A MULTI-GIGABIT NETWORK PACKET INSPECTION AND ANALYSIS ARCHITECTURE FOR INTRUSION DETECTION AND PREVENTION UTILIZING PIPELINING AND CONTENT-ADDRESSABLE MEMORY

by

Jacob J. Repanshek

BS, University of Pittsburgh, 2003

Submitted to the Graduate Faculty of the

School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science in Electrical Engineering

University of Pittsburgh

2004

UNIVERSITY OF PITTSBURGH

SCHOOL OF ENGINEERING

This thesis was presented

by

Jacob J. Repanshek

It was defended on

September 9, 2004

James T. Cain, Professor, Electrical Engineering Department

Alex Jones, Assistant Professor, Electrical Engineering Department

Thesis Advisor: Raymond R. Hoare, Assistant Professor, Electrical Engineering Department

A MULTI-GIGABIT NETWORK PACKET INSPECTION AND ANALYSIS ARCHITECTURE FOR INTRUSION DETECTION AND PREVENTION UTILIZING PIPELINING AND CONTENT-ADDRESSABLE MEMORY

Jacob J. Repanshek, MS

University of Pittsburgh, 2004

Increases in network traffic volume and transmission speeds have given rise to the need for extremely fast packet processing. Many traditional processor-based network devices are no longer sufficient to handle tasks such as packet analysis and intrusion detection at multi-Gigabit rates. This thesis proposes two novel pipelined hardware architectures to relieve the computational load of a processor within network switches and routers. First, the Embedded Protocol Analyzer Pre-Processor (ePAPP) is capable of taking an unclassified packet byte stream directly off of a network cable at line speed and separating the data into individually classified protocol fields. Second, the CAM-Assisted Signature-Matching Architecture (CASMA) uses ternary content-addressable memory to perform the task of stateless intrusion detection signature-matching. The Snort open-source software network intrusion detection system is used as a model for intrusion detection functionality. Structured ASIC synthesis results show that ePAPP supports speeds of 2.89 Gb/s using less than 1% of available logic cells. CASMA is shown to support 1.25 Gb/s using less than 6% of available logic cells. The CASMA architecture is demonstrated to be able to implement 1729 of 1993 or 86.8% of the attack signatures, or rules, packaged with Snort version 2.1.2.

TABLE OF CONTENTS

1.0 INTRODUCTION 1

1.1 IMPORTANCE OF CYBER SECURITY 2

1.2 EFFECTS OF INCREASES IN NETWORK TRAFFIC AND TRANSMISSION SPEEDS 3

1.3 THE NEED TO REDESIGN NETWORK SWITCHES TO FACILITATE INTRUSION DETECTION 5

1.4 PACKET ANALYSIS 7

1.5 USING CONTENT-ADDRESSABLE MEMORY TO ASSIST IN INTRUSION DETECTION 8

1.6 OVERVIEW OF ARCHITECTURE 9

1.6.1 ePAPP: An Embedded Protocol Analyzer Pre-Processor 10

1.6.2 A CAM-Assisted Signature-Matching Architecture 12

2.0 RELATED WORK 17

3.0 ePAPP: AN EMBEDDED PROTOCOL ANALYZER PRE-PROCESSOR 20

3.1 INTRODUCTION 20

3.2 ARCHITECTURE FOR PROTOCOL MAPPING 22

3.2.1 Protocol Memory 24

3.2.2 Jump Register and Jump TLB 28

3.2.3 Length Block 30

3.3 CIRCUIT IMPLEMENTATION OF EPAPP 30
3.4 DESIGN RESULTS AND PERFORMANCE 35
3.5 CONCLUSIONS 37
4.0 CASMA: A CAM-ASSISTED SIGNATURE-MATCHING ARCHITECTURE FOR INTRUSION DETECTION 39
4.1 THE SNORT INTRUSION DETECTION SYSTEM 40
4.1.1 Snort Packet Header Rule Options 42
4.1.2 Snort Packet Payload Rule Options 45
4.1.3 Snort Post-Match Rule Options 49
4.1.4 Summary of Snort Rules 50
4.2 TERNARY CONTENT-ADDRESSABLE MEMORY 52
4.3 PACKET HEADER SEARCHING 54
4.4 PACKET PAYLOAD SEARCHING 57
4.5 SNORT RULE ENCODING EXAMPLES 65
4.5.1 Snort Rule Example #1 66
4.5.2 Snort Rule Example #2 67
4.5.3 Snort Rule Example #3 69
4.5.4 Snort Rule Exceptions 71
4.6 THE CASMA ARCHITECTURE 71
4.6.1 CASMA Data Flow 72
4.6.2 CASMA Timing 76
4.6.3 CASMA Testing Methodology 85
4.6.4 CASMA Technology Mapping Results 85

5.0 CONCLUSIONS AND FUTURE DIRECTIONS 87

APPENDIX A 90

VHDL CODE FOR THE ANALYZER_TOP ENTITY 90

APPENDIX B 100

VHDL CODE FOR THE ASSEMBLER ENTITY 100

APPENDIX C 107

VHDL CODE FOR THE JUMP_TLB ENTITY 107

APPENDIX D 111

VHDL CODE FOR THE LENGTH_BLOCK ENTITY 111

APPENDIX E 117

VHDL CODE FOR THE PROTOCOL_MEMORY ENTITY 117

APPENDIX F 128

VHDL CODE FOR THE SNORT_CAM ENTITY 128

BIBLIOGRAPHY 170

LIST OF TABLES

Table 1. Time to Respond to a Network Packet at Various Peak Transmission Speeds 4

Table 2. IP Protocol Field Description 25

Table 3. UDP Protocol Field Description 25

Table 4. TCP Protocol Field Description 25

Table 5. The Protocol RAM Data 28

Table 6. The Jump Translation Look-Aside Buffer 29

Table 7. Current Content Inside Protocol Memory 32

Table 8. Performance Results After Synthesis 36

Table 9. Performance Comparison With Software 37

Table 10. Example Rule Header Table Format 42

Table 11. Rule Header Example #1 43

Table 12. Rule Header Example #2 43

Table 13. Rule Header Example #3 44

Table 14. Rule Header Example #4 44

Table 15. Snort Rule Header and Body Example #1 45

Table 16. Non-Payload Rule Options 46

Table 17. Payload Rule Options 47

Table 18. Snort Rule Header and Body Example #2 49

Table 19. Meta-data Rule Options 50

Table 20. Post-detection Rule Options 50

Table 21. Protocol Header Fields Included in a Header Search CAM Entry 55

Table 22. Effects on Time and Space Requirement for Varying Payload Shift Amounts............ 59

Table 23. Test Packet #1... 77

Table 24. Test Packet #2... 81

Table 25. Performance Results After Synthesis 85

LIST OF FIGURES

Figure 1. Typical Network Before and After Infection from a Malicious Email Attachment....... 1

Figure 2. Network Incorporating Routers and Switches with Packet Inspection Capabilities....... 2

Figure 3. Overview of the Presented Architecture.. 10

Figure 4. Traditional Network Switch Architecture 15

Figure 5. Network Switch Augmented with the ePAPP and CASMA Architectures 16

Figure 6. The Protocol Analyzer Pre-Processor 23

Figure 7. An Example of a Layered Protocol Description.. 26

Figure 8. The Internal Architecture of the Protocol Analyzer Pre-Processor.............................. 31

Figure 9. Simulation Waveform for ePAPP 34

Figure 10. Number of Occurrences of Each Option in the Entire 1,993-Rule Snort Ruleset....... 51

Figure 11. Example of a Search on a Populated CAM... 53

Figure 12. Shift Amount Between Searches vs. CAM Entries Required 59

Figure 13. Position of CASMA in the Presented Switch Architecture.. 72

Figure 14. Internal Circuit Design of the CAM-Assisted Signature-Matching Architecture....... 74

Figure 15. Flow of the Controller State Machine 75

Figure 16. Header Searching Beginning for Test Packet #1.. 78

Figure 17. Payload Searching Beginning for Test Packet #1 79

Figure 18. Payload Searching Concluding for Test Packet #1 80

Figure 19. Header Searching Beginning for Test Packet #2.. 82

Figure 20. Payload Searching Beginning for Test Packet #2 83

Figure 21. Payload Searching Concluding for Test Packet #2 84

1.0 INTRODUCTION

There are few areas in the contemporary computing technology industry that have received as much attention as cyber security. Rapidly increasing network transmission speeds have marked the computationally heavy task of network packet inspection as a conspicuous bottleneck in the processing and forwarding of information across the network. The need for this function, however, cannot be ignored. Consider the typical local network pictured in Figure 1. Once an infected email attachment has entered the network, there is no protection in a typical switch or router to stop the spread of malicious behavior.

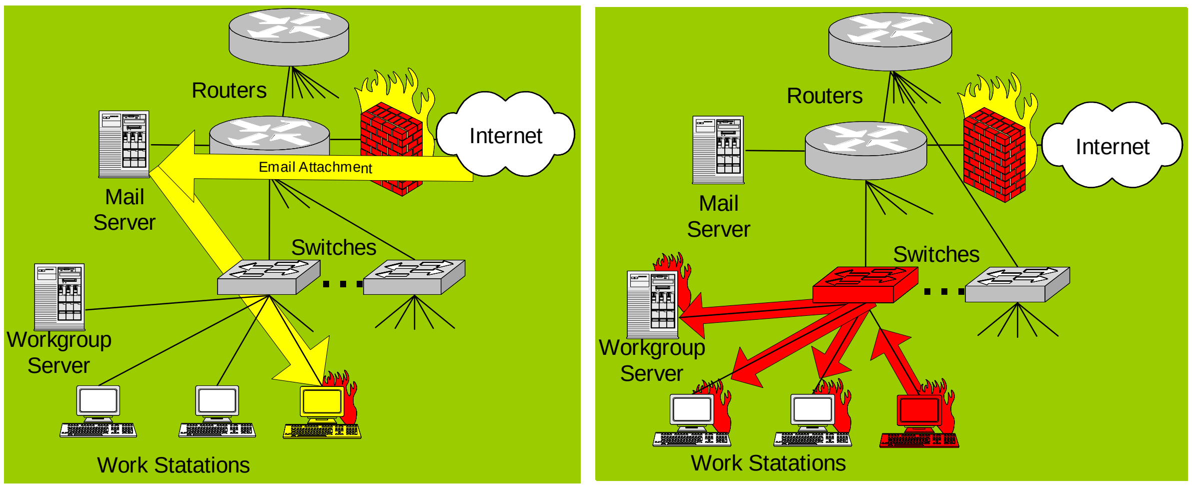


Figure 1. Typical Network Before and After Infection from a Malicious Email Attachment

Now consider the network in Figure 2. The routers and switches in this network have been fortified, which is to say they are capable of detecting and dropping malicious traffic. Even if a single host on the network becomes infected, in this example by an email attachment, the attack will not be allowed to spread.

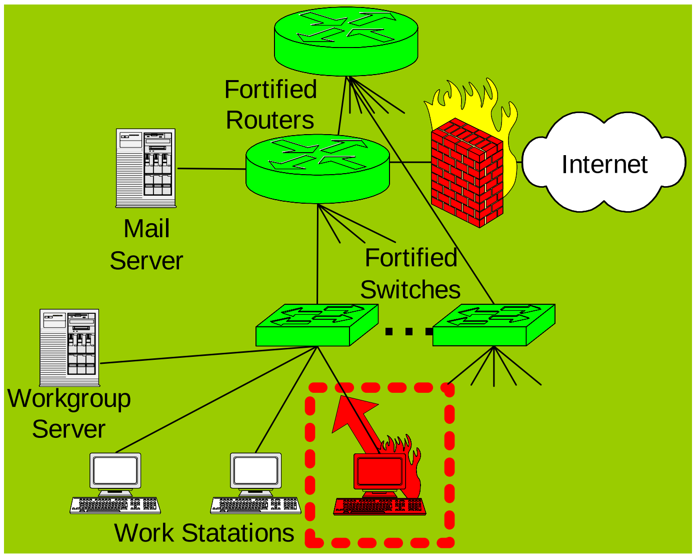


Figure 2. Network Incorporating Routers and Switches with Packet Inspection Capabilities

This research has developed a packet classification and intrusion detection methodology that will demonstrate performance advantages over comparable conventional solutions and allow for incorporation into high-speed network devices such as switches. This chapter clarifies the need for such a solution and introduces the challenges of cyber security, with particular respect to packet classification and intrusion detection. Snort [21], an open-source intrusion detection software solution, is introduced as a benchmark against which this original work is compared.

1.1 IMPORTANCE OF CYBER SECURITY

The most evident indication of the importance of cyber security is the staggering amount of money industry must devote to it. The market research firm IDC predicts that a total of $14 billion will be spent for companies and organizations to protect themselves from network attacks by 2005, an increase from $5 billion in 2000 [2]. Furthermore, IDC predicts that the Intrusion Detection market (including Vulnerability Assessment) alone will have total revenue of more than $1 billion in 2003, with a Compound Annual Growth Rate of 34% during 1999-2004 [3, 4, 5].

These numbers do not seem so unwieldy when one considers that a single malicious attack, the Code Red worm, caused $2.62 billion in losses worldwide [6]. In one poll of primarily large corporations and government agencies, 80% acknowledged financial loss due to computer breaches [7].

Network intrusion can be defined as “any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource” [8]. As the financial figures show, cyber intrusions have created a serious problem that continues to grow. In the past ten years, the technical skills required to launch a successful network attack have decreased significantly, while the sophistication of such attacks has grown [9, 10]. Despite these facts, companies continue to migrate important information and resources to the Internet for reasons of accessibility [9].

1.2 EFFECTS OF INCREASES IN NETWORK TRAFFIC AND TRANSMISSION SPEEDS

In order to ensure the security of a network, every packet sent or received must be considered potentially harmful until proven otherwise. But as network traffic loads become heavier and transmission speeds increase, it becomes more difficult to adequately inspect every bit of information that passes through a network.

The Internet, for example, has seen a spectacular annual growth factor of 4- to 10-times in traffic volumes [1]. Such rapid rates of growth have created a great deal of interest in expanding network transmission bandwidth and increasing transmission speed. As a result, 10 Mb/s networks have been replaced by 100 Mb/s, 1 Gb/s, and even 10 Gb/s networks. With the rise of optical technology, network speeds on the order of 1 Tb/s have been developed and are expected to eventually reach a theoretical limit of about 50 Tb/s [1]. While these developments are exciting for network technology, they have created in their wake another problem for modern networks.

As the speeds of local networks increase, the amount of time available for a network device (such as a switch) to respond to a single packet decreases. As shown in Table 1, a small 64-byte packet can arrive every 51,200 ns for a 10 Mb/s network under peak traffic loads. This provides ample time to respond to each packet. At 1 Gb/s, however, there are only 512 ns to respond and at 10 Gb/s there are only 51 ns to respond.

Table 1. Time to Respond to a Network Packet at Various Peak Transmission Speeds

Peak Network Transmission Speed	Max. # of 64 Byte Packets / Second	Time to Respond (Nanoseconds)		
		Total	Time Per Snort Rule, given 1700 rules	# of 5 ns Memory Accesses
10 Mb/s	19,531	51,200	30 ns	**6**
100 Mb/s	195,313	5,120	3 ns	**< 1**
Gb/s	1,953,125	512	.3 ns	**0**
10 Gb/s	19,531,250	51	.03 ns	**0**

Consider, now, that we wish to check each of the incoming packets to see if it matches any of an extensive set of attack signatures – patterns found within a packet that could indicate malicious traffic. Snort [8], for example, is a software-based intrusion detection system (IDS) that has over 1700 attack signatures, or rules, that must be compared against each incoming packet. Assuming a memory access takes 5 ns, the execution time per rule we wish to check our packet against drops to 30 ns per rule for 10 Mb/s and to 0.03 ns per rule for 10 Gb/s. Given that a 1 GHz processor requires 1 ns per instruction, this is quite infeasible.

