DESIGN AND IMPLEMENTATION OF A HARDWARE LEVEL CONTENT NETWORKING FRONT END DEVICE

by

JEREMY LAYNE BUBOLTZ
B.S. University of Central Florida, 2006

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2007

©2007 Jeremy Layne Buboltz

ABSTRACT

The bandwidth and speed of network connections are continually increasing. The speed increase in network technology is set to soon outpace the speed increase in CMOS technology. This asymmetrical growth is beginning to causing software applications that once worked with then current levels of network traffic to flounder under the new high data rates. Processes that were once executed in software now have to be executed, partially if not wholly in hardware. One such application that could benefit from hardware implementation is high layer routing. By allowing a network device to peer into higher layers of the OSI model, the device can scan for viruses, provide higher quality-of-service (QoS), and efficiently route packets. This thesis proposes an architecture for a device that will utilize hardware-level string matching to distribute incoming requests for a server farm. The proposed architecture is implemented in VHDL, synthesized, and laid out on an Altera FPGA.

TABLE OF CONTENTS

LIST OF FIGURES VI

LIST OF TABLES VII

LIST OF ACRONYMS/ABBREVIATIONS VIII

CHAPTER ONE: INTRODUCTION 1

The OSI Model 1

The Physical Layer 3

The Data Link Layer 3

The Network Layer 4

The Transport Layer 5

The Application Layer 6

The TCP/IP Stack 6

The HTTP GET Request 7

Content Networking 8

Front End Devices 9

String Matching 11

CHAPTER TWO: PREVIOUS WORK 13

CHAPTER THREE: DESIGN 15

Proposed Architecture 15

Sample Case 18

CHAPTER FOUR: RESULTS 20

Component Waveforms 20

Character Decoder 20

D-Type Flip Flop 21

Individual State 21

State Encoder 22

Port Look-Up Table 23

Port Selector 24

System Simulation 24

Synthesis 26

Layout 28

CHAPTER FIVE: CONCLUSIONS AND FUTURE WORK 32

LIST OF REFERENCES 33

LIST OF FIGURES

Figure 1 - The OSI Model... 2

Figure 2 - The TCP/IP Stack... 7

Figure 3 - Example Of A FED In Front Of A Server Farm.. 10

Figure 4 - The Finite State Machine Approach Of The Aho-Corasick Algorithm...................... 12

Figure 5 - FED State Diagram 15

Figure 6 - The Architecture Of The Proposed FED Design 16

Figure 7 - The VHDL Architecture Of The Proposed FED.. 17

Figure 8 - Block Diagram Of An Individual State.. 17

Figure 9 - State Graph Of Proposed FED 19

Figure 10 - Character Decoder Waveform... 20

Figure 11 - D-Type Flip Flop Waveform 21

Figure 12 - Individual State Waveform 22

Figure 13 - State Encoder Waveform 23

Figure 14 - Port Look-Up Table Waveform 23

Figure 15 - Port Selector Waveform... 24

Figure 16 - Completed Simultation... 25

Figure 17 - Annotated RTL View.. 27

Figure 18 – FPGA Utilization For The Proposed FED.. 30

LIST OF TABLES

Table 1 - HTTP V1.1 Methods 8

Table 2 - FPGA Resource Usage Per VHDL Entity... 28

Table 3 - Detailed Timing Analysis For Proposed FED.. 30

LIST OF ACRONYMS/ABBREVIATIONS

ALU	Arithmetic Logic Units
ALUT	Adaptive Look Up Table
API	Application Programming Interface
ARP	Address Resolution Protocol
ASCII	American Standard Code for Information Interexchange
FED	Front End Device
FTP	File Transfer Protocol
HTTP	Hyper-Text Transfer Protocol
IP	Internet Protocol
ISO	International Organization for Standardization
LC	Logic Cell
MAC	Media Access Control
OSI	Open Systems Interconnection
QoS	Quality of Service
RTL	Register Transfer File
SMTP	Simple Mail Transfer Protocol
TCO	Typical Case Optimization
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VHDL	VHSIC Hardware Description Language

VHSIC	Very High Speed Integrated Circuit
WWW	World Wide Web

CHAPTER ONE:
INTRODUCTION

The bandwidth and speed of network connections are continually increasing. The speed increase in network technology is set to soon outpace the speed increase in CMOS technology. This asymmetrical growth is beginning to causing software applications, which once worked with then current levels of network traffic, to flounder under the new high data rates. Systems that are based on software applications not only have to process the software application but also have the overhead involved in unpacking/packing data through the network protocol stack. Processes that were once executed in software now have to be executed, partially if not wholly in hardware. By allowing a network to peer into higher layers of the OSI model, the network can scan for viruses, provide higher quality-of-service (QoS), and efficiently route packets. One such application that could benefit from hardware implementation is high layer routing.

The OSI Model

Currently, the International Organization for Standardization (ISO) maintains a suite of standards, commonly referred to as the Open Systems Interconnection (OSI) model, that define the layers and the partitions that separate these layers of many network architecture models. ISO is comprised of over 150 national standards bodies and contains over 5,200 technical bodies and working groups. Currently the organization has over 16,000 standards. ISO defines standards for policies, processes, services, systems, materials, and products. ISO standards make clear the requirements for products, allowing suppliers to compete worldwide. The standards disseminate

expert information, allowing room for improvement. A number of these standards relate to networking and comprise the OSI model [17, 19].

The OSI model is layered, meaning that related communication functions are abstracted out into a layer, which will commonly be encapsulated into a single protocol. A particular application of the network will then specify the use of a subset of the protocols. This subset is commonly referred to as a protocol stack. The layered model is beneficial when changes to a particular aspect of network communications are needed. Because the interfaces between the layers are defined, when an optimization or correction to the function of a layer needs to be made, it can be done without affecting the other layers. Also, because multiple protocols can exist to perform the same function for a particular layer, for a specific network communication, the most efficient protocol for any given layer can be chosen [15, 16].

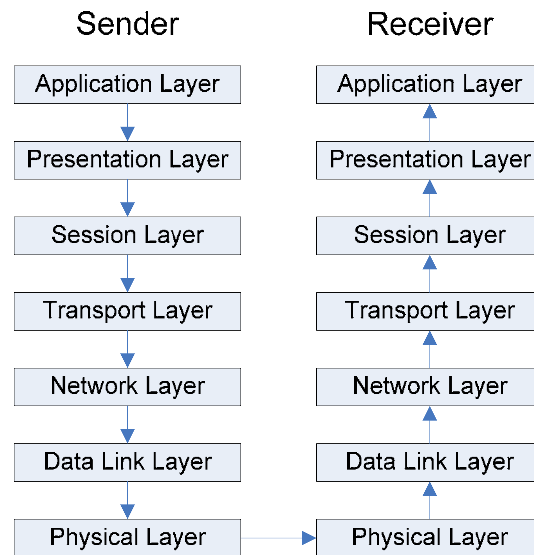

Figure 1 - The OSI Model

The seven layers of the OSI model are: Application, Presentation, Session, Transport, Network, Data Link, and Physical. Figure 1 illustrates the order in which the layers are processed

when sending or receiving data. The OSI architecture model illustrates the common, agreed upon, protocols from the majority of manufactures. However, even though the standards have been agreed upon for years, they are in continual evolution. An example of this evolution is the widely accepted TCP/IP stack, which is commonly used to transport internet traffic. The TCP/IP stack does not utilize the Presentation or Session layers [15, 16, 17]. The following sections review the five commonly used layers of the TCP/IP stack, followed by a more detailed look at the TCP/IP stack.

