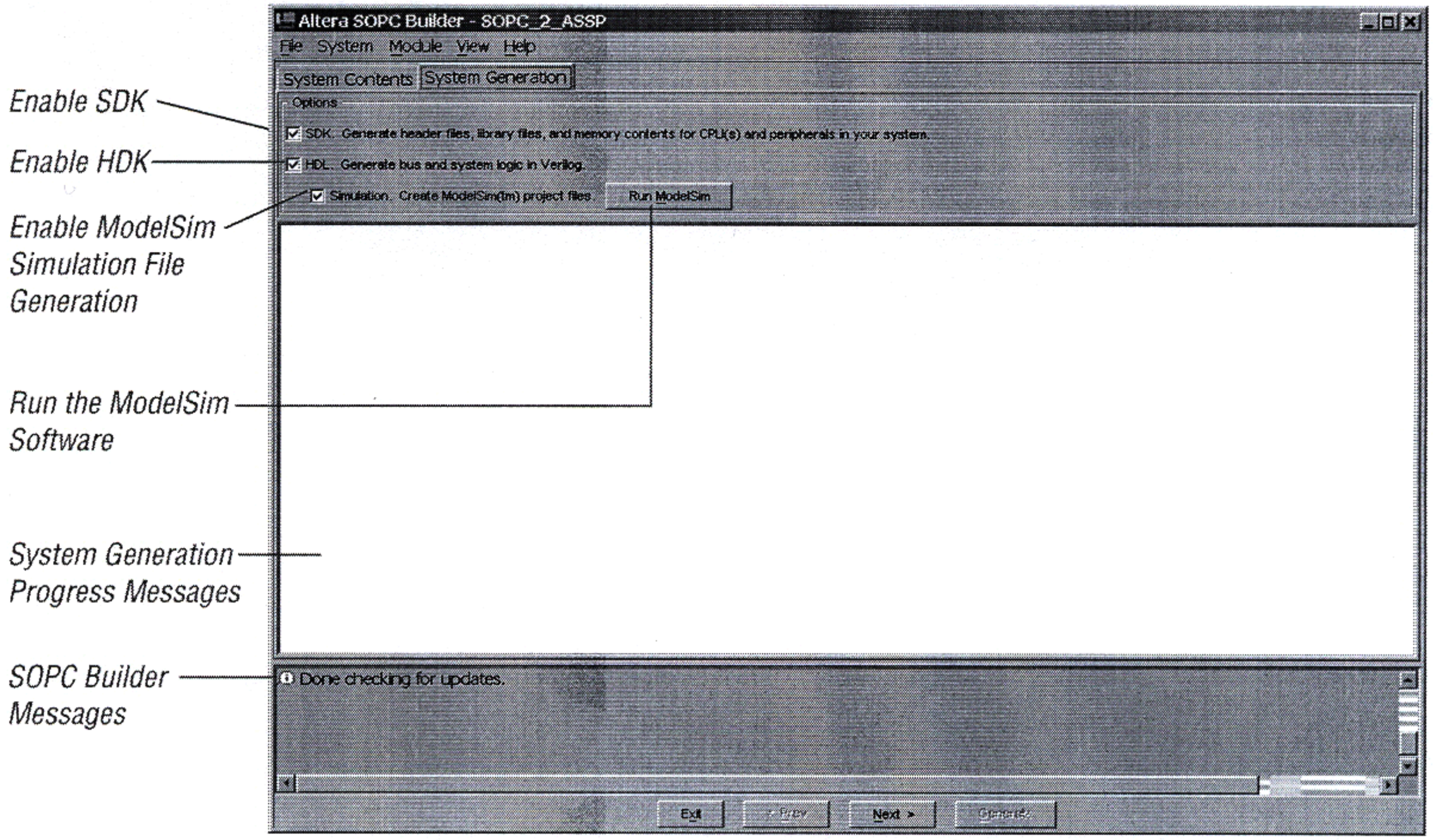
2. System Dependency Page(s)

When certain components are added to a system, such as a CPU like the Nios embedded processor, an additional page(s) appears in SOPC Builder. These page(s) allow setting additional parameters or associations of the component with respect to the other components in the system. For example, the relationship between a CPU and the memory components can be specified to indicate which portion is used as the program memory and which portion is used as data memory. For components that use system dependency pages, a separate system dependency page is created for each instance of the component that has been added to a system.