1.3 THE NEED TO REDESIGN NETWORK SWITCHES TO FACILITATE INTRUSION DETECTION

There are some who mistakenly believe that cyber security can be achieved strictly by controlling what leaves and enters the private network. One popular solution is a firewall [11], which serves as a gateway between the private network and the outside world. By relying entirely on a gateway, however, a single point-of-failure is created. According to a CSI/FBI security report, 90% of attacks bypass firewalls [7]. Obviously, firewall protection alone is not sufficient. Without any additional protection within the network, one compromised machine could attack other hosts within the enterprise without deterrence.

Even if firewalls and other similar measures were enough to prevent all malicious traffic from entering the network, nothing prevents an attack that is launched from within. Information security surveys consistently report that more than half of all incidents are insider attacks. Many security professionals refer to the “80/20 Rule” to describe the relative probability that a problem was caused by insiders as opposed to outsiders [13]. Some form of internal protection must therefore be provided.

One proposed solution is to let each host tackle the task of intrusion detection individually. Host-based intrusion detection software can be installed on every machine on a network in order to inspect all traffic to and from that host [14]. To be effective, this solutions requires not only that every network device (including laptops and wireless devices) have the IDS software installed, but also that every device is configured properly and safe from being turned off or disabled by an authorized user. In a large enterprise network, the sheer volume of systems to be monitored may make this solution impractical. Furthermore, since only the individual machines and not the network itself are protected, any renegade laptop or wireless device that joins the network can be a source of malicious activity.

It is clear that leaving intrusion detection and prevention to the host is unwise, and that a gateway between the internal network and the outside world is not sufficient protection. There is one network device, however, that resides within the local network and sees all of the traffic traveling to and from a host: the network switch. In fact, many cases exist where malicious traffic within a network is seen only by the attack computer, the victim computer, and the switch that connects them. This makes the switch an ideal location to perform intrusion detection and intrusion prevention tasks.

Network switches, however, tend to see a lot of traffic, often at rapid transmission rates. Some switches attempt to use a monitoring port that sees an aggregate of traffic from all of the other ports on the switch [31]. This is a good solution as long as the aggregate switch bandwidth is less than that of the monitoring port. Under heavy loads, however, the monitoring port may be unable to keep up with all of the traffic coming through the switch. Consider, for example, a 48-port 100 Mb/s switch with a single Gb/s monitoring port. If the switch is only operating at 10% of maximum load, the monitoring port would see (48 x 100) x 0.1 = 480 Mb/s of traffic. This is no problem for the monitoring port. But if the load at the switch increased to 50%, the aggregate switch traffic would reach 2.4 Gb/s, much more than the monitoring port can handle.

1.4 PACKET ANALYSIS

Most network devices, such as switches, rely on network processors to perform packet classification and forwarding. The network processor market has emerged as the fastest growing segment of the microprocessor industry [12]. Most network processors (NPs) utilize multiple channel processors to perform packet inspection and data extraction for each network link, typically using 32- or 64-bit RISC architectures but in some instances using VLIW processors.

One of the most fundamental tasks these network processors perform is *protocol analysis*. This is the process by which individual fields within a packet are classified and the protocols within the packet are identified. RISC processors utilize, by definition, a small subset of instructions that, when combined in a program, can execute complex tasks. The problem is that packet processing requires a large amount of bit manipulation to extract particular data fields. For example, the source and destination of an IP packet are octets 13 through 16 and octets 17 through 20, respectively. For a RISC processor, these fields must be placed in a 32- or 64-bit register before processing can take place. To achieve this protocol analysis task by software can be quite complex just to extract a few fields from a packet header. Many proposed network processors exist which use specialized, non-RISC architectures to support multi-gigabit speeds [15, 16]. These solutions are still constrained by the cycle-rich nature of processor architectures.

Application-specific integrated circuits (ASICs) are used in many network nodes to improve packet-processing speeds, however they are rarely flexible enough for rapid adaptation to protocol or standards changes [1]. However, if one were able to generalize the hardware architecture so that protocol classification information could be periodically updated in a structure such as a ROM, use of ASIC designs would become more feasible.

1.5 USING CONTENT-ADDRESSABLE MEMORY TO ASSIST IN INTRUSION DETECTION

Significant research has been done on intrusion detection methodologies. Perhaps the most common approach is signature-based, which is centered on the assumption that intrusion attempts can be characterized by the comparison of user activities against a database of known attacks that lead to compromised system states [17]. Signature-based intrusion detection is the basis for Snort [8], the software IDS solution previously referenced in Table 1. In order to identify potentially harmful packets, Snort must search its ruleset to find any rules that match the packet under inspection. As previously discussed, faster networks and heavy traffic loads make this approach insufficient. Attempts have been made to speedup search times using hierarchical searching and faster pattern matching algorithms [18, 19], however improvements have not been significant enough to handle gigabit network rates.

One possible solution to the delays associated with linear rule searching is the use of ternary content-addressable memory. A content-addressable memory (CAM) chip is fundamentally different than a standard RAM. Whereas a standard RAM returns data based on an address given as input, a CAM is capable of returning one or more addresses where the given input search data can be found. A *ternary* CAM is even more beneficial in that it allows for mask bits to be specified so that only a specified portion of the data stored within the CAM needs to match the input data. Most importantly, CAM searches occur in a fixed amount of time, regardless of the amount of data within the CAM itself.

Consider, for example, a 9 Mb Network Search Engine provided by Integrated Device Technologies [20]. This device contains a ternary content-addressable memory (tCAM) that supports 16,000 576-bit entries, all of which can be searched in a fixed amount of cycles. To compare, suppose that 2000 patterns need to be searched and that each pattern is only 32-bits long. Current IDS systems are processor-based and utilize GHz high-performance Pentium processors. We will conservatively assume that all data is located within L1 cache at 1ns per access. Under such assumptions, the pattern matching would require 2000 memory accesses just to read the patterns. Thus, the time-per-packet is 2,000 ns. For a CAM, all 2000 patterns are searched in parallel and the total execution time is 20ns, which is 100 times faster. Now suppose that our patterns are 320 bits wide. This would require 10 times the number of memory references on the part of the processor, making the CAM 1,000 times faster. Clearly, content-addressable memory has a benefit to the pattern-matching nature of intrusion detection.

1.6 OVERVIEW OF ARCHITECTURE

This thesis provides a novel approach to the problems of packet classification and stateless signature-matching to enable intrusion detection. Figure 3 shows an overview of the architecture presented in this thesis. The ePAPP component performs the task of pipelined protocol analysis, while the CASMA component performs stateless intrusion detection signature matching.

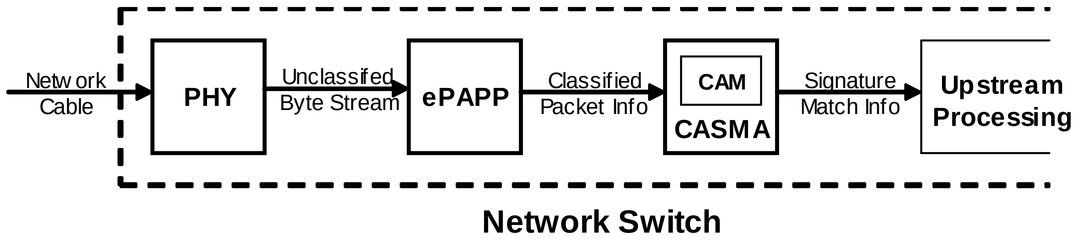


Figure 3. Overview of the Presented Architecture

1.6.1 ePAPP: An Embedded Protocol Analyzer Pre-Processor

To perform the task of packet classification through protocol analysis, the Embedded Protocol Analyzer Pre-Processor (ePAPP) is presented. ePAPP connects directly to the PHY interface, which is responsible for capturing bits off of the transmission line and outputting them in fixed-width chunks. A 100 Mb/s PHY, for example, produces a 4-bit datapath operating at 25 MHz, while a 1 Gb/s PHY produces an 8-bit datapath operating at 125 MHz. Since the goal of this work is to facilitate protocol analysis at a multi-gigabit switch, ePAPP is optimized to work with an 8-bit datapath. However, a simple 4-to-8-bit shift register can be used to convert a 4-bit stream operating at 25 MHz to an 8-bit, 12.5 Mhz stream, providing support for traditional Fast Ethernet networks.

The goal of ePAPP is to take the unclassified byte stream coming from the PHY and partition and classify the data blocks into corresponding protocol fields. These include header information fields such as source and destination addresses, header and payload sizes, and protocol flags, as well as the payload fields themselves. The partitioned data fields are output on a 32-bit bus, accompanied simultaneously by a unique 8-bit field type to identify not only the field’s context (its associated protocol) but also its particular relevance (e.g., source address, payload size, header checksum). Because not all protocol fields are exactly 32 bits in length, field data output is zero-padded as necessary to ensure valid output. Fields that may be larger than 32 bits in length, such as payloads, are divided and output over the necessary amount of 32-bit increments.

While many protocol field types have a fixed length associated with them, others such as payload and optional fields can be of variable lengths. The widths of these fields must be dynamically determined based on information within the protocol header itself. ePAPP is designed to detect and use relevant information within a packet header to calculate and apply these variable widths.

ePAPP utilizes a pipelined hardware architecture to achieve the aforementioned functionality in a fixed number of cycles. Hardware architectures confined to ASIC solutions, however, are typically very static and extremely cost-prohibitive to change. Reconfigurable FPGA solutions are not as inflexible to change, but due to size and cost are ill-suited to implementation in a network device. Therefore, a time-, cost-, and space-efficient protocol analysis solution in hardware must be adaptable to changes in protocol without frequent redesign. To solve this problem, ePAPP uses a "protocol memory" to enumerate possible protocol field types as well as information about the field types themselves. By placing this information in a loadable component, such as a ROM, the functionality of ePAPP can be modified to meet protocol changes without necessitating a complete hardware redesign.

The ePAPP design proposed in this thesis currently supports the following protocols: Ethernet (IEEE 802.3), Ipv4, ARP, TCP, and UDP. While these provided a sufficient base for functional testing and synthesis results, it should be noted that the generalized nature of the ePAPP architecture is expandable to include additional protocols (e.g., ICMP, IPv6) with little more than an addition to the contents of the ROM.

Since ePAPP classifies *every field in an incoming packet*, it is adaptable to virtually any application that requires protocol analysis. Its general, multi-use nature makes it quite cost-and space-efficient in a network device. One potential application for ePAPP processing is intrusion detection. Signature-based intrusion detection, in particular, relies on both a wealth of header information and on packet payloads. This thesis utilizes the ePAPP as a tool to present an associated intrusion detection architecture with the information it needs to perform content signature-matching.

1.6.2 A CAM-Assisted Signature-Matching Architecture

To perform the task of stateless intrusion detection, this thesis presents a CAM-Assisted Signature-Matching Architecture (CASMA). As previously discussed, the parallel-searching nature of content-addressable memory is a good fit to the search-intensive task of signature-matching. In order to establish the requirements of the CASMA architecture, we chose to use Snort as our model and performance benchmark [21]. As previously mentioned, Snort is open-source network intrusion detection software widely used and supported in industry. While newer versions of Snort are packaged with various pre-processors that provide additional services such as stateful matching, Snort's primary role is rule-based packet signature matching. Stateful inspection considers packets within a larger context that evaluates packets with respect to the current connection state. This requires the storage of connection state information, which is beyond the scope of this work. CASMA has been developed as an attempt to replicate the stateless signature matching capability of Snort in a pipelined hardware solution.

Snort rules are comprised of a number of different elements, including the type of traffic the rule pertains to (IP, TCP, UDP, ICMP), the source and destination addresses and ports for which the rule is valid, and a bevy of rule options (rules and rule options are discussed in detail in Chapter 4). Furthermore, each of these rule options can be classified as pertaining to some characteristic of either packet header information or packet payload information. This gives way to a natural delineation between “header rules” and “payload rules,” which is exploited by CASMA. While Snort handles both of these types in a single rule, the proposed design will address header rules and payload rules separately.

Separating header searches from payload searches has several advantages. First, since header information will arrive from ePAPP before the payload, header searching can be initiated while the packet payload is still being buffered. Secondly, combining header and payload information into a single search string can make the size of the string impractical for use in a CAM.

Matching header information is generally a simpler process than payload content matching. If header information in an incoming packet meets all of the header requirements of a given Snort rule (for instance, the source address and port number match specific values), the packet is tagged as a possible hit (contingent upon a corresponding payload match). For most rules, it is this simple. There are, however, some exceptions that do not merely require an exact match. Some rules require that a value fall within a certain range (for instance, the source port must be less than 1024). Other rules specify that a piece of header information must *not* be a particular value (for instance, the destination port cannot be 80). Presently, ternary CAMs do not have the ability to do range or inequality checking. Therefore, each of these exceptional header search requirements are performed with hardware preprocessing and incorporated into the header search string as a 1-bit flag.

Payload searches are different from header searches in that, instead of trying to simultaneously match a number of header field values in a single combined search, the payload search is attempting to match certain strings that appear at certain locations within a packet. Since the entire payload must be searched in the CAM, and since many payloads are larger than the maximum CAM width, payloads must be incrementally shifted and searched at pre-defined offsets. For reasons discussed in Chapter 4, CASMA performs a CAM search at each 8-byte offset within a payload. Therefore, a payload of *n* bytes will require ((n-1) / 8) + 1 CAM searches in order to sufficiently check a payload for all possible payload signature matches. Fortunately, a payload of *n* bytes requires *n* cycles to arrive from the ePAPP unit.

Certain rules specify that a payload signature to be matched be regarded as a hit only if the signature string is found at a certain range of depths within the payload. In order to incorporate this functionality into a standard CAM search, a novel encoding scheme is proposed that specifies within a CAM entry at which depths payload string matches are valid. By inputting into the CAM as part of the search string the depth associated with the payload content being searched, payload string depth requirements can be enforced.

Results from associated header and payload searches must be correlated by a post-processor to determine any and all rule matches, as well as determine the threat severity of the detected traffic. The specific nature of this correlation is beyond the scope of this research, however generic result outputs are provided to more easily enable rapid processing.

Figure 4 shows the general architecture of a typical switch. Packets are captured by the PHY and undergo MAC-layer processing. Packets are then passed on to a network processor that performs any processing that is required on the packet, which may include some form of packet inspection (e.g., packet filtering, intrusion detection).

Figure 5 shows a network switch that has been augmented with the ePAPP and CASMA architectures. ePAPP accepts unclassified bytes of packet data directly from the PHY and classifies them through protocol analysis. The formatted packet fields are forwarded to the CASMA unit, which performs header and payload CAM searches to find potential attack signature matches. Result data is passed to the network processor to be correlated. Thus, the entire burden of packet inspection is removed from the network processor through the addition of pipelined hardware.