The Physical Layer

The Physical Layer standards define the medium and means with which signals are sent. This includes the physical device interfaces, physical medium, device security, voltage levels, and timing requirements. There is a multitude of different Data Link Layer protocols, the Physical Layer standards must provide means to send data using Ethernet, BNX, or WiFi. Each standard can also provide for varying speeds. As technology improves, so does the standard to provide for the fastest network possible [16, 18].

The Data Link Layer

The Data Link Layer provides connection between two individual, sequential, nodes of a network. Some of the more common Data Link Layer protocols include: Ethernet, Token Ring, 802.11 wireless (WiFi), and arguably, various fiber optic connections such as ATM and FDDI. The Data Link Layer is responsible for moving data across a single link (or node-to-node) in the

network path. The combination of the Data Link Layer and Physical Layer is commonly referred to as the Media Access Control (MAC) protocols. A MAC address of a machine is the, one of a kind, physical hardware address of that machine [15, 16, 20, 21].

Many of the Data Link Layer protocols provide a multitude of services to assure that a datagram has been passed (node-to-node). The Data Link Layer will encapsulate the packet passed down from the network layer in the data portion of its own packet, thus adding an additional header onto the packet. This technique of encapsulating is commonly called Framing. Data Link Layer protocols can also implement several features to control access to the physical medium. The link access, reliable delivery, and flow control features of many Data Link Layer protocols: assure that two senders don't try to send data at the same time corrupting both sets of data; verify that the receiver received the data that was sent; and assure that the sender does not send too much data causing the receiver to overfill buffers and drop packets. Many of the protocols provide means for error detection while few protocols can perform error correction. Also, Data Link protocols can be either full or half-duplex [15, 16].

The Network Layer

The goal of the Network Layer is very similar to the Data Link Layer: to get data from the sending machine to the receiving machine. However, for the Network Layer, the two machines might not be directly, physically connected. The Network Layer must assure that a packet has traveled from the sender to the receiver, across however many other machines it might take. In essence, the Network Layer allows machines on different networks to be able to communicate. The main device that makes this cross-network communication possible is the

router. The Network Layer protocols often provide means for guaranteed, in-order, packet delivery [15, 16].

The IP protocol has a mechanism to address a machine with an IP address. Current IPv4 addresses contain 32 bits with which to identify a machine. However, more evidence of the constant evolution of networking is the gradual move towards IPv6. IPv6 addresses contain 128 bits with which to identify machines. A host machine will only have a single Data Link Layer MAC address per network interface. However, each network interface may be associated with multiple Network Layer IP addresses. The mechanism to translate between IP addresses and MAC addresses is called the Address Resolution Protocol (ARP) [15, 16, 20, 21].

The Transport Layer

The transport layer provides an abstraction so that many of the networking details are removed from the programmer's perspective. The transport layer extends the addressing scheme beyond destination addresses (IP addresses) and specifies a port on the destination machine. Typically, applications will bind to a specific port on a machine. For instance, port 80 has been defined for HTTP use. So, many web servers will default to using port 80. The transport layer can be implemented using either reliable or unreliable transmissions. The most common reliable communications means is TCP. TCP uses hand-shaking to make and break connections between the host and destination machines. Once a connection is made, the destination will acknowledge receiving any data that the sender has sent. This allows the sender to know if any data has been dropped. The most common unreliable Transport Layer protocol is UDP. UDP is a connectionless protocol, meaning that it does not perform any handshaking. UDP is often used to

stream either voice or video to multiple viewers. If a packet where to be dropped, by the time the receiver asked for it to be resent, the data would no longer be valid. Thus, there is no need to provide guaranteed delivery of the packet [15, 16].

The Application Layer

The Application Layer directly provides the networking services to the user. Just like the other layers, the data structures of the Application Layer must be agreed upon by the sender and the receiver. Some of the protocols of the Application Layer are: HTTP, SMTP and FTP. The different protocols of the Application Layer often provide APIs that abstract away much of the difficulties of setting up network communications within a program. An application programmer simply needs to choose the services required by their application. Reliable Data Transfer, Bandwidth, and Delay are all factors that can be affected by what lower level protocols are used [15, 22].

The TCP/IP Stack

The most commonly implemented protocol stack used for internet traffic is the TCP/IP stack. With the details of the individual layers previously discussed, an example of how the layers work together will now be covered. An example implementation of using an internet browser and viewing a webpage would use the following protocols per layer: HTTP at the Application Layer, TCP at the Transport Layer, IP at the Network Layer, then Ethernet/CAT5 at the Data Link/Physical Layer. Figure 2 illustrates the relationship that the layers have to each

other. Two of the most common devices that internet traffic travels through are switches and routers. Switches work at the Data Link Layer and allow point to point communication of machines on the same network. Routers work at the Network Layer and allow devices on different networks to communicate with each other [22].

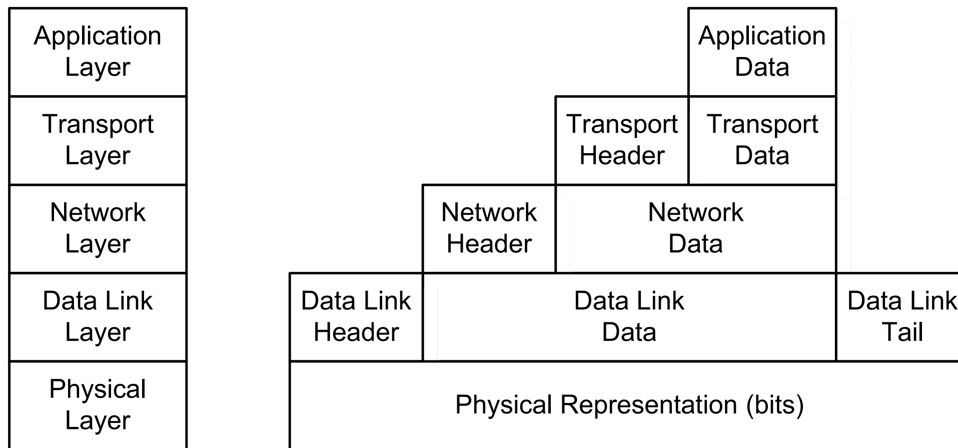

Figure 2 - The TCP/IP Stack

The HTTP GET Request

The Hyper-Text Transport Protocol (HTTP) has become the dominant vehicle to convey information across the World Wide Web (WWW). As mentioned earlier, HTTP is an Application Layer protocol that is used to communicate between clients and servers. HTTP has several methods that provide the ability to send, retrieve, and modify data on a server. As of version 1.1, there are 7 HTTP methods. These methods are: Options, Get, Head, Post, Put, Delete, Trace, and Connect. Table 1 lists each method with a short description of what the method does [24, 25]. For the purposes of this thesis, the GET request is of highest interest.

Table 1 - HTTP V1.1 Methods

Method	Description
OPTIONS	Requests available options for URL
GET	Retrieves requested URL (Headers & Body)
HEAD	Retrieve Requested URL (Headers only)
POST	Sends information to the server
PUT	Uploads the file to the indicated URL
DELETE	Deletes the URL from the server
TRACE	Used to troubleshoot communications with the server
CONNECT	Used for proxy server content tunneling

To identify an HTTP packet, we will verify the packet from the top down. First, the Ether Type field in the Ethernet header is checked. A decimal value of 2048 identifies the packet as an IP packet. Next, the Protocol field of the IP header is referenced. A decimal value of 6 identifies the packet as a TCP packet. The HTTP GET Request is then denoted by the first 3 bytes of data in the Application Data being the ASCII value of the letters 'GET'. Next, the GET Request contains the URL to be retrieved. The URL is followed by optional header lines, which contain metadata about the requester. The GET request is then terminated by a Carriage Return with an optional body section [23, 25].