Additionally, if the processor components have some associated software components, the software components will be shown on the system dependency pages. The software components examples range from utility libraries to real-time-operating systems (RTOSs). Several software components are provided by Altera in development kits, such as the Plugs Library (a compact, full-featured TCP/IP protocol stacks) that comes with the Nios development kits.

3. System Generation Page

This page is where a system is generated. It includes options that can be set to control the generation process such as device family support and simulation. This page reports the system generation progress message(s) during system generation. Figure D.2 shows the System Generation page.


Enable SDK
Enable HDK
Enable ModelSim Simulation File Generation
Run the ModelSim Software
System Generation Progress Messages
SOPC Builder Messages
Altera SOPC Builder - SOPC_2_ASSP
File System Module View Help
System Contents
System Generation
Options
Run ModelSim
Exit
< Prev
Next >
Generate

☑ SDK. Generate header files, library files, and memory contents for CPU(s) and peripherals in your system.

☑ HDL. Generate bus and system logic in Verilog.

☑ Simulation. Create ModelSim(tm) project files.

Done checking for updates.

Figure D.2 The System Generation Page (from [27])

1) SDK

When the SDK option is turned on, SOPC Builder creates a custom SDK for each CPU in a system every time the system is generated. Software files (drivers, libraries, and utilities) for any system components that provide software support in their library definition is contained in the SDK.

Software applications can be built as part of the generation process by the processor components, such as the Nios embedded processor and Excalibur™ devices.

The following directories are the directories that the software files are arranged into:

- inc—Three things are contained in this directory: header files with the definitions of memory maps, register declarations for the peripherals, and macros that can be used to create embedded software applications.
- lib— The library files is contained in this directory. If the component supports GNU tools, the libraries are compiled by SOPC Builder during system generation.
- src— The source code is contained in this directory. The source code can be written and modified for the system using any text editor, also the Quartus II Text Editor can be used, which supports syntax coloring for C and C++ source code.

2) HDL

When the HDL option is turned on, SOPC Builder generates a system-level hardware description language (HDL) file in Verilog HDL or VHDL, depending on which language is specified when the system in SOPC Builder is first set up. The HDL file contains (according to [25]):

- An instance of every component in the system
- Bus logic to interconnect the components, including the following items:
  - Address decoders
  - Data bus multiplexers
  - Arbiters for shared resources
  - Reset-generation and conditioning logic
  - Interrupt prioritization logic
  - Dynamic bus sizing (for adapting masters to slaves with wider or narrower data buses)
  - Passive interconnections between master and slave ports
- A simulation testbench that:
  - Instantiates the system module
  - Drives clock and reset inputs with default behaviors
  - Instantiates and connects any simulation models for system external components if provided (e.g., memory models)

4. Generating a System

After a system is built and generation options are specified, the system can be generated by clicking the Generate button. SOPC Builder creates the following items (according to [25]):

- The SDK
- HDL files for each component in the system
- A Block Symbol File (.bsf) for the top-level system module
- ModelSim files
- A Tcl script that sets up all of the files needed for Quartus II compilation

D.2 Quartus II Development Software

The Quartus II development tool allows designers to process multi-million gate designs and streamline development flows. A comprehensive environment for SOPC design is provided by the Quartus II development software. Because of its interfaces to industry-standard EDA tools the software integrates into nearly any design environment. Also, an embedded logic analysis feature provides the ability to verify chip functionality and timing by observing internal and I/O signal values at system clock speeds (SignalTap). This feature was used extensively in the development of the Silicon Firewall.

The Nios development kit includes the Quartus II development software, which contains support for the EP20K200EFC484 device that populates the Nios development board.

D.3 GNUPro Toolkit Compiler and Debugger

The GNUPro toolkit from Red Hat is an industry-standard compiler and debugger tool suite, which is an open-source C/C++ development tool suite optimized for the Nios embedded processor. An environment familiar to software design engineers is provided by the GNUPro toolkit.