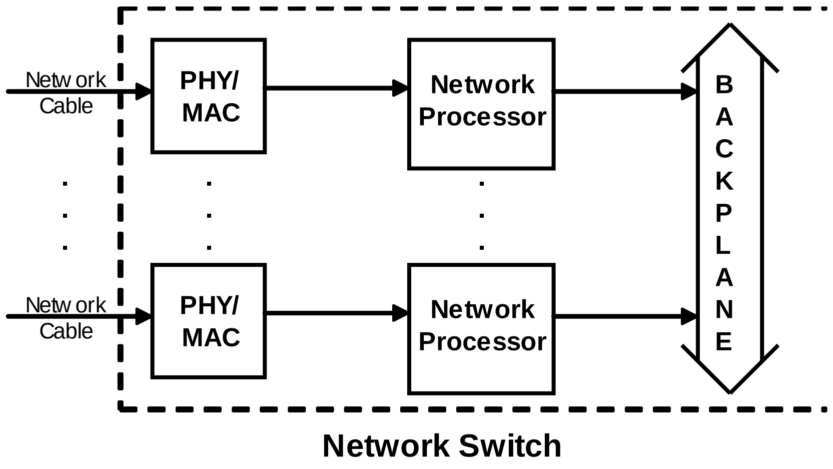


Figure 4. Traditional Network Switch Architecture

Chapter Two will examine research related to the goals of this thesis. Chapter Three will discuss the design and performance of the Embedded Protocol Analyzer Pre-Processor. Chapter Four will discuss the design, performance, and issues related to the CAM-Assisted Signature-Matching Architecture. Chapter Five will examine conclusions and future directions for this research.

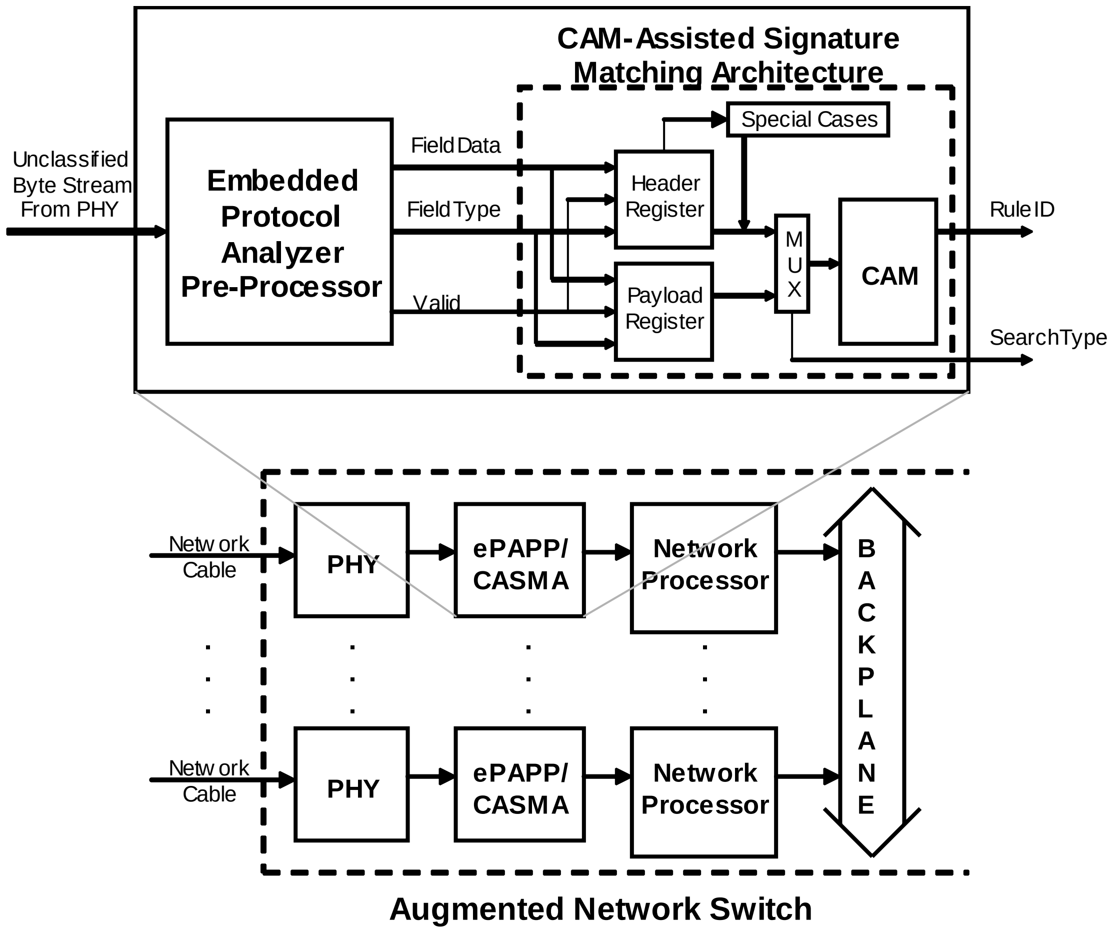


Figure 5. Network Switch Augmented with the ePAPP and CASMA Architectures

2.0 RELATED WORK

A hardware-based TCP/IP content scanning system is proposed in [22]. This work combines protocol processing engine, a per-flow state store, and content scanning engine into a single hardware architecture. Of primary interest to this research is the content scanning engine. Incoming TCP packets are streamed through the content scanner, a hardware model that is capable of scanning the payload of packets for a set of regular expressions using Deterministic Finite Automata (DFAs), as proposed in [23]. The content scanner claims speeds of up to 2.5 Gb/s. This design, however, supports only TCP flows, which is quite incomplete for intrusion detection purposes. Also, the design is targeted to an FPGA, which is cost- and size-prohibitive for use in a network device.

The work in [24] uses JHDL, a Java-based programmatic structural design tool, to create a module capable of generating circuits that match arbitrarily large regular expressions. The overall goal of this work is to prove the feasibility of using such a module on an FPGA to accelerate the task of string matching for network security applications. The work is based on the groundbreaking efforts on FPGA-based regular expression searching by Sidhu and Prasanna [25]. Search data is streamed through the match circuit one character at a time. Testing results indicate a throughput of one search character per clock cycle using the proposed method, irrespective of the length of the search string. This solution, as before, is impractical because of its dependence on a reconfigurable FPGA platform.

The method proposed in [26] attempts to implement the Snort intrusion detection engine in reconfigurable hardware on an FPGA. The research is based on an early version of Snort (v.1.8.7) that supported 1239 rules. The proposed architecture performs first a header search based on source and destination ports and addresses as well as protocol type, and then a subsequent payload search based on the results of the header search. The proposal suggests the use of CAMs to perform the necessary header and payload searches. While the work proposes many of the same fundamental concepts as this thesis, there are no design details or results available to support a proof of concept. Again, the choice of an FPGA as a target for the architecture is an impractical choice.

The architecture proposed in [28] is another attempt to map Snort rules directly into reconfigurable hardware. The design features content pattern-matching engines that match payload chunks in 4 byte increments against a given rule. Each rule, represented by its own generated structural VHDL, is searched in parallel. Initially, packet header information (source and destination ports and addresses, protocol type) is checked against the header requirements corresponding to a given rule. If there is a match, 4 byte shifts of payload information are fed into the rule's individual content pattern-matching engine. If a match is found, the corresponding packet is flagged and potentially dropped. Only 105 Snort rules are implemented in this design, and no explanation is given how these rules are mapped to VHDL. By its own admission, this contribution relies on frequent reconfiguration to account for new rules, hence its FPGA target. This suffers from the same drawbacks as much of the previously mentioned research.

The goal of [29] is to speed up multi-field packet classification using CAM-like memory. Citing the expensive nature of high-density ternary CAMs, 128 bit-wide ternary CAM-like memories (CLMs) are proposed which function like individual CAM entries but are touted as more cost-effective. The research focuses on classification of IPv6 packets based on IP source and destination addresses, source and destination ports, and protocol type. Each of the five field types is searched individually in parallel with the other field types, and results are correlated with specialized combinational logic to produce a rule match index uniquely identifying a rule hit. The proposed architecture has elements in common with this thesis, however its restriction to IPv6 packet classification does not satisfy the larger scope of the thesis effort.

The Granidt (Gigabit Rate Network Intrusion Detection Technology) architecture is proposed in [27]. This work proposes an integrated hardware/software solution that improves Snort performance by performing rule matching in custom hardware using CAMs. The software "rule compiler" is responsible for accepting a subset of Snort rule syntax and producing a hardware representation of the rule fields to be matched. The rule compiler creates several tables that are used to initialize the search CAMs, as well as an internal representation of the rules that links fields specified by the rules to the CAM tables and range tables. The "rule processor" software component initializes the hardware CAMs and initiates packet processing. Header and payload searches are performed separately, each facilitating a number of individual CAMs. A match vector that indicates the results of the CAM searches is correlated to the internal rule database to determine the appropriate action for a matched rule. Since the rule compiler is in software, new rules can be added to the design without resynthesis. This design is flawed, however, in that it requires several individual CAMs, which is cost-prohibitive. Also, the maximum supported signature size is 20 bytes, which is not sufficient for a large number of Snort rules. Details of the mapping of the rules themselves into hardware are not clearly defined.

3.0 ePAPP: AN EMBEDDED PROTOCOL ANALYZER PRE-PROCESSOR

This chapter presents a hardware Embedded Protocol Analyzer Pre-Processor (ePAPP) that performs the protocol analysis of network packets at line speeds of at least 1 Gb/s using only a small amount of area on a structured ASIC technology. The device fits between the physical network interface and an upstream processor or additional hardware pipeline (CASMA) and replaces the software protocol analysis program typically run on a processor, achieving a significant performance increase. The presented solution is additionally advantageous in that every protocol field of a network packet is classified. The particular protocols being analyzed can be configured in an internal memory within the pre-processor, allowing easy protocol upgrades and product versatility in different applications. A single configuration in the protocol memory can handle hundreds of protocols without reprogramming. A prototype of ePAPP that supports various protocols including Ethernet, IPv4, ARP, TCP, and UDP has been designed in VHDL and synthesized for a 130 nm a structured ASIC technology. Results show that 2.89 Gb/s can be achieved when implemented on a structured ASIC, using less than 1% of available logic cells. The prototype is also demonstrated to have a decoding latency 75 times faster than the conventional software.

3.1 INTRODUCTION

As the speeds of local networks increase, the amount of time available for a network device (such as a switch) to respond to a single packet decreases. To improve upon traditional software solutions, network processors have been developed which are capable of performing packet classification and forwarding at higher speeds. Most network processors (NPs) utilize multiple channel processors to perform packet inspection and data extraction for each network link, typically using 32- or 64-bit RISC architectures but in some instances using VLIW processors.

One of the most fundamental tasks these network processors perform is *protocol analysis*. This is the process by which individual fields within a packet are classified and the protocols within the packet are identified. RISC processors are by definition a small subset of instructions that, when combined in a program, can execute complex tasks. The problem is that packet processing requires a large amount of bit manipulation to extract particular data fields. For example, the source and destination of an IP packet are octets 13 through 16 and octets 17 through 20, respectively. For a RISC processor, these fields must be placed in a 32- or 64-bit register before processing can take place. To achieve this protocol analysis task by software can be quite complex just to extract a few fields from a packet header.

The physical layer interface (PHY) to a network cable translates analog signal levels into a stream of fixed width digital data. For example, the interface to a 100 Mb/s PHY is a 4-bit wide data path that operates at 25 MHz and a 1 Gb/s PHY is an 8-bit wide data path that operates at 125 MHz. This thesis presents a hardware Embedded Protocol Analyzer Pre-Processor (ePAPP) that resides between the PHY and the channel processor to replace the protocol analysis work that is usually done by software in the processor, with a great performance improvement over software solutions. The objective is to assemble and output the individual protocol fields of a packet at line speeds, augmenting the field data output with a field type that is unique to the particular protocol and field description. Moving the protocol analysis work from software to the hardware ePAPP drastically reduces the execution time for protocol analysis, as well as removes software overhead in the Channel Processor.

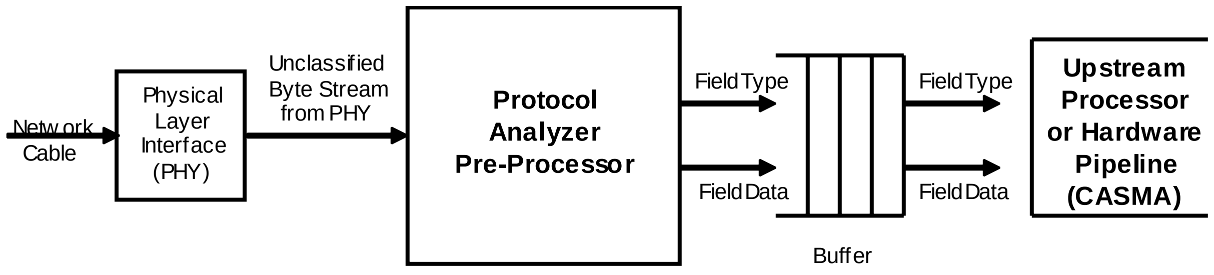
Many proposed network processors exist which use specialized, non-RISC architectures to support multi-gigabit speeds [15,16]. These solutions are still constrained by the cycle-rich nature of processor architectures. We propose a design that will be able to separate and classify *every field of a network packet* in a minimal number of cycles.

Application-specific integrated circuits (ASICs) are used in many network nodes to improve packet processing speeds, however they are rarely flexible enough for rapid adaptation to protocol or standards changes [1]. By utilizing a ROM to store protocol classification information, our proposed design allows for incorporation of new standards and designs without complete ASIC redesign.

In this chapter, Section 3.2 presents the reconfigurable architecture for mapping the protocols into hardware. Section 3.3 presents a circuit implementation of a prototype supporting protocols including Ethernet, IPv4, ARP, UDP, and TCP. Section 3.4 introduces performance of the design and performance benefits over conventional software.

3.2 ARCHITECTURE FOR PROTOCOL MAPPING

ePAPP is a hardware-based Protocol Analyzer that will receive a fixed width stream of packet data from the PHY and output a stream of organized data by forwarding the Field Type and Field Data values. As shown in Figure 6, values can be forwarded to an upstream network processor or another pipelined hardware unit, such as CASMA. The Field Type is a unique number that specifies a particular field of a particular protocol. It is used to classify the data that is simultaneously presented on the Field Data bus.



Network Cable
Physical Layer Interface (PHY)
Unclassified Byte Stream from PHY
Protocol Analyzer Pre-Processor
Field Type
Field Data
Buffer
Field Type
Field Data
Upstream Processor or Hardware Pipeline (CASMA)

Figure 6. The Protocol Analyzer Pre-Processor

The width of the Field Data bus is the same as the register widths inside the upstream processor. Field Data is gathered directly from the incoming stream of packet data and is zero-padded depending on the width of the associated protocol field. Thus, rather than seeing a stream of 4- or 8-bit values, an upstream processor would see a stream of packet field information. This would drastically reduce the amount of pre-processing an upstream processor would have to perform before it could determine the proper packet handling.

Field widths in data packets vary a great deal, and many fields will not consume the entire register width. This will not reduce the efficiency of the upstream processor as long as the average field width is larger than the fixed width stream coming off of the PHY. In most cases, the field width will be significantly larger and, in some cases, may need to be broken into two or more separate fields, as with packet payloads. This is beneficial to the upstream processor as it better fits its architecture.

Due to the continuing development of Internet protocols, we store the definition of protocol types in a Protocol Memory, which makes it easy for protocol updates, reconfiguration, or expansion.