Content Networking

Content networking (also known as layer-7 routing) has many applications. By accessing the payload of packets, more details about that packet can be gained. Content networking can have impacts when implemented in load balancing, QoS support, session integrity guarantee and flexibility in content deployment. Routers, firewalls, spam and virus filters could all benefit

from Content networking. Some of these applications, like firewalls, spam and virus filters, require analyzing the entire contents of a packet. However, some applications do not require analysis of the entire payload. For routing, a particular aspect of the payload is of interest. Perhaps the field of interest is the host, a cookie or a URL [7].

A difficulty in content networking, which causes content switches to be relatively rare, is that most of the high layer protocols are designed to be worked with on general purpose CPUs and also involve complex protocol processing [11]. For those applications that require inspecting the entire payload, the major bottleneck is having to process the entire payload. This can consume a vast amount of resources. This is especially true of software based systems that are limited by their base hardware which was designed for general purpose execution. By implementing a single purpose hardware device, the speed of the device will be greatly increased. The ever increasing speed of network traffic is only stressing this point.

Front End Devices

Clusters of workstations or PCs are becoming a popular platform to deliver websites and content on the Internet. This architecture has proven to provide high performance for a relatively low cost [8, 9]. Another benefit is the ability to scale the resources to meet the demands placed on the system. To deliver requests to the individual machines of a server farm, a device is needed to accept all incoming traffic and to assign a request to a particular machine to handle (see Fig. 1) [10]. These devices are called Front End Devices (FEDs), because they sit at the front of a server farm accepting all requests. A major problem now arises, if a similar workstation or PC is

used for a FED, then the number of attached machines in the server farm is typical limited to ten or less [9]. This limits the scalability of the server farm, detracting from one of the major benefits of this architecture.

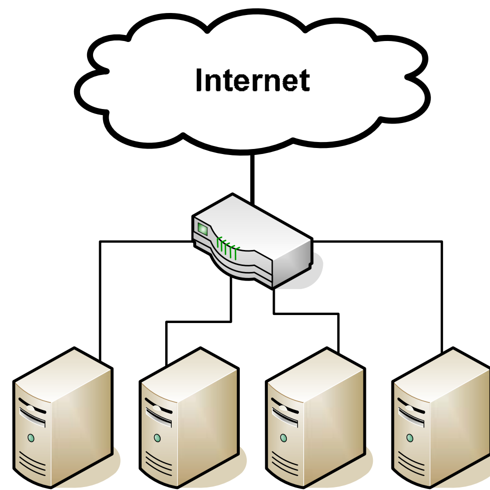

Figure 3 - Example Of A FED In Front Of A Server Farm

In this thesis, a design and implementation of a hardware FED is presented. A FED is used as a gateway to a group of servers. This device is used to accept incoming requests and to pass the request to the appropriate server [10]. One application would be a group of web servers specialized for specific tasks. For example, an administrator might assign a specific server to handle all multi-media requests, whereas another server is used to handle all XML processing. Or perhaps a server has been assigned to hold databases and handle all database requests. Because of the ever increasing size of the Internet and increasing network speeds, a software based FED could be stressed to handle the load of a moderate server farm. However, by implementing the FED in hardware only then the increase in operating speed would result in the ability to handle larger server farms.

String Matching

The most important component of implementing a hardware FED is being able to find the appropriate server the packet needs to be assigned to. To do this, the hardware FED must match the request with one of a known set of strings. Therefore, a strong hardware level string-matching mechanism is needed.

With a wide variety of applicability, string matching has been continually optimized. String matching can be applied to all sorts of problems. String matching solutions can be categorized into two subcategories: exact string matching and approximate string matching. Each field can then be further divided into hardware and software based solutions. Approximate string matching is commonly used for spell checking or other related functions. Exact string matching is more suited for the requirements of a FED. Two of the most popular algorithms for exact string matching are Aho-Corasick [1] and the Boyer-Moore [12] algorithms. The Boyer-Moore algorithm is sub-linear for the average case. However, for length of text, n, and length of pattern, m, in the worst case complexity is $O(n*m)$. For the Aho-Corasick algorithm, time complexity is $O(n+m)$. Another method to search strings is with a content addressable memory (CAM). A CAM can search a specific length of data in a single memory access. By activating every memory cell with every search, the CAM has proven to be very fast. However, this method also consumes lots of power and can take up a fair amount of silicon area [2, 13].

The Aho-Corasick (AC) algorithm is capable of being implemented as a finite state machine. Characters are fed into the state machine serially, with each successful match leading further down a state path to a final state of a match. Failures will either result in a return to the initial state or a jump to another portion of the state graph [1]. With the AC algorithm, only one

state will be active at a time. Thus, the AC implementation will consume less power than a CAM implementation. With the AC algorithm's logical transition to a hardware structure the control circuitry needed will be minimized. The Boyer-Moore algorithm would have needed a large amount of control circuitry to implement in hardware. This results in a larger design with a larger energy footprint than an AC implementation.

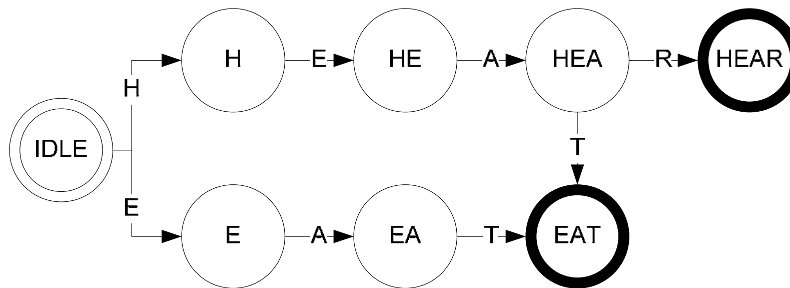

Figure 4 - The Finite State Machine Approach Of The Aho-Corasick Algorithm.

CHAPTER TWO:
PREVIOUS WORK

In [7], a router is implemented that checks for the URL in a request packet. There was a large overhead found related to the retrieval and converting the string. The solution was to format a link on a website into a code. The special links were denoted with a preamble before to coded portion of the link began [7]. The disadvantages are that the router could not be used on an existing website without any work having to be done to the website. Also, a factor of the human interface of the website was degraded by encoding the URL of links.

In [6], a firewall was implemented in hardware. By allowing the firewall to search the content of the packet, it was capable of performing spam and virus filtering. The firewall also limited the rate of per-flow traffic to prevent denial-of-service attacks. The firewall was implemented as a system-on-programmable-chip with an architecture that will help mitigate against future attacks as new exploits are developed. If an ASIC was used, then when adding extra functionality the entire chip would have to go through the design process again. By implementing their design in hardware, the authors were able to achieve multiple gigabit throughput speeds.

Much of the current research has been applied towards creating high wire-speed network intrusion detection systems (NIDSs). Because of the large number of threats faced by networks and computers and increasing line speeds, software based NIDS have become unusable [3, 4, 14].

A method is presented in [5] to reduce space consumption by removing the character matchers from the individual states and creating a single ‘character decoder.’ This helps to

eliminate repetition in the design. This method is also implemented in the FED proposed in this thesis.

In [11], work was performed to help eliminate a difficulty in the content-based switching. To access any payloads, a TCP connection must first be made. Once the connection has been made with the switch, it is difficult to migrate the connection to the server. In [11], a form of TCP splicing allows the port controllers on the switch to handle any minor processing needed to continue the connection. This allows the processor to be freed for other actions.

CHAPTER THREE:
DESIGN

Proposed Architecture

To help illustrate the proposed FED architecture lets first follow the flow of a packet through the system. As packets arrive, the IP header is checked to verify that it is a TCP packet. If a packet is determined to be a GET request, the application layer data is extracted from the packet in the stored buffer. Next that data is sent, a byte at a time, to the Character Decoder. The Character Decoder is used to eliminate the need for individual decoders located at each state. After being decoded, a single output line will be enabled. If that output line is connected to the first state, then that state will become active. The process will continue the same way for the next byte. Once one of the output states has been reached, the state encoder will encode the data and send it to the port look-up table as an address. The port look-up table will hold a value that is used to select a port. Once a port has been selected, the data will move from the Buffer, through the Port Selector and out the specified port.

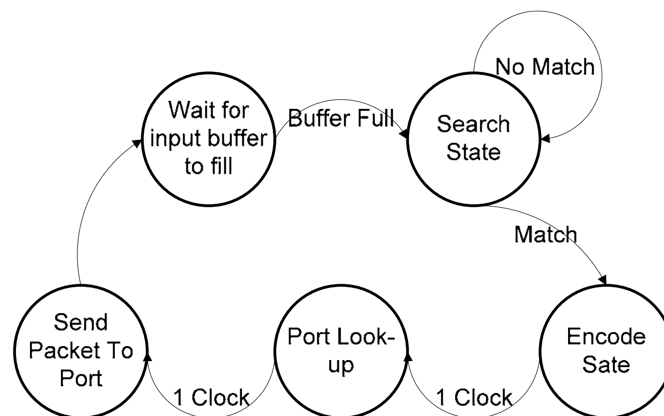

Figure 5 - FED State Diagram

Figure 5 is a state diagram that illustrates the described flow through the proposed FED. This diagram shows some of the communication lines that will need to be present to help the control circuit to navigate from state to state. Figure 6 is a diagram of the architecture. By following the description above, a logical flow through the state diagram, as well as through the architecture should be evident.

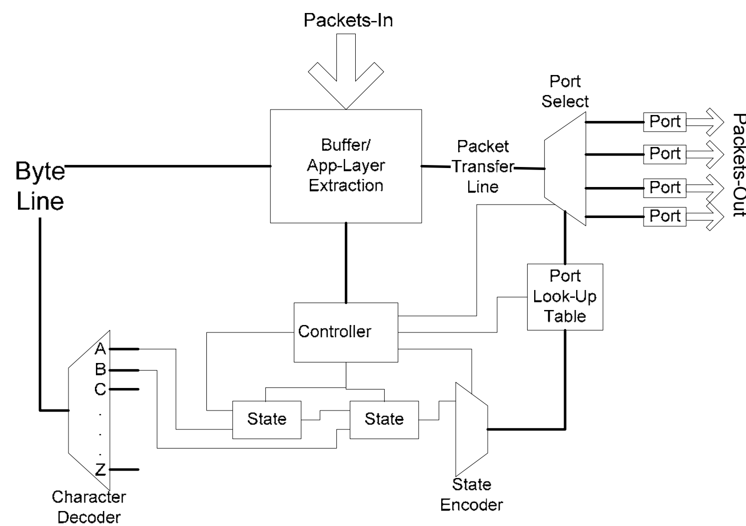

Figure 6 - The Architecture Of The Proposed FED Design

Figure 7 depicts the design hierarchy for implementing the proposed architecture in VHDL. This diagram shows how the individual components are architected and how these components are then joined together to form the completed overall design. From the diagram, there are seven individual components that will be utilized. Most will only have one instance, whereas the IndividualState component will require multiple instances.

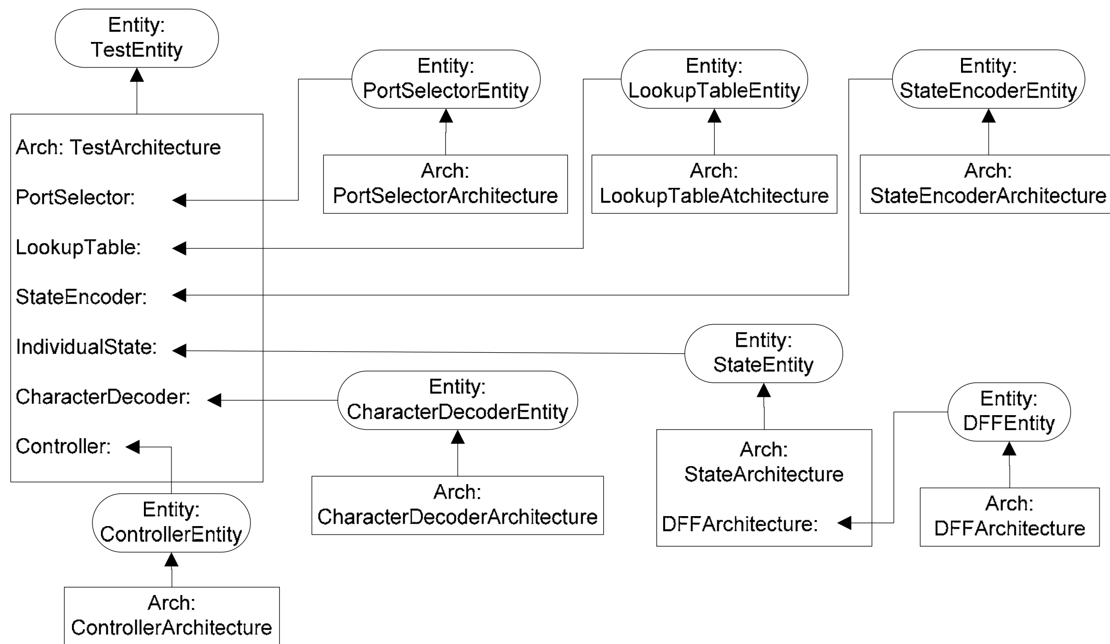

Figure 7 - The VHDL Architecture Of The Proposed FED

An individual state is diagramed in Figure 8. There are three inputs and a single output. The inputs are an enable, the character line from the Character Decoder, and the clock signal. The enable line is the output from the pervious state. It takes both the enable line and the correct character line to set a state. If all of the possible characters were needed then the Character Decoder would be an 8-to-256 decoder. Given *J* number of search strings, the State Encoder will be a J-to-$log_2(J)$ encoder. Given *N* number of ports, the memory will be $log_2(J)$ long and $log_2(N)$ wide. The port selector will then be a 1-to-N demultiplexer.

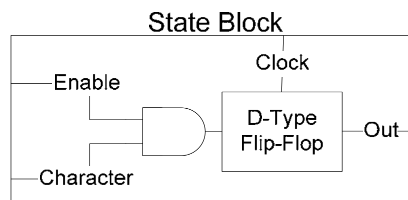

Figure 8 - Block Diagram Of An Individual State

Sample Case

To implement the design and test the overall system, an imaginary server farm was created. The system entailed four separate servers for specific purposes. Therefore, the proposed FED will have a single port entering and four ports exiting to the servers. First, all video requests will be tasked to a single server. Second, there is a server for XML processing. Third, there is a server for basic file transfers. And lastly, there is a default server that all remaining traffic will be sent to. Figure 9 lists all of the file types that will be matched by the proposed FED. The file types selected were chosen because of their prevalence on the Internet. If the proposed FED reaches the end of a packet and no match was found, then the packet will be sent to the default server.

In total there are 17 characters represented (not case-sensitive). Therefore, the Character Decoder will be an 8-bit to 17 output lines. Next, there will be 41 D-Flip-Flop gates. There are fifteen end states, with a sixteen end state of no match. So, the State Encoder will be a 16 line to 4-bit encoder. Then the Port look-up Table will be 16 rows by 2-bits. The Port Selector will then be a 1-to-4 MUX with a 2 bit select line.