Apart from the Protocol Memory, there are two other parts assisting with the protocol field classification: a “Jump Translation Look-aside Buffer (TLB)” with a “Jump Register” to handle branches between different layers of protocols according to the information found in packet headers; and a “Length Block” to get length information for variable length protocol fields (for example, a payload field) according to packet headers. The following sections describe in detail of how to map protocols into these three blocks.

3.2.1 Protocol Memory

The Protocol Memory, which stores the protocol field information used for classification, provides protocol information to the Protocol Analyzer. A unique number is assigned to each field type of each supported protocol. The current Internet protocols are very complex, involving multiple headers of dynamic lengths (one for each layer of abstraction) and demanding branches between encapsulated protocol layers.

An example using the IPv4, UDP, and TCP protocols is shown here. The associated protocol field descriptions are given in Tables 2 through 4, respectively. The Protocol Field (octet 10) of the Internet Protocol defines which protocol is used above the Network layer. Specifically, the Protocol Field of IPv4 defines the format of the Data fields within the IP packet, octets 21 and greater, which can either be the UDP format as shown in Table 3, or the TCP format showed in Table 4. Furthermore, octet 3 and 4 define the Total Length of the IP packet, from which can we derive the length of IP payload. The Internet Header Length can be gathered from the second nibble of octet 1, which will dictate the existence of optional octets 21 through 24. Thus, the data within a packet header determines how the packet is to be interpreted. This traditional layering scheme requires that the Protocol Analyzer have a decision tree to determine how it should interpret the stream of data.

Table 2. IP Protocol Field Description

OCTET POSITION	IP FIELD DESCRIPTION	ABREVIATION
OCTET 1	Version (4 bit)+IHL (4 bit)	(VER, IHL)
OCTET 2	Type of service	(TOS)
OCTET 3,4	Total Length	(TOL)
OCTET 5,6	Identification	(ID)
OCTET 7,8	Flags (3 bit)+Fragment Offset (13 bit)	(FLG, FRO)
OCTET 9	Time to Live	(TTL)
OCTET 10	Protocol	(PRO)
OCTET 11,12	Header Checksum	(IP_SUM)
OCTET 13,14,15,16	Source Address	(SRC)
OCTET 17,18,19,20	Destination Address	(DEST)
(OCTET 21,22,23,24)	(Options + Padding)	(OPT)
OCTET 21, 22…	Data	

Table 3. UDP Protocol Field Description

OCTET POSITION	UDP FIELD DESCRIPTION
OCTET 1,2	Source Port
OCTET 3,4	Destination Port
OCTET 5,6	Length
OCTET 7,8	Checksum
OCTET 9,10…..	Data

Table 4. TCP Protocol Field Description

OCTET POSITION	TCP FIELD DESCRIPTION	ABBREVIATION
OCTET 1,2	Source Port	(SRC_PORT)
OCTET 3,4	Destination Port	(DEST_PORT)
OCTET 5,6,7,8	Sequence Number	(SEQ)
OCTET 9,10,11,12	Acknowledgement Number	(ACK)
OCTET 13,14	Data Offset(4 bit)+ Reserved(6 bit)+ Control Flags(6 bit)	(DTO, FLG)
OCTET 15,16	Window	(WIN)
OCTET 17,18	Checksum	(TCP_SUM)
OCTET 19,20	Urgent Pointer	(URP)
(OCTET 21,22,23,24)	(Options + Padding)	(OPT)
OCTET 21,22…	Data	

For clarity and generality, we graph a packet containing layered protocol fields A through J below (Figure 7). In this packet, fields A and B are in a lower layer protocol and the value of B determines the protocol and the meaning of the data fields that follow. Figure 7 shows the three different protocol stacks that are possible. The first protocol layer contains the fields A and B. The contents of B determine which protocol is used at the next layer up the stack. In this example, the protocol CD is used if B=20 and protocol EF is used if B=40. However, in the EF protocol, the E field is used to determine the next protocol layer but the next protocol does not start until after F. Protocol GH is used if E is 71 and IJ is used if E is 72. The field values were randomly selected for this example. DT1, DT2, DT3 are payload data fields for protocols CD, GH, and IJ, respectively.

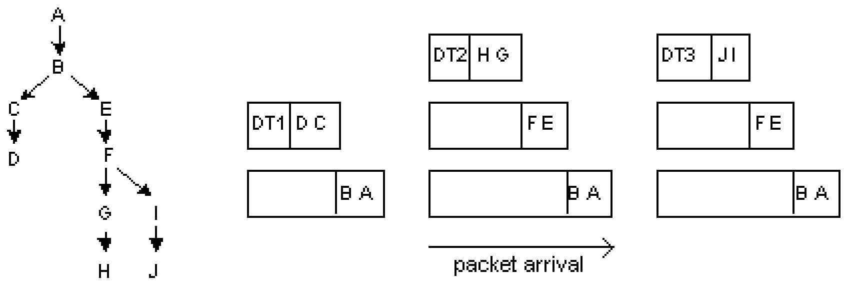


Figure 7. An Example of a Layered Protocol Description

Most of the protocol fields have constant widths that are defined by the protocol format and are labeled as W(A), W(B),…,W(DT1) in this example. Some other fields have a variable length that is calculated from a length field appearing earlier in the same packet. For example, DT2 and DT3 have variable lengths that depend on the values of fields G and J, which define the total length of protocol GH and protocol IJ, respectively.

The following characteristics of each protocol field type are stored in the Protocol Memory:

- *Field Type,* a unique number identifying the particular protocol field. No two fields inside the Protocol Memory may have the same Field Type, even if they are from different protocols. This simplifies the processing of the packets as the Field Type number also defines the context (i.e. protocol layer) in which the field exists.
- *Field Width*, defines the width of the field if it is of constant length.
- *Protocol Indicator,* a one-bit value that indicates whether the contents of the field are indicative of the next protocol layer to be used. If *Protocol Indicator* is set to '1', the field's contents are used by the "Jump TLB" to determine the initial Field Type of the next protocol. This value is stored in the "Jump Register" until a jump to the next protocol is indicated.
- *Branch Indicator,* a one-bit value that indicates whether there should be a branch to an upper layer protocol immediately following the current field. If the value of *Branch Indicator* is '0', the next Field Type used is the current Field Type incremented by one. If the value of *Branch Indicator* is '1', the next Field Type is taken from the "Jump Register," as stored by the "Jump TLB."
- *Length Indicator,* a one-bit value that indicates whether the current field contains any length information for an upcoming variable length protocol field. If the value of *Length Indicator* is '1', the field contains length-related information and should be passed to the "Length Block" to be used in the appropriate calculations.
- *Variable Length,* a one-bit value that indicates whether the current field is of variable length. If the value of *Variable Length* is '1', the field is of a variable length that must be retrieved from the "Length Block."

- *Packet Done,* a one-bit value that indicates whether the previous field was the last of the packet. If the value of *Packet Done* is '1', the Protocol Analyzer halts until a new packet is received from the PHY.

Table 5. The Protocol RAM Data

Protocol Field	Field Type	Field Width	Branch Indicator	Protocol Indicator	Length Indicator	Variable Length	Packet Done
A	0	W(A)	0	0	0	0	0
B	1	W(B)	1	1	0	0	0
C	2	W(C)	0	0	0	0	0
D	3	W(D)	0	0	0	0	0
DT1	4	W(DT1)	0	0	0	0	0
EOF1	5	0	0	0	0	0	1
E	6	W(E)	0	1	0	0	0
F	7	W(F)	1	0	0	0	0
G	8	W(G)	0	0	1	0	0
H	9	W(H)	0	0	0	0	0
DT2	10	0	0	0	0	1	0
EOF2	11	0	0	0	0	0	1
I	12	W(I)	0	0	0	0	0
J	13	W(J)	0	0	1	0	0
DT3	14	0	0	0	0	1	0
EOF3	15	0	0	0	0	0	1

Additional key fields may also be incorporated into this architecture. Table 5 shows the hypothetical Protocol RAM content for the example shown in Figure 7.

3.2.2 Jump Register and Jump TLB

The Jump Register and the Jump Translation Look-aside Buffer (TLB) handle the branches between different protocols. The Jump Register contains the Field Type of the next encapsulated protocol layer, as set when triggered by the "Protocol Indicator" field of the Protocol Memory. The Jump TLB is a look-up table that determines which upper layer protocol to use based on the current Field Type and Field Data when the "Protocol Indicator" signal is asserted. Table 6 shows the Jump TLB for the protocol example shown in Figure 7.

When the current Field Type and Field Data match a row in the TLB and Protocol Indicator is ‘1’, the value in the “Jump Address” column is stored in the Jump Register. This value is used for the next Field Type when “Branch Indicator” is set to ‘1’, effectively branching to a new protocol.

In the protocol example shown in Figure 7, the “B” field’s Protocol Memory entry has a value of ‘1’ in both its “Branch Indicator” and “Protocol Indicator” fields. Thus, the Field Type for B (1) and the current Field Data (either 20 or 40) are used to determine the Jump Address. If the Field Data is 20, the Jump Address is 2, which is the start of the CD protocol. If the Field Data is 40, however, the Jump Address is 6, which is the start of the EF protocol. Since protocol field B has a ‘1’ in its “Branch Indicator” field, the Jump Address is immediately used to indicate the next protocol after B.

For the EF protocol, the E field has a ‘1’ in its “Protocol Indicator” entry and thus is used to determine the next encapsulated protocol in the stack. The Field Data corresponding to protocol field E (either 71 or 72) is matched with the Field Type (6), to get either 8 or 12 as the Jump Address. This value is stored in the Jump Register until the completion of field F, which indicates a branch to the next protocol.

Table 6. The Jump Translation Look-Aside Buffer

Field Type	Field Data	Jump Address
1 (B)	20	2
1 (B)	40	6
6 (E)	71	8
6 (E)	72	12

3.2.3 Length Block

The Length Block is responsible for calculating and storing the lengths of variable length protocol fields. Such fields include optional fields, alignment padding, and payloads. The lengths of these fields are determined by utilizing length information contained in previous protocol fields. For instance, the specified header length for a protocol header may be used to determine how many option and padding bytes the header contains. In another case, the header length of a protocol layer may be subtracted from the total length of that layer to determine the number of payload bytes.

In the example in Table 5, field G defines the total length of the GH protocol layer, which includes field G, field H, and payload field DT2. Since field DT2 is not fixed in length, as indicted by a ‘1’ in its “Variable Length” entry, its length must be calculated in the Length Block. Knowing that G contains the total length of the GH layer, the width of the payload field DT2 is G – W(G) – W(H), where W(x) is the fixed width of protocol field x. This value is stored in the Length Register until the width of the payload field is requested.

3.3 CIRCUIT IMPLEMENTATION OF EPAPP

A prototype system supporting protocols Ethernet, IPv4, ARP, TCP, and UDP is implemented with the ability to process packets coming from a 1 Gb/s Ethernet PHY. Therefore, the input is a parallel stream of 8 bits from the 1 Gb/s PHY and the outputs are parallel streams of bits including an 8-bit line for the Field Type and a 32-bit line for the Field Data, as well as a single Valid bit to indicate valid output. The structure of the Protocol Analyzer Pre-Processor is shown in Figure 8, consisting of the Protocol Memory and Protocol Address Register, the Jump TLB, the Length Block, the Assembler, and a FIFO. The FIFO is at the beginning stage of ePAPP, connecting directly to the Ethernet PHY as a buffer to transfer the incoming data to the rest of ePAPP.

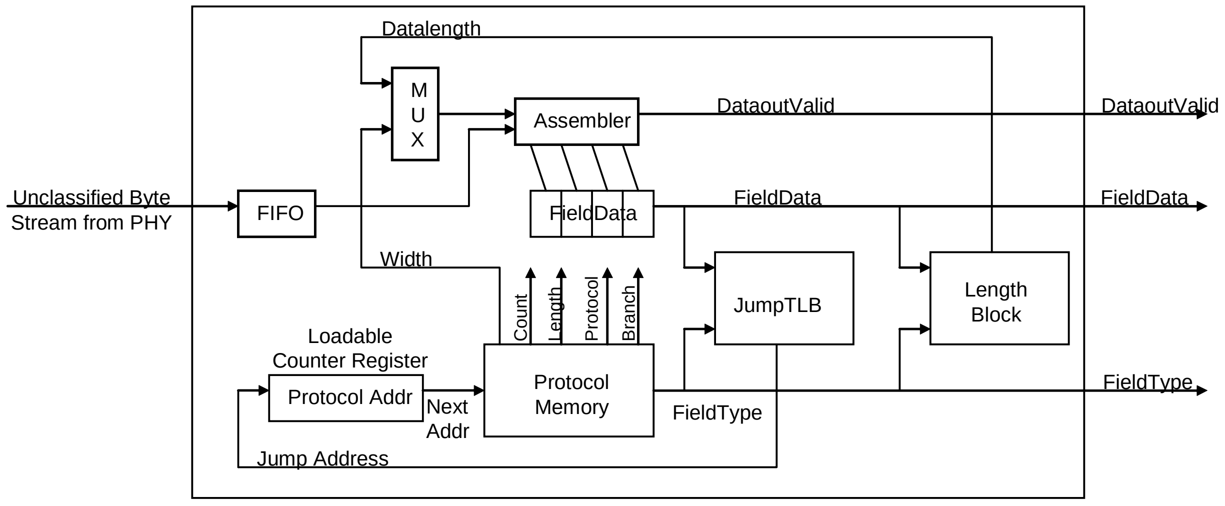


Figure 8. The Internal Architecture of the Protocol Analyzer Pre-Processor

The Assembler receives the parallel bit stream from the FIFO and places the bits into the correct location of the Field Data register. In essence, the Assembler is a shifter that has an internal counter loaded with the width of the current protocol field as indicated by the Protocol Memory. The contents of the Protocol Memory are shown in Table 7.

The shifter consists of four 8-bit registers to assemble the incoming blocks of 8 bits into 8-, 16-, 24-, or 32-bit chunks, as determined by the field width. A 2-to-1 multiplexer is utilized to allow the selection of a fixed field width (from the Protocol Memory) or a variable field width (from the Length Block). For fields that do not contain 32 bits worth of data, the most significant bits will be padded with zero.

Table 7. Current Content Inside Protocol Memory

Address	Description	Field Width	Branch Indicator	Protocol Indicator	Length Indicator	Variable Length	Count Done
Ethernet							
0	Preamble _high	3	0	0	0	0	0
1	Preamble_low	3	0	0	0	0	0
2	Destination_high	3	0	0	0	0	0
3	Destination_low	1	0	0	0	0	0
4	Source_high	3	0	0	0	0	0
5	Source_low	1	0	0	0	0	0
6	Type	1	1	1	0	0	0
ARP							
7	Hardware	1	0	0	0	0	0
8	Protocol	1	0	0	0	0	0
9	Hardware Address Length	0	0	0	0	0	0
A	Protocol Address Length	0	0	0	0	0	0
B	Operation	1	0	0	0	0	0
C	Sender Hardware Address (High bits)	3	0	0	0	0	0
D	Sender Hardware Address (Low bits)	1	0	0	0	0	0
E	Sender Internet Address	3	0	0	0	0	0
F	Target Hardware Address (High bits)	3	0	0	0	0	0
10	Target Hardware Address (Low bits)	1	0	0	0	0	0
11	Target Internet Address	3	0	0	0	0	0
12	Data Padding	3	0	0	0	0	0
13	Data Padding	3	0	0	0	0	0
14	Data Padding	3	0	0	0	0	0
15	Data Padding	1	0	0	0	0	0
16	(Ethernet)Frame Checksum	3	0	0	0	0	0
17	End of Frame	0	1	1	0	0	0
IP							
18	Version+Header Length	0	0	0	1	0	0
19	Type of service	0	0	0	0	0	0
1A	Total Length	1	0	0	1	0	0
1B	Identification	1	0	0	0	0	0
1C	Flags (3 bit)+Fragment Offset (13 bit)	1	0	0	0	0	0
1D	Time to Live	0	0	0	0	0	0
1E	Protocol	0	0	1	0	0	0
1F	Header Checksum	1	0	0	0	0	0
20	Source Address	3	0	0	0	0	0
21	Destination Address	3	1	0	0	0	0
22	(Options + padding)	X	1	0	0	1	0

Table 7 (continued)

TCP							
23	Source Port Number	1	0	0	0	0	0
24	Destination Port Number	1	0	0	0	0	0
25	Sequence Number	3	0	0	0	0	0
26	Acknowledgement Number	3	0	0	0	0	0
27	Header Length, Reserve, URG…	1	0	0	1	0	0
28	Window Size	1	0	0	0	0	0
29	TCP Check Sum	1	0	0	0	0	0
2A	Urgent Pointer	1	0	1	0	0	0
2B	(Options + paddings)	X	0	0	0	1	0
2C	Data	X	1	1	0	1	0
UDP							
2D	Source Port	1	0	0	0	0	0
2E	Destination Port	1	0	0	0	0	0
2F	UDP length	1	0	0	1	0	0
30	Checksum	1	0	0	0	0	0
31	Data	X	1	1	0	1	0

The shifter consists of four 8-bit registers to assemble the incoming blocks of 8 bits into 8-, 16-, 24-, or 32-bit chunks, as determined by the field width. A 2-to-1 multiplexer is utilized to allow the selection of a fixed field width (from the Protocol Memory) or a variable field width (from the Length Block). For fields that do not contain 32 bits worth of data, the most significant bits will be padded with zero.

The Protocol Address Register is a register that saves the current address of the Protocol Memory. This register is actually a loadable counter that can either be incremented by one to move to the next field within a protocol, or can be loaded with the value of the Jump Address from the Jump TLB. This loading capability allows for branches between protocols within the Protocol Memory.

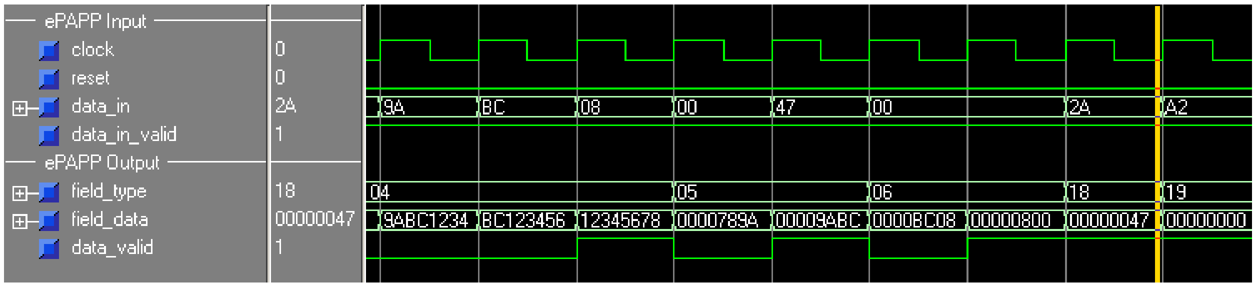


Figure 9. Simulation Waveform for ePAPP

Figure 9 shows a screenshot of a simulation waveform for ePAPP when processing a UDP packet. The inputs to and outputs from ePAPP are shown. The “data_in” signal is representative of data arriving on an 8-bit PHY. Note that, due to FIFO and queuing delays, there is a three cycle delay between arrival of data on the PHY and its appearance in the “field_data” output signal.

The “field_data” and “field_type” outputs are valid when the output “data_valid” is high. For example, when the Field Type is 0x06 and “data_valid” is asserted, the value of “field_data” is 0x00000800. Field Type 0x06 corresponds to the protocol field of an Ethernet header, and the data value 0x00000800 indicates the IP protocol. Consequently, one can observe a branch to Field Type 0x18. Logically, 0x18 corresponds to the first field of an IP header.

One significant benefit of this design is the generic nature of the field classification outputs. Most probably, the outputs would be buffered and made accessible to a channel processor through memory-mapped registers. This provides a very general architecture that leaves the use of the classification information to the specific processor implementation. Consider this implementation running at 250 MHz, which would support a 2 Gb/s peak rate. Further consider a network processor running at 2 GHz. If one aggressively estimates that on average a valid field is classified every two cycles, one is left with an average of 16 processor cycles per packet field. The processor would simply need to execute a set of instructions corresponding to a particular field type, lifting a significant pre-processing burden off of the network processor itself.

3.4 DESIGN RESULTS AND PERFORMANCE

The performance of the Protocol Analyzer must be sufficient to keep pace with the incoming data. For Gigabit Ethernet, the PHY transmits 8 bits every cycle at 125 MHz, thus requiring the same clock rate for the Protocol Analyzer. Fortunately, the internal architecture of the Protocol Analyzer utilizes only a small memory and TLB, both of which can reside internal to the chip, allowing a high clock rate.

It is expected that the final implementation of the Protocol Analyzer will be inside of a Network Processor using standard ASIC technology. However, to gauge this architecture's performance and size, the design has been targeted at a 130 nm structured ASIC solution. To do this, VHDL was created for the Protocol Analyzer and simulated for a Gigabit Ethernet PHY. The synthesis result is shown in Table 8.

Table 8. Performance Results After Synthesis

	130 nm Structured ASIC
Standard-cell Instance Count	2530 / 1.7M
Size as a % of Total Cells	< 1%
Speed in MHz	360 MHz
Throughput	2.8 Gb/s

Table 9 compares the performance of the hardware ePAPP with the software-based decoder inside the "Snort" Intrusion Detection System. The software-based decoder in Snort captures packets utilizing the widely-used *libpcap* sniffing interface, a public domain library of packet capturing utilities. The decoder program in Snort version 1.9.0 needs more than 1700 lines in C code and an average of 1245 ns execution time when running on a Dell Power-Edge 4400 server which has dual 866MHz Pentium III Xeon processors, 1GB RAM, and is running the Redhat 7.2 OS. The hardware-based ePAPP uses less than 1% of a structured ASIC target and processes packets at line speeds. Protocol analysis is performed in parallel with the capturing process, and most of the processing time overlaps with the capturing process, with a maximum input to valid output latency of 6 clock cycles (less than 17 ns when circuit operates at 360 Hz). Assuming the capturing times for software and hardware are the same, we found that the pipelined protocol analysis in ePAPP brings a 75x performance improvement (1245 ns / 17 ns) in the latency for getting analysis results after capturing a packet. This presents ePAPP with greater processing capability in high-speed networks. ePAPP and Snort support the same protocol group, with the exception of ICMP, however analysis shows that ICMP could be implemented without an appreciable effect on processing speed.

Table 9. Performance Comparison With Software

	Software-Based (Part of the Decoder in Snort IDS)	Hardware-Based ePAPP
Execution Type	Sequential	Six-stage Pipeline
Architecture	Dual 866MHz Pentium III Xeon Processors	130 nm structured ASIC
Size	1700 lines in C code	< 1% of target area
Protocol Support	Fixed based on available software instructions	Expandable by adding to or updating the Protocol RAM
Processing Speed	0.8 million packets per second	Line speed
Decoding Latency	(Average) 1245 ns	17 ns (6 clock cycles)
Speedup		75x Latency Decrease

In addition, Hardware ePAPP can connect directly to a physical layer interface (PHY), avoiding the use of time-consuming capturing software (including *libpcap*, which is reported to be a possible bottleneck of network monitoring process). The hardware solution may become indispensable to replace conventional software solutions in the network security area as faster networks are developed.

3.5 CONCLUSIONS

This chapter has introduced the Embedded Protocol Analyzer Pre-Processor (ePAPP) and shown that it performs protocol analysis pre-processing for packet processors by augmenting and transforming the data stream from the PHY to a stream of 8-bit and 32-bit words that uniquely identify individual protocol fields and their corresponding data, respectively. Currently, ePAPP supports protocols Ethernet, ARP, IPv4, UDP, and TCP, with a performance improvement of 75 times over the conventional functionally-equivalent software. Furthermore, by targeting this architecture at structured ASIC technology it has been shown that ePAPP uses less than 1% of the chip logic cells and executes at 361.5 MHz, and thus can be used for multi-Gigabit Ethernet.

Future directions include using the protocol analyzer in conjunction with embedded firewalls and network attached devices. In one implementation, this design would be paired with hardware using a ternary content-addressable memory (tCAM) to perform intrusion detection signature-matching as described in Chapter 5. Support will be added for additional protocols such as ICMP and IPv6, which should require only a trivial amount of redesign. Field length calculation information could be moved to a ROM in order to make protocol additions and changes involving length more easily reprogrammable. For Ethernet applications, ARP and RARP packets could be handled automatically from within the hardware and thus not burden the embedded processor.

4.0 CASMA: A CAM-ASSISTED SIGNATURE-MATCHING ARCHITECTURE FOR INTRUSION DETECTION

As transmission speeds and the amount of network traffic produced increase within modern networks, the amount of potentially harmful traffic grows as well. More than ever before, it is essential to continuously monitor internets and intranets for packets that are of suspicious origin or intent. The firewall has been a traditional solution to network protection, however simple packet-filtering firewalls on the incoming edge of a network are not a sufficient guard against network attacks for several reasons. One reason is that packet-filtering firewalls do not examine packet payload information, which could contain signatures instrumental in detecting malicious traffic. And even if a simple firewall could prevent any attack from entering the network, it would be helpless against an attack launched from within.

Intrusion detection systems (IDSs) have proven to be an improved measure of supplemental protection. Operating in a passive mode, intrusion detection systems analyze all traffic arrive through a specific network interface or network interfaces. Host-based IDSs can be an effective measure of protection for a single device, however the network at large can still fall victim to an unnoticed attack originating from a single unprotected host. Therefore, network intrusion detection systems (NIDSs) are the optimal choice to evaluate all traffic traveling through a network.

The question of where to position the NIDS on the network is an important consideration. Standalone NIDS devices have been made available, however adding additional devices to a network infrastructure can be costly and produce unwanted latency. Embedding the NIDS in the firewall is effective for traffic leaving and entering the network but useless for traffic traveling between internal hosts. In fact, an optimal location for intrusion detection on a local network is within the network switch. Any and all traffic sent to or from a host within a local network must travel through the network switch. Therefore, by augmenting traditional switching hardware with a hardware-based intrusion detection architecture, intrusion detection functionality can be achieved with minimal change to the network infrastructure.

The list of technical capabilities falling under the umbrella of intrusion detection has continued to grow. Stateful processing, anomaly or statistical-based detection, and packet decryption features have become available in many off-the-shelf IDS products. The aim of this research is to accelerate one critical aspect of the intrusion detection process: stateless packet signature matching. Signature-based detection is at the root of almost all IDSs. As the number of attack signatures to be searched grows, so does the latency required to perform a sequential search. Through the use of custom combinational logic and ternary content addressable memory (tCAMs), this work will attempt to demonstrate a clear performance improvement over Snort, a comparable software IDS solution.

4.1 THE SNORT INTRUSION DETECTION SYSTEM

Snort is an open source network intrusion detection system, based entirely in software. To capture packets directly from the network card, Snort uses the *libpcap* library [30]. libpcap is a system-independent C interface for low-level packet capture. Captured packets are passed to a packet decode engine, which serves much of the same purpose as ePAPP does in this work. Once packet capture and classification has been completed, Snort exposes packets to a series of preprocessors as defined by the current Snort configuration. These perform tasks such as stateful analysis, packet stream reassembly, performance monitoring, and other functions beyond the scope of this thesis.

Once packet preprocessing has been completed, the task of rule-matching begins. Snort rules are text-based, and lists of rules are typically stored in files to be read by Snort at startup. Each rule is its own line of text and can be broken into two sections: the rule header, and the rule body.

The information contained within a rule header can be divided into four main categories:

- Rule action
- Protocol
- Source Information
- Destination Information

The rule body is comprised of the rule options. A Snort rule does not require a rule body to be a complete rule. In most cases, however, it is the rule body that provides detail to a rule and differentiates the capabilities of Snort from a simple packet filter. Snort supports a number of rule options, some of which deal with protocol header information (non-payload options), others that deal with payload content (payload options), and still others that specify response characteristics or provide additional information or direction when a rule is matched (meta-data and post-detection options).

Although a single Snort rule may contain information to match any part or parts of a packet, it is useful to identify and separate the parts of a rule that deal with header information, payload information, or how to respond to a rule hit. Inherently, these rule options function in different ways. Header data is found at fixed locations within a packet, and therefore matching header data against a rule is fairly straightforward. Finding a certain data string within a payload, however, is much more difficult, since in many cases the data can appear at any byte offset within the payload. Response to a rule hit is not relevant until after a match has occurred. Section 4.1.1 discusses aspects of Snort rules pertaining to packet header searching. Section 4.1.2 focuses on rule options pertaining to packet payload searching. Section 4.1.3 considers rule options that dictate how to respond following a rule match.

4.1.1 Snort Packet Header Rule Options

As previously discussed, Snort rules are comprised of two parts: the rule header and the rule body. It is important that the term "rule header" not be confused with the parts of a Snort rule that enable packet header matching to occur. Options to assist in performing packet header matching are found in both the rule header and the rule body. Snort rule headers appear in the following format:

rule_action protocol source_addr source_port direction destination_addr destination_port

Consider the following example:

alert tcp 111.222.111.222 80 → 222.111.222.111 80

For the purposes of clarity, example rule headers will, in the future, be shown in table format. For example, the above rule would appear as:

Table 10. Example Rule Header Table Format

Rule Action	Protocol	Source Address	Source Port	Direction	Destination Address	Destination Port
alert	tcp	111.222.111.222	80	→	222.111.222.111	80

The protocol portion of the rule header is used to specify to what protocol a rule pertains. Currently, Snort supports four protocol types: IP, TCP, UDP, and ICMP. Because certain attack behaviors apply only to traffic of a particular protocol type, specifying a protocol in the rule header significantly lowers the number of false positives triggered by Snort.

The source information and destination information portions of the rule header are used to apply rules to specific source and destination IP addresses and port numbers. Consider the following rule header example in Table 11:

Table 11. Rule Header Example #1

Rule Action	Protocol	Source Address	Source Port	Direction	Destination Address	Destination Port
alert	tcp	any	any	→	136.142.42.14	80

The above example specifies that we perform an alert action any time TCP traffic is detected from any IP address and port to port 80 at IP address 136.142.42.14. The “any” keyword is used as a wildcard to specify a “don’t care” value for a port or IP address. Now consider the example in Table 12:

Table 12. Rule Header Example #2

Rule Action	Protocol	Source Address	Source Port	Direction	Destination Address	Destination Port
alert	udp	any	any	<>	$HOME_NET	0

In this example, the “→” operator is replaced by the “<>” operator. “→” indicates that a rule applies only to traffic in a particular direction. The previous example in Table 11, for instance, does not apply to traffic originating from port 80 at IP address 136.142.42.14. The “<>” operator, however, indicates that a rule is applicable regardless of the directionality of the traffic flow. “$HOME_NET” is a Snort variable. In fact, any rule entry beginning with a “$” is a variable. Variables can represent either a specific IP address or port, or a range of IP address or port. In this case, $HOME_NET refers to any IP address on the local network that Snort is monitoring. Therefore, the significance of the rule above is as follows: alert on any UDP traffic that is destined for or being sent from port 0 on any internal IP address.