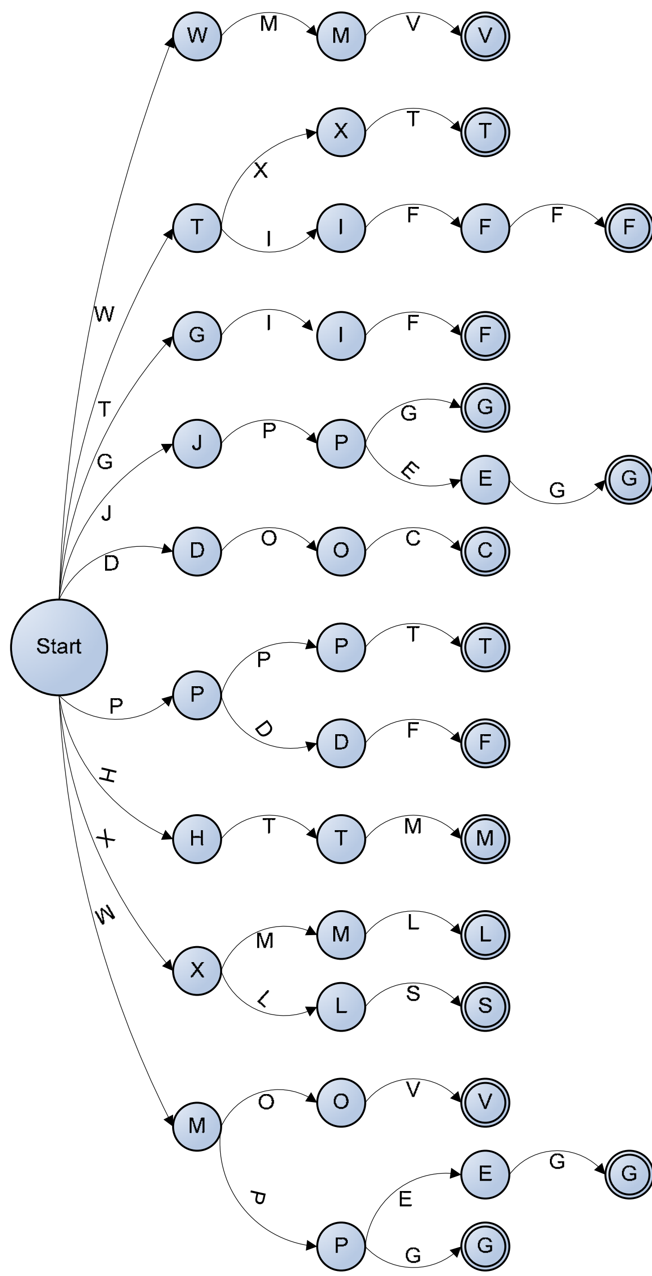

Figure 9 - State Graph Of Proposed FED

CHAPTER FOUR:
RESULTS

Component Waveforms

Each of the individual VDHL entities were constructed and tested before they were integrated into the VHDL FED entity. Below, the waveforms from the test benches of the individual components are explained. For the VHDL code that was used to generate each waveform see Appendix A.

Character Decoder

The character decoder is going to take the individual bytes of a packet and assert a specific output line if the character matches. This specific character decoder is designed to assert an assigned letter output if the input is either the lower-case or upper-case ASCII value of that letter. Below an output line assigned to a specific letter is asserted when either the lower-case or capitalized ASCII value of a letter is applied.

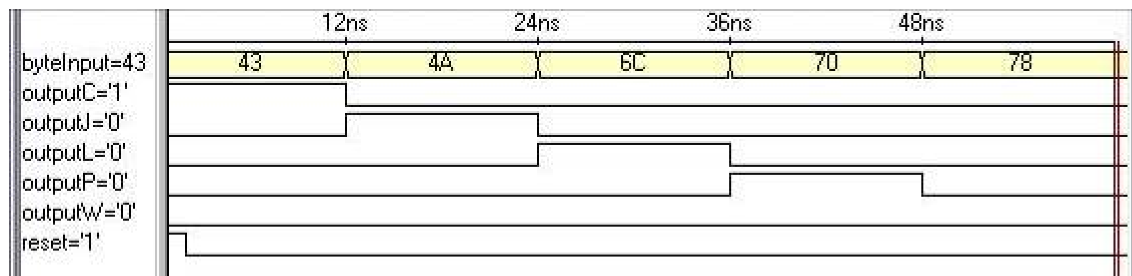

Figure 10 - Character Decoder Waveform

From Figure 10, with an input value of x43 (the ASCII value for the capital letter 'C') then the C line is asserted. When the input value changes to x4A (the ASCII value for the capital letter 'J') then the J line is asserted. Then when the input value is changed to x6C (the ASCII value for the lower-case letter 'l') then the L output is asserted. At any one time, only a single output line can be asserted.

D-Type Flip Flop

The D-type Flip-Flop is the foundation of the individual state. The output (Q) of the flip-flop is asserted on the rising edge of the clock when the input (D) is asserted.

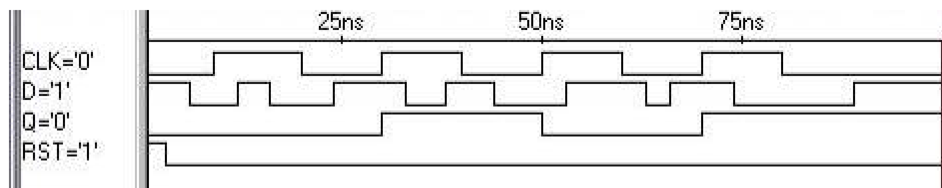

Figure 11 - D-Type Flip Flop Waveform

From the diagram, 10ns is the first rising-edge of the clock. However, the input D is not asserted at 10ns, therefore the output Q stays logical 0. At the next rising-edge, at 30ns, the input D is asserted. Because the input is asserted on the rising edge, then the output, Q, is asserted.

Individual State

The individual state is built off of the D-Flip-Flop. In the Architecture section of the paper is Figure 8 which shows that the state has two inputs (Previous State and Character). If

both of the individual state's inputs are asserted on the rising clock edge, then the output (Next State) will be asserted. In Figure 12 below, during the first three rising-edges of the clock there is either none, or only one of the inputs asserted. However, on the fourth rising-edge of the clock, at approximately 47ns, both of the inputs Character and PreviousState are asserted. This results in the output NextState to be asserted until the next rising edge of the clock.

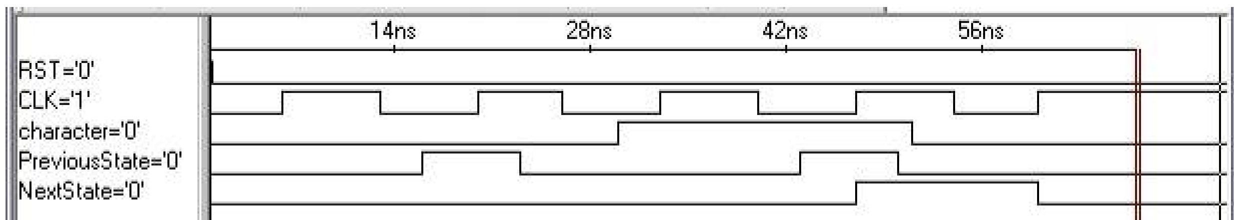

Figure 12 - Individual State Waveform

State Encoder

The state encoder will receive one of the end states and will encode which of the final states is reached into a four bit binary number. The State Encoder has 16 inputs, each representing a different file format. These inputs are connected to the NextState outputs of the final state for each of the state chains for the 16 different file formats. Therefore, once the last letter of a match has been input to the state machine, then the output of the last state to assert an input to the State Encoder.