Snort rules also support Classless Inter Domain Routing (CIDR). This makes it much easier to specify a range of supported IP addresses, much as one would by using a subnet mask. Consider Table 13:

Table 13. Rule Header Example #3

Rule Action	PROTOCOL	Source Address	Source Port	Direction	Destination Address	Destination Port
alert	ip	any	any	→	192.168.0.0/16	any

This rule alerts on any IP traffic destined for an IP address beginning with “192.168.” The number following the “/” character indicates the number of bits of the IP address that are to be considered, counting from the most significant bit. If, for instance, an IP packet were destined for “192.168.1.1,” an alert would be produced.

Snort also provides an easy way of specifying ranges of ports, as in Table 14:

Table 14. Rule Header Example #4

Rule Action	PROTOCOL	Source Address	Source Port	Direction	Destination Address	Destination Port
alert	tcp	$HOME_NET	21:23	→	any	any

The rule above triggers on any traffic originating from an internal IP address that is from ports 21, 22, or 23. Certain rules apply only to traditionally "privileged" ports, so it is not uncommon to see ":1024" specified in a rule, which implies any port from 0 to 1024.

The Snort rule body also contains options used to match packet header information. These options are known as the "non-payload" options and are summarized in Table 16. Consider the example rule below:

alert tcp any any → any any (ack:0; seq:0; fragbits:M;)

The options specified in this rule are separated in Table 15:

Table 15. Snort Rule Header and Body Example #1

Rule Action	Protocol	Source Address	Source Port	Dir.	Destination Address	Destination Port	ACK #	Seq #	Fragbits
alert	tcp	any	any	→	any	any	0	0	M

This rule will match a TCP packet with any source and destination in which the TCP ACK number is 0, the TCP Sequence number is 0, and the More Fragments flag is set. The contents of the rule body are always within parentheses, and each rule option is always terminated by a semi-colon.

4.1.2 Snort Packet Payload Rule Options

The Snort rule body contains another set of options known as "payload" options. Predictably, these are options that deal with matching packet payload data. These options are summarized in Table 17.

Table 16. Non-Payload Rule Options

Option	Description
fragoffset	This option compares the fragment offset field of an IP packet with a specified numeric value.
ttl	This option compares the time-to-live field of an IP packet with a specified numeric value.
tos	This option compares the type of service field of an IP packet with a specified value.
id	This option compares ID field of an IP packet with a specified numeric value.
ipopts	This option checks to see if any specific IP options have been set.
fragbits	This option checks to see if certain fragmentation and reserve bits have been set in the IP header.
dsize	This option compares the size of the packet payload with a specified numeric value.
flags	This option checks to see if certain TCP flag bits have been set in the TCP header.
flow	This option is used in conjunction with a Snort preprocessor to add some information about the state of connection to a Snort rule. This implies some level of stateful analysis, and therefore the *flow* keyword is ignored for the purposes of this research.
seq	The options compares the TCP sequence number with a specified numeric value.
ack	This option compares the TCP acknowledge number with a specified numeric value.
window	This option compares the TCP window size with a specified numeric value.
itype	This option compares the type value of an ICMP packet with a specified numeric value.
icode	This option compares the code value of an ICMP packet with a specified numeric value.
icmp_id	This option compares the ICMP ID number with a specified numeric value.
Icmp_seq	This option compares the ICMP sequence number with a specified numeric value.
rpc	This option searches for particular information in SUNRPC CALL requests. This rule relies on a preprocessor, and will not be supported in this thesis.
ip_proto	This option looks for a particular value in the IP protocol field, such as those corresponding to ICMP, TCP, UDP, and others.
sameip	This option checks to see if the source IP address is the same as the destination IP address.

Table 17. Payload Rule Options

Option	Description
content	The content keyword is the option that enables payload content matching. The content keyword precedes a string encapsulated in quotation marks, which is the content to be searched for in the payload. Not all data to be matched, however, can be nicely represented with ASCII characters. Everything else is specified in bytecode enclosed within pipe characters (|). For example: content:“hello|00 00|hello” The content string above would match a binary string containing the ASCII representation of “hello” followed by sixteen 0’s, followed immediately by the ASCII representation of “hello” once again. The *content* keyword has several modifier keywords that can be used to make content matches more specific. Note that multiple *content* rules can be specified within a single Snort rule.
nocase	This keyword modifies the *content* keyword and specifies that matches are to be case-insensitive.
rawbytes	This keyword modifies the *content* keyword and specifies that matches should ignore all changes to the payload performed by Snort preprocessors. Since preprocessing is not being considered in this thesis, the *rawbytes* keyword is ignored.
depth	This keyword modifies the *content* keyword and specifies the maximum depth for which Snort should continue looking for a content match. A depth value of five would tell Snort to look for the start of a content match only in the first five bytes of a payload. *Depth* may be specified from the start of the payload or from some *offset*.
offset	This keyword modifies the *content* keyword and specifies the minimum depth for which Snort should start looking for a content match. An offset of five would tell Snort to look for the start of a content match only after the first five bytes of a payload.
distance	This keyword modifies the *content* keyword and specifies the minimum depth for which Snort should continue looking for a content match *relative to the end of the previous pattern match.* The *distance* keyword is like the *offset* keyword, however it must be preceded by another *content* string.
within	This keyword makes sure at least N bytes are between pattern matches. It should be used in conjunction with the *distance* keyword.
uricontent	This keyword works the same as *content*, except only the URI section of a packet is searched as opposed to the entire payload. Snort uses a preprocessor to normalize the URI content before searching, however this work will not focus on performing normalization.
isdataat	This options verifies that data exists at a specified location, optionally relative to the end of a content match.
pcre	This option allows rules to be written using perl-compatible regular expressions.

Table 17 (continued)

byte_test	This option reads a set amount of data from a specified location in the payload, converts it to its numerical equivalent, and compares it to a specified value. The location to get the data within the payload can be relative to the start of the payload or a previous content match.
byte_jump	This option is the same as *byte_test*, except instead of comparing the value read to another specified value, the value read is used to jump a relative amount within the payload. This allows for relative pattern matches to take into account values found in the payload itself.

Consider the example rule below:

alert ip any any → any any (content:"VIRUS"; offset:2; depth:3;)

The options specified in this rule are separated in Table 18:

Table 18. Snort Rule Header and Body Example #2

Rule Action	Protocol	Source Address	Source Port	Dir.	Dest. Address	Dest. Port	Content String	Offset	Depth
alert	ip	any	any	→	any	any	"VIRUS"	2	3

This rule will match an IP packet with any source and destination in which the payload contains the string "VIRUS". Specifically, the string must be begin in the third, fourth, or fifth byte of the payload.

4.1.3 Snort Post-Match Rule Options

The only component of a Snort rule header that has not yet been described is the "rule action". There are currently five rule actions defined that can be specified as part of a Snort rule: pass, log, alert, dynamic, activate. *Pass* indicates that if a rule is matched, the packet should be ignored and analysis should continue onto the next packet. *Log* specifies that a packet should be logged for future analysis. *Alert* indicates that a packet should be logged and that some form of alert should be generated in response to the rule match. *Dynamic* rules remain dormant until triggered on by an *activate* action. *Activate* rules behave like *alert* rules but have the additional ability to enable *dynamic* rules. This thesis focuses exclusively on *alert* rule types.

There are two additional categories of Snort rule options not yet discussed. "Meta-data" options contain information about the rule, such as its origin and severity. "Post-detection" options provide detail about how the rule hit should be handled. These options are summarized in Table 19 and Table 20, respectively:

Table 19. Meta-data Rule Options

Option	Description
msg	This is the message that accompanies the alert that fires when the corresponding rule is matched.
reference	This identifies the source of additional information about the attack the rule addresses.
sid	The sid keyword is used to uniquely identify a Snort rule. It serves as the primary key for the rule.
rev	This keyword is used to specify the revision of a particular rule, and must be proceeded by an sid.
classtype	This defines the classification of the attack the rule addresses. Snort defines a default list of classtypes, each of which is assigned a priority.
priority	This assigns the rule an explicit priority. If a classtype is also present, the classtype's default priority is overridden.

Table 20. Post-detection Rule Options

Option	Description
logto	This option tells Snort to log the corresponding packet to a special output log file.
session	This option is used to extra data from user TCP sessions to a log.
resp	This option is used to attempt to close an open session when an alert is triggered.
react	This option allows for a flexible, customizable response to the appearance of a certain traffic type.
tag	This option allows rules to more than just the packet that matched the rule. Future packets with the same source or destination addresses are also logged to facilitate better post-attack analysis.

4.1.4 Summary of Snort Rules

Mapping Snort rules into content-addressable memory entries is not a trivial task. There are obviously quite a few Snort rule options to take into consideration. Because CAM entries are of a limited finite length, it is infeasible to attempt to encapsulate an entire Snort rule into a single CAM entry. The need arises to distinguish more than one type of search. As previously discussed, every option applies either to information found in the protocol headers of a packet, to information found within a packet's payload, or to a post-match response. Therefore, it is a logical choice to separate searches dealing with packet header information from searches attempting to match payload content. This design decision is further reinforced by the nature of the ePAPP component, which classifies and outputs all of the header information before any payload data is produced. Post-match behavior will be left to post-processing of the CAM search results, so these options will not be considered in the CASMA architecture.

For the purposes of this thesis, the Snort rules being evaluated for adaptation to the CASMA architecture are those that were supplied with Snort version 2.1.2. In total, this amounted to 1993 individual Snort rules. Figure 10 shows a breakdown of the number of times each rule option appears within the ruleset.

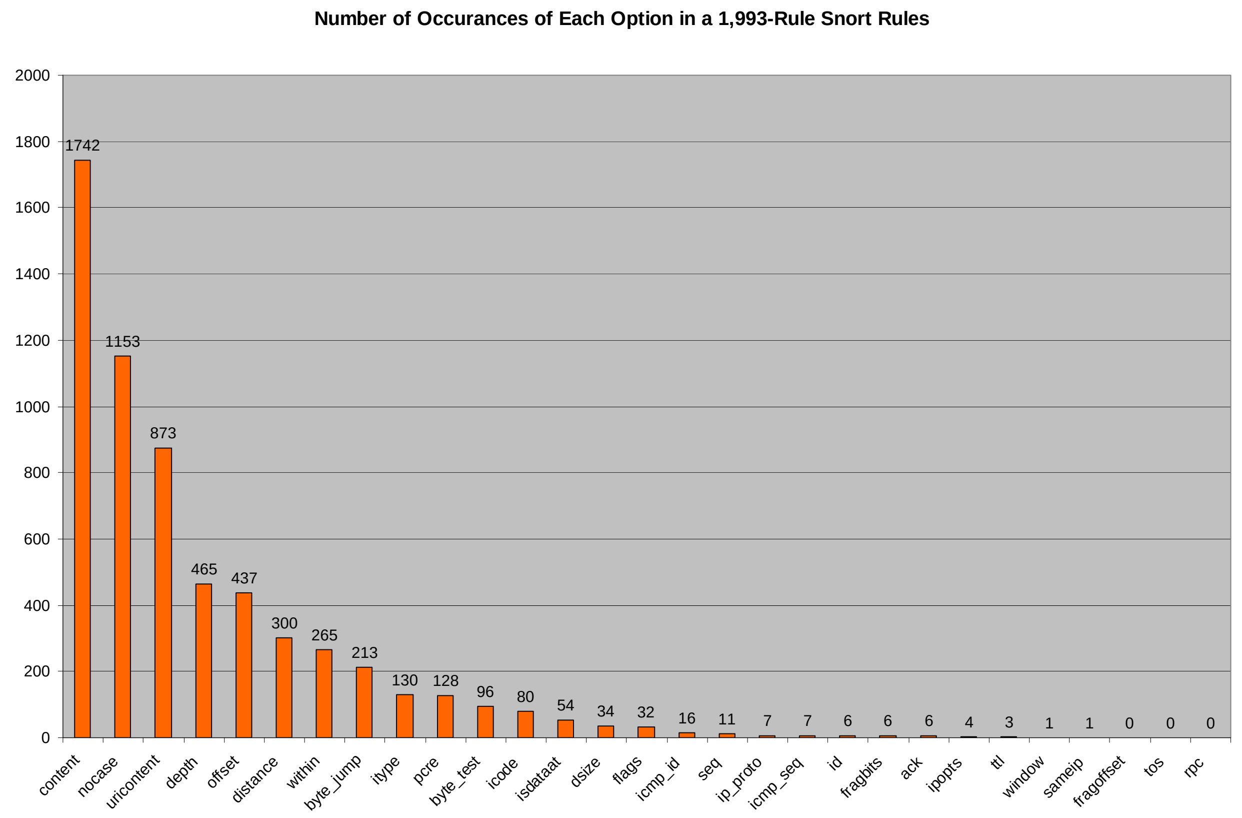


Figure 10. Number of Occurrences of Each Option in the Entire 1,993-Rule Snort Ruleset

4.2 TERNARY CONTENT-ADDRESSABLE MEMORY

The ever-increasing numbers of network attacks necessitate a corresponding rise in the number of intrusion detection rule signatures. Software-based and even specialized processor IDS solutions rely on sequential searching techniques, and therefore increases in the amount of rules to be searched adversely affect search times. Thus, an architecture that lends parallelism to the search process will achieve not only an immediate increase in search performance, but also a performance increase that will grow relative to future increases in the length of the ruleset.

This is the motivation behind the use of content-addressable memory (CAMs) in the search architecture. CAMs are capable of accepting a search string as input and producing search results in a fixed amount of time that is irrespective of the number of CAM entries (confined, of course, to the maximum depth of the CAM component.) Specifically, this design uses a *ternary* content-addressable memory (tCAM) to perform signature matching. tCAMs have the added ability to specify mask bits for each individual CAM entry, indicating which of the entry's bits are to be used in searching and which are "don't care" values. A mask bit value of '0' indicates a "don't care," while a value of '1' indicates a bit that will be used for searching. Consider the following 8-bit example, where an 'x' represents a "don't care" value in the string represented:

CAM data entry: 0 1 0 1 0 1 0 1

Associated mask entry: $\underline{1\ 1\ 1\ 1\ 0\ 0\ 0\ 0}$

Resulting string: 0 1 0 1 x x x x

Thus, only the first four bits of the CAM entry are to be considered when performing a search. A search using search string "0 1 0 1 1 1 1 1" would result in a match, for instance, while a string such as "1 1 1 1 0 1 0 1" would not.

Figure 11 shows an example with a populated CAM. A search performed using the input shown would produce hits on addresses three and five. Remember that each entry is actually comprised of a data component and a mask component.

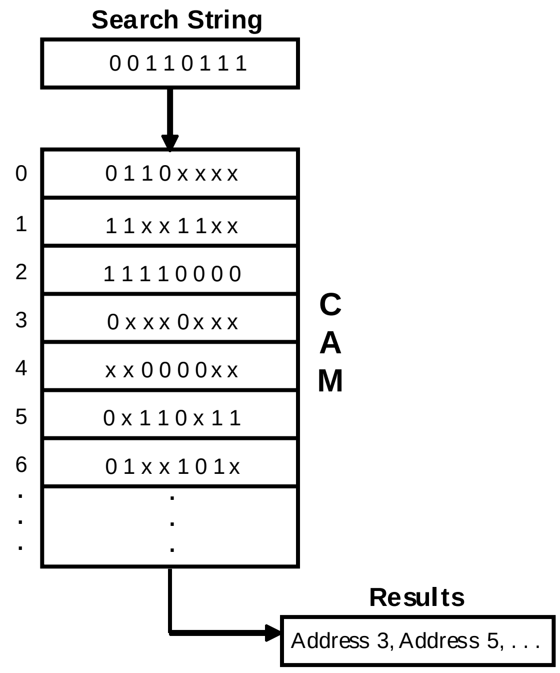


Figure 11. Example of a Search on a Populated CAM

For the purposes of this specific design, the 9Mb Network Search Engine provided by Integrated Device Technologies is used as the tCAM component. Capable of operating at clock speeds up to 200 MHz, a single instance of the NSE can hold 16,384 576-bit entries. A single 576-bit search that produces no result can be completed in 20 clock cycles. A single 576-bit search that produces hits requires 22 clock cycles, plus 8 additional cycles for every match found in the CAM.

4.3 PACKET HEADER SEARCHING

There are quite a few components of a Snort rule that pertain to information found in the protocol headers of a packet. Because packet headers are found in a static location within the packet, they can be addressed with a single CAM search. This section demonstrates that all packet header-related options within a Snort rule can be encoded into a single CAM entry for that rule, with entry bits to spare for future rule expansion.

The first part of a Snort rule, as discussed in Section 4.1, is the Snort rule header. This part of the rule identifies the source and destination IP addresses, the source and destination ports, the protocol being referenced, and the directionality of the traffic flow. All of this information is exposed in protocol headers. The following rule options are also dependent on information found in protocol headers: fragoffset, ttl, tos, id, ipopts, fragbits, dsize, flags, seq, ack, window, itype, icode, icmp_id, icmp_seq, ip_proto, and same_ip. Thus, the fields listed in Table 21, organized by protocol header type, must be included in a header search CAM entry.
The widths of all of these fields combined totals 312 bits. Assuming we are using a CAM entry width of 576 bits, and that 1 bit is used to indicate whether the CAM entry is a header search rule or a payload search rule, there are:

$$(576 - 1) - 312 = 263 \text{ unused bits}$$

Also, in order to facilitate the *dsize* rule option, we must calculate the length of the payload. In the case of TCP, the payload size is equal to the IP total length field minus the sum of the IP header and TCP header lengths. In the case of UDP, the payload size is equal to the IP total length field minus the sum of the IP header and UDP header lengths. Assuming that the maximum packet size is 1500 bytes, the length of the maximum Ethernet frame, the payload size can be represented with 11 bits. Therefore:

Table 21. Protocol Header Fields Included in a Header Search CAM Entry

Protocol Field	Width (in bits)	Field Description
Ethernet Header		
Type	16	This is required to identify an IP packet. Other packet types that travel over Ethernet, such as ARP and RARP, should not trigger any Snort rules.
IP Header		
Source IP Address	32	This is used to match against a source IP address specified in a rule.
Destination IP Address	32	This is used to match against a destination IP address specified in a rule.
Source Port	16	This is used to match against a source port specified in a rule.
Destination Port	16	This is used to match against a destination port specified in a rule.
Type of Service	8	This is used with the *tos* rule option
Identification	16	This is used with the *id* rule option.
Flags	3	This is used with the *fragbits* rule option.
Fragment Offset	13	This is used with the *fragoffset* rule option.
Time-To-Live	8	This is used with the *ttl* rule option.
Protocol	8	This is used with the *ip_proto* rule option. This will also be used to match TCP, UDP, and ICMP packets with the rules that correspond to each respective protocol.
Options	8	This is used with the *ipopts* rule option.
TCP Header		
Sequence Number	32	This is used with the *seq* rule option.
Acknowledge Number	32	This is used with the *ack* rule option.
TCP Flags	8	This is used with the *flags* rule option.
TCP Window Size	16	This is used with the *window* rule option.
ICMP Header		
ICMP Type	8	This is used with the *itype* rule option.
ICMP Code	8	This is used with the *icode* rule option.
ICMP ID	16	This is used with the *icmp_id* rule option.
ICMP SEQ	16	This is used with the *icmp_seq* rule option.

263 – 11 = 252 unused bits

If header searching were as simple as matching header field values extracted from a packet against values stored in CAM entry, there would be nothing left to do for header searching. Unfortunately, it is not that simple. Many of the Snort rule options support “greater than” (>), “less than” (<), and “not equal to” (!) operations. While tCAMs are excellent for performing exact matches, they offer no assistance with range checking or negation operations. We must instead use customized combinational logic to pre-compute all of the special rule cases that require a “>”, “<”, or “!” operation. Within the 1993 rules packaged with Snort version 2.1.2, there are 35 special cases that must be accounted for. Each can be represented by a 1-bit flag within the header search CAM entries. If the special case must be true to satisfy a rule, the header search CAM entry that corresponds to the rule will have a ‘1’ in the bit position that corresponds to the special case flag. Otherwise, the bit will be set to ‘0’. To accommodate these flags, 35 more bits of the CAM entry must be used:

252 – 35 = 217 unused bits

The remainder of these unused bits will be reserved for special case flags that will be added to accommodate future additions to the ruleset.

It is worthwhile to note that there will be many rules with the same header search requirements, so the number of header search CAM entries will be significantly less than the total number of entries. Several rules require, for instance, only that the destination port be port 80 for the rule to be applicable. Only one entry enforcing this requirement is necessary. Also, if there are one or more rules that require a condition A, and one or more rules that require a condition B, there is no need to create a separate entry for a rule that requires conditions A and

B. Each of these conditions will be matched individually during a CAM search, and search results can be correlated by a post-processor to indicate a rule hit.

4.4 PACKET PAYLOAD SEARCHING

While header searches attempt to find a match for information that spans several fields of several protocol headers, payload searches pertain to a single field: the packet payload. And while a packet’s header information can be checked for matches with a single search, many payloads will require a number of separate CAM searches to ensure an exhaustive search for content matches. This is because a content string that matches a rule can be found at any byte offset within a payload. Considering many payloads are greater than 72 bytes, the maximum width of a CAM entry, there is no way to avoid performing multiple payload searches. Fortunately, payload searching can begin as soon as there is enough data buffered to fill an entire search string, in many cases before the entire payload has been buffered.

The fact that a content string can be found at any byte offset within a payload complicates the payload search issue and gives rise to an important performance trade-off. Consider, for example, the content string “HELLO”. For the purposes of this example, assume that the width of a CAM entry is 10 bytes. One solution to the payload search problem would be to create a single CAM entry “HELLOxxxxx” where ‘x’ indicates a “don’t care” value. The payload could be searched 10 bytes at a time, shifting one byte after every search until the payload had been searched at every byte offset. With this method, a payload of length *n* would require *n* searches.

Another solution would be to create six different CAM entries: “HELLOxxxxx”, “xHELLOxxxx”, “xxHELLOxxx”, “xxxHELLOxx”, “xxxxHELLOx”, and “xxxxxHELLO”. Using this method, the payload would only have to be searched at every six-byte offset as opposed to every byte offset. Only (n-1)/6 + 1 searches would be required by a payload of length *n*. Clearly, the second method is much more efficient in terms of time – there is a six-fold decrease in time required for searching. In terms of space, however, there is a six-fold *increase* in the number of CAM entries required. A balance must be chosen which combines a reasonable amount of search time with acceptable space requirements.

Consider, now, that CAM entries are 72 bytes wide. Two of the 72 bytes are reserved for offset and depth information (explained later). This leaves 70 bytes with which to encode content match strings. Assume that payloads are searched at 8-byte offsets. This means that each content string would necessitate eight CAM entries, with the content string appearing zero bytes to seven bytes deep in the entry, respectively. Consider once again the content string “HELLO”. The following CAM entries would be required:

“HELLOxxxxxxxx . . . “
“xHELLOxxxxxxx . . .”
“xxHELLOxxxxxx . . .”
“xxxHELLOxxxxx . . .”
“xxxxHELLOxxxx . . .”
“xxxxxHELLOxxx . . .”
“xxxxxxHELLOxx . . .”
“xxxxxxxHELLOx . . .”

In the worst case, the content string begins seven bytes deep into the CAM entry. Therefore, a single CAM entry can support a content string of maximum length:

$$70 - 7 = 63 \text{ bytes}$$

Table 22 shows a relationship between the length of the payload (*n*), the number of payload content strings requiring CAM entries (*c*), and the number of resulting CAM entries for a shift amount of *s* bytes. This relationship is graphically represented in Figure 12.

Table 22. Effects on Time and Space Requirement for Varying Payload Shift Amounts

Shift Amount Between Searches, s (in Bytes)	# of Searches Required for Payload ((n-1)/s + 1)	# of Resulting CAM Entries	Maximum Supported Content String Length (70 – (s-1))
1	n	c	70
2	((n-1)/2 + 1)	2c	69
4	((n-1)/4 + 1)	4c	67
8	((n-1)/8 + 1)	8c	63
35	((n-1)/35 + 1)	35c	36
70	((n-1)/70 + 1)	70c	1

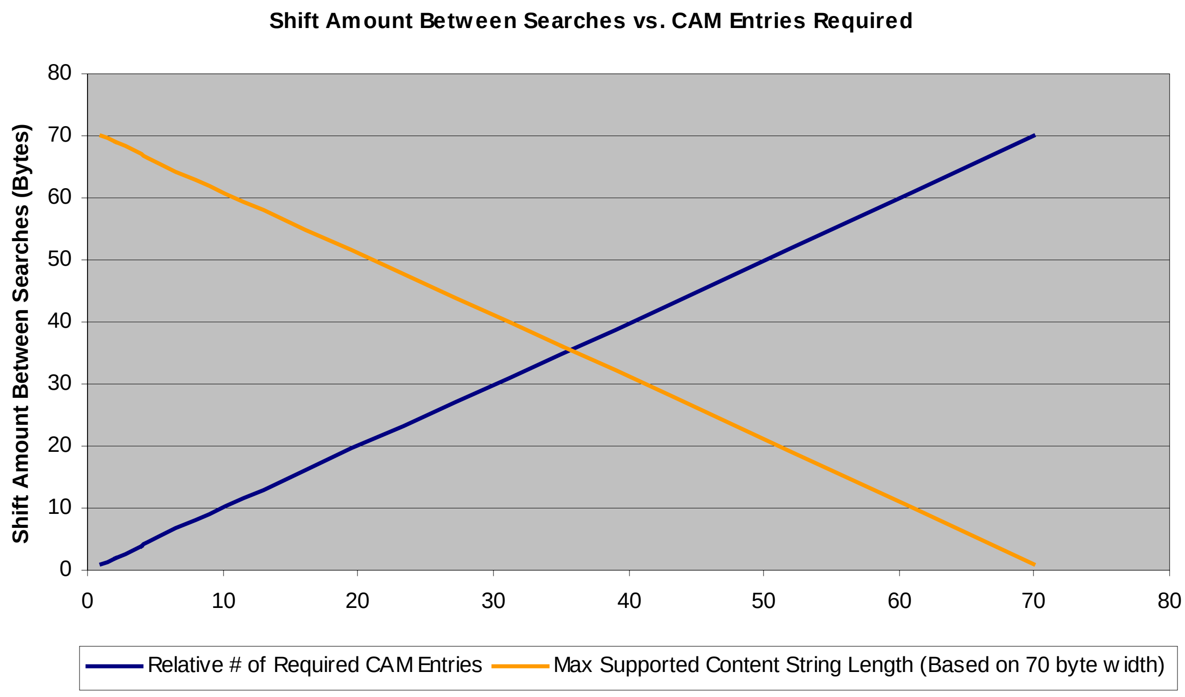


Figure 12. Shift Amount Between Searches vs. CAM Entries Required

Of the 1742 *content* and 873 *uricontent* strings specified by the Snort ruleset being used, only three strings are longer than 63 bytes. The ability to fit entire content strings in a single CAM entry reduces the amount of result correlation that must be performed by a post-processor, as well as limits the number of false positive CAM matches. This is the justification behind the design decision to search payload data at 8-byte offsets.

Some Snort rules contain several content strings. (Note that from this point forward, the term “content string” will be used interchangeably to refer strings found after a *content* or *uricontent* keyword within a Snort rule. The methodology to handle each keyword type is identical.) In the case where the relative positions of the strings within the payload are unrelated, each is regarded as a separate sub-rule. For instance, consider a rule where the strings “HELLO” and “GOODBYE” must both be present, but with no particular relation to one another. Encoding this into a single CAM rule would be quite infeasible, considering that a unique CAM entry would be required for each possible spacing of the strings within a payload. Furthermore, this method would not work if the strings appeared more than the width of a CAM entry apart. Instead, each unrelated content string is regarded as a separate sub-rule. Post-processing can be used to correlate search results on related sub-rules.

In some cases, content strings within a Snort rule are related through a content modifier keyword, such as *distance*, or *within.* In this case, all of the content strings that are related in some way must be considered in the same CAM rule, since the spacing between the strings within a payload is critical to discerning a rule hit. Consider the following rule body example:

(content:“HELLO”; content:“GOODBYE”; within:5;)

This rule states that a payload must contain “HELLO” and “GOODBYE”, and that “GOODBYE” must start within five characters (bytes) of the end of “HELLO”. To accommodate this rule, five new content strings can be developed to represent these requirements: “HELLOGOODBYE”, “HELLOxGOODBYE”, “HELLOxxGOODBYE”, “HELLOxxxGOODBYE”, “HELLOxxxxGOODBYE”. Obviously, as the *within* value becomes larger, more CAM entries will be required. Consider this example:

(content:“HELLO”; content:“GOODBYE”; within:5; content:“HOWDY”; within:5;)

This rule contains three content strings related to one another in a chain. To allow for all possible combinations within a payload, 25 contents strings must be considered in the CAM, ranging from “HELLOGOODBYEHOWDY” to “HELLOxxxxGOODBYExxxxHOWDY”. Remember, also, that each content string will result in eight separate CAM entries to account for every byte-offset. Now consider this example:

(content:“HELLO”; content:“GOODBYE”; distance:2; within:5;)

This is virtually the same as the first example considered, except that the *distance* keyword requires that two extra “don’t care” values be added after the “HELLO” string. The following content strings would result: “HELLOxxGOODBYE”, “HELLOxxxGOODBYE”, “HELLOxxxxGOODBYE”, “HELLOxxxxxGOODBYE”, “HELLOxxxxxxGOODBYE”. The *distance* keyword mandates a minimum offset from the end of a content match from which to search for the next content string.

None of the examples considered thus far have placed a restriction upon where in the payload a content string or group of related content strings must be found. The *depth* and *offset* Snort rule options do exactly that, however. Consider the following examples:

(content:“HELLO”; offset:3;)

(content:“HELLO”; depth:4;)

(content:“HELLO”; offset:3; depth:4;)

In the first example, a match is only valid if “HELLO” is found beyond the first three bytes of the payload. In the second example, a match is only valid if “HELLO” is found within the first four bytes of payload. In the third example, “HELLO” must be found within the four bytes following the first three bytes (bytes four through seven). When a payload search is performed, there must be some way of indicating the current depth of the payload information being searched. There must also be some way of indicating within a CAM rule entry at which payload depths the rule applies.