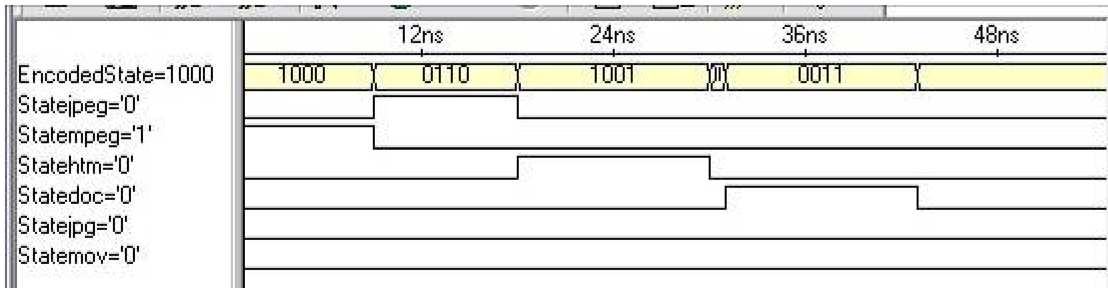

Figure 13 - State Encoder Waveform

In the Figure 13, we can see that as one of the input lines is asserted, a different value for the encoded state is output. The exact encoded state value is not particularly important other then it matches the value known for that state in the Port Look-up Table.

Port Look-Up Table

The Port Look-up Table is used to store the port number that will be used for each specific end state. Therefore since there are 16 end states, which get encoded into a number, the input is 4-bits long. Also, since there are only 4 ports, the output only needs to be 2-bits long to support the encoded value.

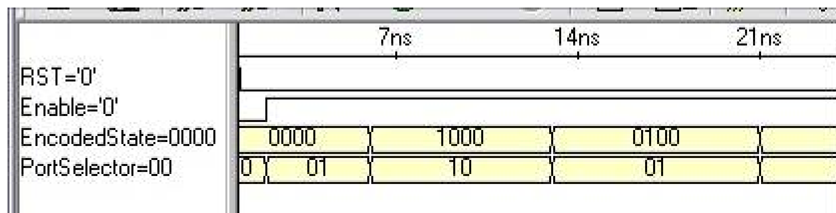

Figure 14 - Port Look-Up Table Waveform

In the Figure 14, the Look-up Table is programmed to select port 1 when the end state was one of the first 8 and to select port 2 when the end state is one of the last 8 end states.

Port Selector

The Port Selector has similar functionality to a 4-to-1 MUX. Depending on the output value of the look-up table, one of the ports will be selected. Then any input that is placed on the inLine will be output to the selected port, and only the port that is selected.

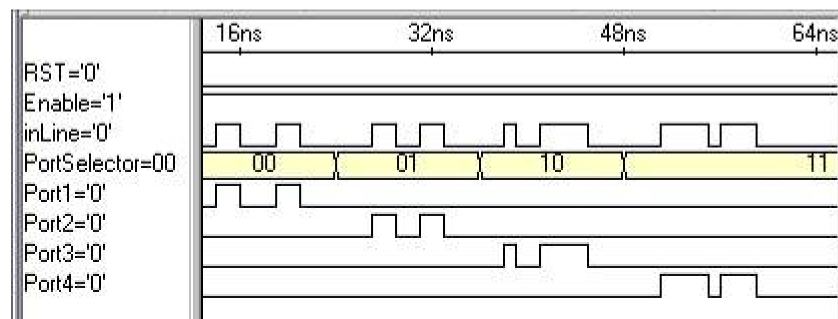

Figure 15 - Port Selector Waveform

System Simulation

Once all of the components are developed and tested individually, the components are connected for integration and system level testing. The overall circuit has been simulated with the results shown in Figure 16. The packet data is read in and transferred using the 's' signal. The data is then passed to the Character Decoder. The Character Decoder then asserts the appropriate output lines, which are connected to the inputs of the individual states. Next, we can see that the proposed FED matches a file format as states 17, 18 and 19 are traversed. These states represent the three letters of the file format 'gif'. Because of the delay for the characters to get decoded

and for the D-flip-flops to latch onto the output in the Individual States we can see that the Individual States actually output an active signal a full clock cycle after the byte has been passed to the Character Decoder. This phenomenon can be seen as the ASCII value of a capital G (71) is passed to the character decoder. A full clock cycle later, the output of state 17 (The first state in the string of states to match to 'gif') goes active.

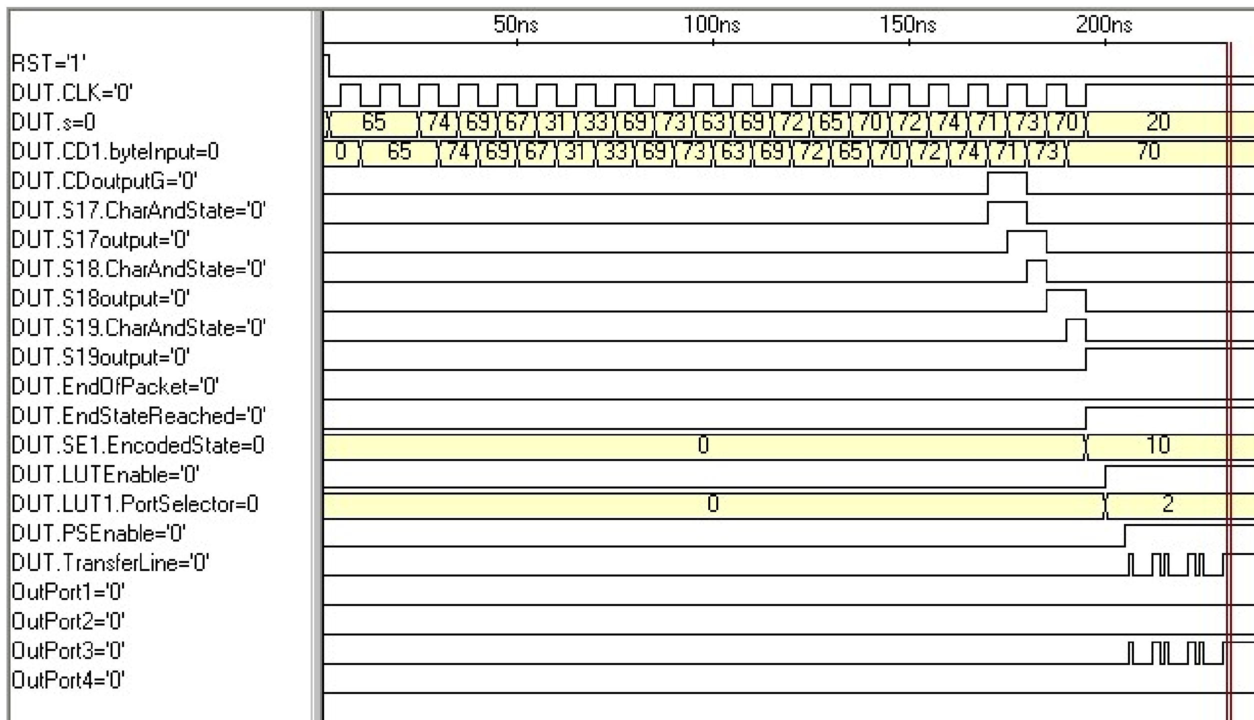

Figure 16 - Completed Simultation

Once a final state has been reached and the character for the last state has been matched, then the output to the final state is asserted. In Figure 16, state 19 is the final state in the string of states used to match the file format 'gif'. When the output of a final state is asserted, then the EndStateReached variable is asserted. This signal then kicks off the second portion of the circuit. Once a file format has been matched, now the appropriate port must be selected. First, the state is encoded by the State Encoder, as seen in the EncodedState variable. In Figure 16, the '10' is just

an enumeration used to represent that 'gif' has been matched. Next, the look-up table is enabled and the encoded state is used to find the appropriate port number in the table. The look-up table was pre-encoded with the appropriate file format to server matching. In Figure 16, the PortSelector variable is outputting a value of 2, signaling that the packet should be assigned to the third server. Next, the Port Selector is enabled and the packet information is sent to the Port Selector on a transfer line. This can be seen in Figure 16, as the PSEnable variable is asserted, then the packet is transferred out of OutPort3.

Synthesis

After the functional simulation was completed, the VHDL code was synthesized and laid-out using Altera's Quartus II software. Version 7.1 of the Web Edition of the Quartus II software allows for schematic or text-based HDL input. The software is capable of synthesizing a design and performing layout for many of Altera's FPGA boards. Altera releases the software to provide a strong platform for users to create designs and easily program the companies' FPGAs.

After the VHDL code for the Front End Device is imported into the Quartus II program, VHDL compilation can occur. The Quartus program creates a register transfer file (RTL) in the process of compilation. The RTL design can then be analyzed to verify a design's structure using the RTL Viewer. A design's hierarchy can be navigated, particular items can be found and items can be traced back directly to the source schematic or HDL file.

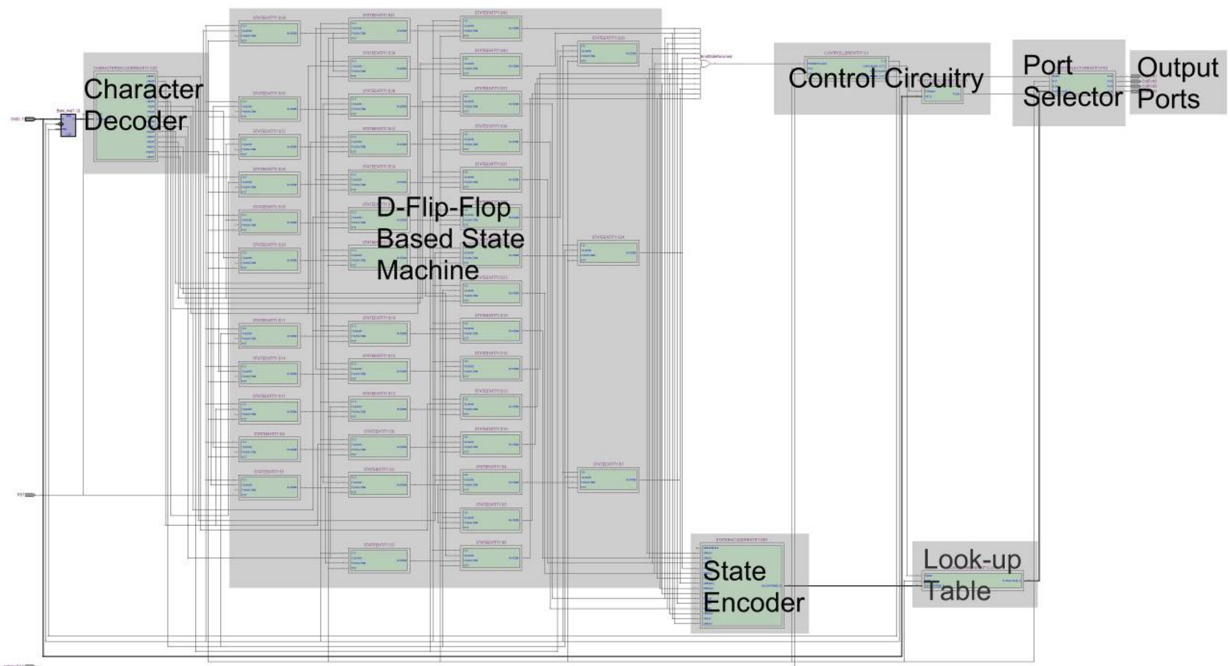

Figure 17 - Annotated RTL View

Figure 17 is an annotated schematic of the RTL for the proposed FED. All of the components from Figure 6 have been highlighted for comparison. In Figure 17, the individual states of the Aho-Corasick implementation can be seen. In fact, a resemblance can even be seen to Figure 9.

As part of the synthesis process, the Quartus software then translates the VHDL into hardware. This hardware can be represented as combinational logic, registers, or Logic Cells (LCs). These hardware components are available resources on the FPGA chosen as the target board. Table 2 lists the total number of FPGA elements used per VHDL entity, and the number of elements that were implemented at the stated level of implementation. For instance, the FEDEntity block implements all of the VHDL entities, so it consumes 115 LCs. However, because the FEDEntity only implements other VHDL entities, it does not require any logic, only

registers. This is denoted by the seven in parenthesis following the number 115. However, referencing the Register Only LCs column shows that the seven LCs consumed by the FEDEntity were simply for the use of registers. An interesting point to notice is that there are two different implementations for the 42 StateEntities. The first version represented by 'StateEntity – State 1' in the table does not require the standard AND logic preceding the D-Flip-Flop, as in Figure 8. This is because the state is the first state in a matching string. So, there is no input to denote that the previous state is active. Since the D-Flip-Flop requires an ALUT itself, the total FPGA resource consumption for this version of the StateEntity is a single ALUT. However, the typical StateEntity does contain internal logic as well as a D-Flip-Flop, resulting in the 'StateEntity – State 2' VHDL entity in 2. This entry shows that 2 ALUTs are used, one for the D-Flip-Flop and one for the internal logic.

Table 2 - FPGA Resource Usage Per VHDL Entity

VHDL Logic Block	Logic Cells	LUT only LCs	Register only LCs
FEDENTITY (Total)	115 (7)	108 (0)	7 (7)
ControllerEntity	3 (3)	18 (18)	0 (0)
CharacterDecoderEntity	18 (18)	3 (3)	0 (0)
LookUpTableEntity	2 (2)	2 (2)	0 (0)
PortSelectorEntity	4 (4)	4 (4)	0 (0)
StateEncoderEntity	9 (9)	9 (9)	0 (0)
StateEntity – State 1	1 (0)	2 (1)	0 (0)
State 1: DFFEntity	1 (1)	1 (1)	0 (0)
StateEntity – State 2	2 (1)	1 (0)	0 (0)
State 2: DFFEntity	1 (1)	1 (1)	0 (0)

Layout

Once, the Quartus software has completed synthesis, all of the hardware required to completely implement the circuit on the FPGA is known. So, the next step is to layout the pieces, or in the case that the circuit is going to be implemented on a FPGA, to assign the required resources to the available resources on the FPGA. For the sample circuit, the layout process was simplified because of the relatively small size of the circuit. For example, a model EMP240F100I5 from the Max II FPGA family from Altera contains 240 Logic Elements. Referring to Table 2, the sample circuit implemented requires roughly 50% of the available resources of the FPGA. As a result, the layout in Figure 18 appears relatively sparse. In Figure 18, the shade of the cell shows the utilization of the FPGA. The diagram can be broken down into two regions. The border cells represent the I/O pins of the FPGA while the interior cells represent the utilization of the FPGA's resources. On the perimeter, higher utilization is represented by darker browns. On the interior, higher utilization is represented by darker blues. Because of the relatively small size of the implemented circuit, only a small amount of the FPGA's total resources are required.

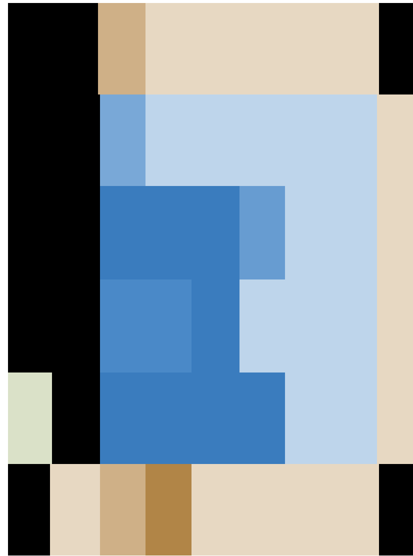

Figure 18 – FPGA Utilization For The Proposed FED

Once layout has been performed, a more detailed timing analysis can be performed. The reason why detailed timing analysis must wait until after layout is complete is because the length of the interconnect (wire) between nodes must be known to provide an accurate delay across the circuit. Table 3 contains the detailed timing analysis of the proposed FED. The analysis includes the worst-case set-up time (tsu), clock-to-output delay (tco), intrinsic [pin-to-pin] delay (tpd), and hold times (th) for the FPGA operating at 140 MHz.

Table 3 - Detailed Timing Analysis For Proposed FED

Worst-case tsu	0.106 ns
Worst-case tco	17.997 ns
Worst-case tpd	8.921 ns
Worst-case th	1.433 ns

Lastly, a power analysis can be performed once layout is complete. The power analysis requires layout to be complete because the amount of power dissipation of a wire is dependant on the wires length. So, similar to the timing analysis, the power analysis must wait until the layout has been performed. With a toggle rate of 50% on the input I/O pins, the proposed FED design will consume 61.48 mW. This is the sum of the Core Dynamic Thermal Power Dissipation of 9.97 mW, Core Static Thermal Power Dissipation of 39.61 mW and the I/O Thermal Power Dissipation of 11.91 mW.

CHAPTER FIVE:
CONCLUSIONS AND FUTURE WORK

Modern web services and large websites are moving towards distributing the amount of incoming traffic among multiple servers. However, as the speed and amount of internet traffic increase, a single point of failure is the distribution node. As the rate of growth of network speed overtakes the rate of growth of CMOS speed, software solutions which have to process the software application as well as packing and unpacking through the TCP/IP stack begin to fail under the increased network load. To increase the front-end device's ability to handle ever increasing workloads, this paper proposed migrating the FED to a hardware only architecture. A VHDL implementation of a FED for hardware level string matching has been constructed. This device would make an ideal component or add-in to a router or any other front-end device with other features such as security. The proposed FED was synthesized for a MAX II FPGA from Quartus. A synthesized operational frequency of 140 MHz with a power consumption of 61 mW was achieved.

Future work for this project would involve more fully integrating the FPGA with the design of popular routers. This would involve accessing the data path of the router, allowing the network processors to send data to the FGPA chip. Also, a mechanism for TCP hand-off could be implemented. By handing off the TCP connection, stress would be relieved from the router.

LIST OF REFERENCES

[1] A. Aho and M. Corasick. Efficient String Matching: An Aid to Bibliographic Search, *Communications of the ACM*, 18(6):333-340, 1975.

[2] S. Fide and S. Jenks. A Survey of String Matching Approaches in Hardware, Technical Report, UC-Irvine, 2006.

[3] M. Aldwairi, T. Conte, and P. Franzon. 2005. Configurable string matching hardware for speeding up intrusion detection. *SIGARCH Comput. Archit. News* 33, 1 (Mar. 2005), 99-107.

[4] B. L. Hutchings, R. Franklin, and D. Carver. Assisting Network Intrusion Detection with Reconfigurable Hardware. *Proceedings of Field-Programmable Custom Computing Machines*, 2002.

[5] C. R. Clark and D. E. Schimmel. Scalable Parallel Pattern-Matching on High-Speed Networks. *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2004.

[6] J. W. Lockwood, C. Neely, C. Zuver, J. Moscola, S. Dharmapurikar, and D. Lim. An extensible, systemon-programmable-chip, content-aware Internet firewall. *Proc. of the Field Programmable Logic and Applications*, Lisbon, Portugal, Sept. 2003.

[7] M. Luo, C. Yang, and C. Tseng. Content management on server farm with layer-7 routing. *Proc. of the 2002 ACM Symposium on Applied Computing*, Madrid, Spain, March 11 - 14, 2002.

[8] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. *Proc. of the 2000 USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[9] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. *SIGPLAN*, no. 33, 11; Nov. 1998.

[10] E. A. Brewer. (2001, Jul/Aug) Lessons from giant-scale services. *IEEE Internet Computing* [Online]. 5(4). pp. 46-55 Available: http://ieeexplore.ieee.org.ucfproxy.fcla.edu/iel5/4236/20329/00939450.pdf

[11] G. Apostolopoulus, D. Aubespin, V. Peris, P. Pradhan, D. Saha. (2000, Mar) Design, Implimentation and Performance of a Content-Based Switch. *INFOCOM 2000* [Online]. 3(3) pp. 1117-1126 Available: http://ieeexplore.ieee.org.ucfproxy.fcla.edu/iel5/6725/17999/00832470.pdf

[12] R.S. Boyer and J.S. Moore, A fast string searching algorithm. *Carom. ACM* 20 (10), 262-272, 1977.

[13] T. Kocak and F. Basci. A power-efficient TCAM architecture for network forwarding tables. *Journal of Systems Architecture*, 52 (5), 307-314, 2006.

[14] T. Kocak and I. Kaya. Low-power bloom filter architecture for deep packet inspection. *IEEE Communications Letters*, 10 (3), 210-212, 2006.

[15] J.F. Kurose, K.W. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*, 3rd ed., Patparganj, Delhi, India: Pearson Education, 2005.

[16] G. Nutt, *Operating Systems*, 3rd ed., Patparganj, Delhi, India: Pearson Education, 2004, pp. 615-661.

[17] D. Comer, *Computer Networks and Internets: with Internet ApplicationsI*, 4th ed., Patparganj, Delhi, India: Pearson Education, 2005.

[18] N. Pope, Standards in Commercial Security. *Designing Secure Systems, IEE Colloquium on*, Jun. 1992

[19] *ISO in Brief*, International Organization for Standardization, Genève 20, Switzerland, 2006. Available: http://www.iso.org/iso/en/prods-services/otherpubs/pdf/isoinbrief_2006-en.pdf

[20] H. Altunbasak, S. Krasser, H. Owen, J. Sokol, J. Grimminger. Addressing the weak link between layer 2 and layer 3 in the Internet architecture. *Local Computer Networks, 2004. 29th Annual IEEE International Conference on,* Nov. 2004. pp. 417-418.

[21] S. H. Kim, H. C. Kim, H. Kang. The ARP cache structure of LAN emulation server in ATM-LAN system. TENCON 99. Proceedings of the IEEE Region 10 Conference, Sept. 1999. pp. 218-221.

[22] C. Smythe, *Internetworking: Designing the Right Architectures,* Workingham, England. Addison-Wesley Publishing Company, 1995.

[23] M. Hofmann, L. Beaumont, *Content Networkin: Architecture, Protols, and Practice,* Amsterdam. Morgan Kauffmann Publishers, 2005.

[24] S. Ros, *Content Networking Fundamentals,* Indianapolis, In. Cisco Press, 2006.

[25] C. Shiflett, *HTTP: Developer's Handbook,* Indianapolis, In. Sams Publishing, 2003.