Consider the following example:

(content:“HELLO”; offset:10; depth:4;)

One can extract from the rule that “HELLO” must be found within bytes 11 to 14. It is also known that eight payload byte offsets are checked every payload search. Therefore, we know that bytes 11 through 14 would be checked during the second payload search. Our CAM entries for this rule are the following:

“HELLOxxxxxxxx . . . “
“xHELLOxxxxxxx . . .”
“xxHELLOxxxxxx . . .”
“xxxHELLOxxxxx . . .”
“xxxxHELLOxxxx . . .”
“xxxxxHELLOxxx . . .”
“xxxxxxHELLOxx . . .”
“xxxxxxxHELLOx . . .”

If our payload were “HELLOABCDEHELLOFGHIJ . . .”, the first payload search would search the following string:

“HELLOABCDEHELLOFGHIJ. . .”

The second payload search would use this string:

“DEHELLOKLMNO . . .”

One can see that there would be a match during each payload search. According to the content rule, however, only the second match should be valid. By specifying during which payload searches a CAM entry is valid based on the current search depth, we can achieve this functionality.

This paper proposes a one-hot encoding scheme to indicate search depth within a CAM rule entry. Fifteen bits of a CAM entry are employed, the least significant of which corresponds to the first payload search iteration, the most significant of which corresponds to search iterations 15 or greater. These bits will be referred to as “depth bits”.

If the content string in a CAM rule is valid at a certain payload search depth, a “don’t care” value is placed in the CAM bit that corresponds to that depth. If the content string is not valid at a given depth, a ‘0’ value is placed in the corresponding bit position. In the example above, matches are valid only during the second payload search iteration, so the 15 depth bits would be set as follows:

0 0 0 0 0 0 0 0 0 0 0 0 0 x 0

When a payload search is performed, 70 bytes of payload (beginning at the current 8-byte offset) are appended to 15 bits that are one-hot encoded to indicate the current depth and 1 bit set to ‘1’ to indicate that the present search type is payload search. For the first payload search iteration, the 15 encoded bits would be:

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

For the fifth search iteration, the 15 bits would be:

0 0 0 0 0 0 0 0 0 0 1 0 0 0 0

For search iterations 15 and above, the 15 bits would be:

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

When a CAM search is performed, the 15 bits that indicate the current search iteration are compared against the depth bits specified in the CAM entry. If the bit indicating the search depth lines up with a “don’t care” value, then the CAM entry is valid if a content match is found. If, however, the bit indicating the search depth lines up with a ‘0’ value, this indicates that a content match is not valid at this depth and a CAM hit will not be produced regardless of a content match.

Recall that each content string derived from a Snort rule requires eight CAM entries, each beginning at a different byte-offset. The fact that one of these offsets is valid for a certain search iteration not does guarantee that all of them are valid for the same iteration. Consider again our previous example:

(content:“HELLO”; offset:10; depth:4;)

A content match is only valid if “HELLO” is found beginning at bytes 11 through 14. Below are the eight content string offsets and their corresponding depth bit values:

“HELLOxxxxxxxx . . .” → 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
“xHELLOxxxxxxx . . .” → 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
“xxHELLOxxxxxx . . .” → 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
“xxxHELLOxxxxx . . .” → 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
“xxxxHELLOxxxx . . .” → 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
“xxxxxHELLOxxx . . .” → 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
“xxxxxxHELLOxx . . .” → 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
“xxxxxxxHELLOx . . .” → 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Four of the eight values are actually never valid and for optimization purposes can be left out of the CAM completely. Consider another example:

(content:“HELLO”; offset:10; depth:26;)

In this case, the depth at which a match is valid spans several search iterations, so all of the content string offsets will be valid. Below are the eight offsets and their corresponding depth bit values:

“HELLOxxxxxxxx . . .” → 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0
“xHELLOxxxxxxx . . .” → 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0
“xxHELLOxxxxxx . . .” → 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0
“xxxHELLOxxxxx . . .” → 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0
“xxxxHELLOxxxx . . .” → 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0
“xxxxxHELLOxxx . . .” → 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0
“xxxxxxHELLOxx . . .” → 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0
“xxxxxxxHELLOx . . .” → 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0

Because 15 bits are employed for this scheme and because each payload search employs an 8-byte offset, *offset* and *depth* values of less than 8 * 15, or 140, can be encoded. Recall that if an *offset* and a *depth* value are specified, the sum of these values must be less than 140.

When the *distance* keyword when not used in conjunction with *within*, it is difficult to encode in a CAM entry. The *distance* keyword works the same as *offset*, but begins counting from the end of a previous content match, which is unknown ahead of time. Content strings related only by the *distance* keyword should be treated as separate content strings. By passing the current search iteration value to the post-processor that is performing result correlation, it can be determined how deep into a packet a match occurred and, more importantly, whether one match occurred at an appropriate distance from another.

4.5 SNORT RULE ENCODING EXAMPLES

This section will demonstrate some sample encodings of representative Snort rules. CAM entries are 576 bits (72 bytes) wide, and will be displayed in eight rows of 72 bits each. Header information will be encoded in the following order: source address, destination address, source port, destination port, type of service, identification, IP flags, fragmentation offset, time-to-live, protocol, IP options, TCP sequence number, TCP ACK number, TCP flags, TCP window size, ICMP type, ICMP code, ICMP id, ICMP sequence number, Ethernet type, payload size, and special-case flags. Each CAM entry is shown with its corresponding mask entry. CAM entry “don’t care” values will be annotated with an ‘x’ character, but must actually be entered into the CAM as a ‘1’ or ‘0’. The last bit of a header search CAM entry is always set to ‘0’, and the last bit of a payload search CAM entry is always set to ‘1’, to differentiate the search types.

4.5.1 Snort Rule Example #1

alert tcp any any -> [232.0.0.0/8,233.0.0.0/8,239.0.0.0/8] any (msg:"BAD-TRAFFIC syn to multicast address"; flags:S+; classtype:bad-unknown; sid:1431; rev:6;)

This rule is triggered when packets with the SYN flag set are sent to multicast addresses. The rule contains no *content* or *uricontent* keywords, so no payload search CAM entries will be necessary. Three header rules will be necessary, however, one for each of the three destination address subnets specified. Four pieces of information must be encoded into each header rule:

- The Ethernet type must be set to IP.
- The IP protocol field must be set to TCP.
- The destination IP address must be specified.
- The TCP SYN flag must be set as on.

The following is the header search CAM entry for source IP address 232.0.0.0/8:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx11101000xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000110xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxx1xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxx0000100000000000xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx0
```

The corresponding mask entry is:

```
000000000000000000000000000000001111111100000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
111111110000000000000000000000000000000000000000000000000000000000000000
000000000000001000000000000000000000000000000000000000000000000000000000
000000001111111111111111000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000001
```

The other two required header search CAM entries would be identical, except for the change in the destination IP address part of the entry.

4.5.2 Snort Rule Example #2

```
alert udp any any -> any 69 (msg:"TFTP GET nc.exe"; content: "|0001|"; offset:0; depth:2;
content:"nc.exe"; offset:2; classtype:successful-admin; sid:1441; rev:2;)
```

This Snort rule pertains to Trivial FTP traffic, and triggers when filename “nc.exe” is found within the payload, indicating a possible malicious file. The rule will require one header search CAM entry and payload search CAM entries corresponding to two unrelated content strings. Three pieces of information must be encoded into the header search CAM entry:

- The Ethernet type must be set to IP.
- The IP protocol field must be set to UDP.
- The destination port must be specified.

The following is the header search CAM entry:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxx0000000001000101xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00010001xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxx0000100000000000xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx0
```

The corresponding mask entry is:

```
000000000000000000000000000000000000000000000000000000000000000000000000
000000001111111111111111000000000000000000000000000000000000000000000000
111111110000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000001111111111111111000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000001
```

The first content string, "|0001|", must be found within the first two bytes of payload. Only two CAM entries will be necessary to represent this rule. The second of these two entries is shown:

```
xxxxxxxx0000000000000001xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx00000000000000x1
```

The corresponding mask entry is:

```
000000001111111111111111000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000011111111111101
```

The second content string, "nc.exe", can be matched anywhere after the second byte of the payload. Eight CAM entries are required to represent this rule, the first of which is shown below (note that “011011100110001100101110011001010111100001100101” is the binary equivalent of “nc.exe”):

```
011011100110001100101110011001010111100001100101xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx01
```

The corresponding mask entry is:

```
11111111111111111111111111111111111111111111111111111111000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000011
```

4.5.3 Snort Rule Example #3

```
alert tcp any any <> $HOME_NET 179 (msg:"MISC BGP invalid type (0)"; content:"|ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff|"; offset:0; depth:16; content:"|00|"; distance:2; within:1; dsize:>1;
classtype:bad-unknown; sid:2159; rev:4;)
```

This rule triggers when a Border Gateway Protocol (BGP) packet with an invalid type has been detected. The rule will require two header search CAM entries and payload search CAM entries corresponding to two related content strings. Two header search CAM entries are required because the rule is bi-directional, and therefore the source and destination addresses must be swapped. Five pieces of information must be encoded into the header search CAM entry:

- The Ethernet type must be set to IP.
- The IP protocol field must be set to TCP.
- The source or destination address must be specified. In this case, a variable ($HOME_NET) is used, which for the purposes of this example corresponds to 123.123.0.0/16.
- The source or destination port must be set.
- The “dsize>1” flag must be set.

The following is the header search CAM entry which corresponds to $HOME_NET as the source address:

```
0111101101111011xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx00000000
10110011xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000110xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxx0000000010000000xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx1xxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx0
```

The corresponding mask entry is:

```
111111111111111100000000000000000000000000000000000000000000000011111111
111111110000000000000000000000000000000000000000000000000000000000000000
111111110000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000001111111111111111000000000000000000000000000000000001000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000001
```

Despite the apparent complexity of this rule, only content search string must be converted to CAM entries. The combination of a *distance* vale of 2 and a *within* value of 1 simply means that the content string must begin exactly three bytes after the other the other match ends. Eight CAM entries are needed to represent this rule, and the last of the eight is shown:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx1111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111xxxxxxxxxxxxxxxx00000000xxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx0000000000000xx1
```

The corresponding mask entry is:

```
000000000000000000000000000000000000000000000000000000001111111111111111
```

```
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111100000000000000011111110000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000011111111111001
```

4.5.4 Snort Rule Exceptions

There are a few Snort rule options that are not compatible with the suggested architecture. The *flow* option requires stateful processing which is beyond the scope of this thesis. The *nocase* option would require pipelined case-conversion of the payload. This work has already been accomplished elsewhere and is not addressed in this paper. The *isdataat* option cannot be performed within a CAM, since content-addressable memory has way of knowing whether the data being input for a search is valid. The *byte_test and byte_jump* function based on values read from within the payload itself. These rules are unpredictably dynamic and do not feasibly port to CAM entries. Finally, the *pcre* option supports perl-compatible regular expressions. These expressions do not convert well to static CAM entries, and therefore the *pcre* option is not implemented in this design.

Of the entire 1993-rule Snort ruleset packaged with Snort version 2.1.2, the architecture presented is capable of encoding 1729 of the rules into ternary content-addressable memory entries. Thus, 86.8% of the rule are supported.

4.6 THE CASMA ARCHITECTURE

This section describes the actual architecture used to prove the validity of the methodologies discussed earlier in this chapter. The 9Mb Network Search Engine provided by Integrated Device Technologies is used as the tCAM component. This device has the capacity to support 16,348 addressable entries, each of which supports a 576 bit (72 byte) data component and a 576-bit (72 byte) mask component.

Figure 13 recaps the position of the CASMA architecture within the pipelined switch architecture presented by this thesis. CASMA accepts packet data parsed into protocol fields from ePAPP and uses this data to perform stateless intrusion detection signature-matching, the results of which are passed to an upstream processor. Thus, the processor is relieved of the resource-consuming task of performing pattern matching.

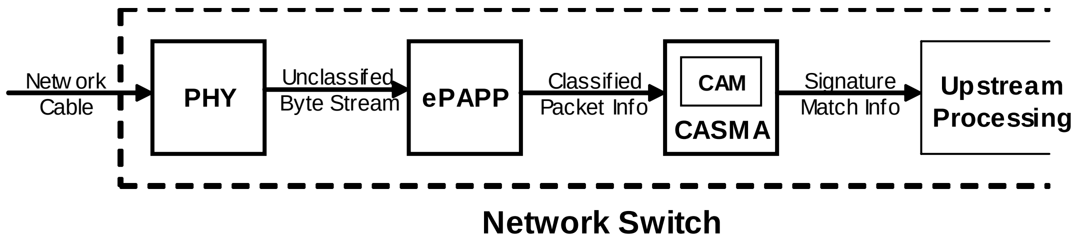


Figure 13. Position of CASMA in the Presented Switch Architecture

4.6.1 CASMA Data Flow

Data arrives to CASMA from the preceding ePAPP component in the form of a FieldData input, a FieldType input, and a Valid input. FieldData is a 32-bit bus that delivers chunks of packet data, separated into individual packet fields. FieldType is an 8-bit bus that concurrently identifies the type of packet field being delivered by FieldData. The Valid input is high whenever the data on the FieldData bus is a properly offset valid packet field.

Data on the FieldData bus that is identified as a packet header field by FieldType is stored in the Packet Header Register. Once all header information for a packet has been received from ePAPP, the HeaderReady signal is asserted. This alerts the Controller, a finite state machine (FSM), to initiate a packet header CAM search. Because the width of the tCAM component being used is 576 bits, this must also be the length of the search string passed from the Controller to the tCAM. The packet header search string is assembled with information from the Packet Header Register as specifically described in Section 4.5. Extra search string bits that are currently unused are padded with zeros. The Controller passes the search string to the tCAM in eight consecutive 72-bit chunks. The final bit of the final chunk is always set to ‘0’ to indicate that the search string contains header information.

While a packet header search is occurring, CASMA continues to receive data from ePAPP. This remaining data is payload data and must be buffered for future use in a packet payload search. As payload data is received a byte at a time, it is stored in a byte-wide FIFO. The depth of the FIFO must be such that the maximum payload size can be accommodated, in this case 1472 bytes. This is derived by subtracting the minimum IP header size (20 bytes) and the minimum UDP header size (8 bytes) from the maximum Ethernet frame size (1500 bytes).

The output of the FIFO is connected to a chain of 70 1-byte registers. There is also a single valid bit corresponding to each register that specifies whether or not the register contains valid payload data. On each cycle, the valid bit of the register farthest from the FIFO is checked. If it is set to ‘0’, a byte is read out of the FIFO into the first register, and all other registers and valid bits shift their values to the next register. This is done until the valid bit of the last register is set to ‘1’, indicating that 70 bytes of valid data have been retrieved from the FIFO. This data is passed to the Controller to be used as the data part of the packet payload search string.

After each payload search iteration, the entire payload must be shifted eight bytes. This is achieved by clearing the valid bits associated with the last eight registers in the payload register chain. This requires eight FIFO reads to repopulate eight invalidated bytes, and thus an eight-byte shift results.

Two counters are kept to keep track of the payload buffering status. Both counters begin counting when the first byte of payload data is received. The first counter is initialized to the length of the payload and counts down. When it reaches zero, payload buffering is complete and will not occur again until a new packet payload arrives from ePAPP. The second counter counts up from zero and triggers the PayloadReady signal when enough bytes have been buffered to begin performing packet payload searches.

An internal "Remaining" signal is kept which is initialized to the length of the payload. Each time the payload is shifted eight bytes, eight is subtracted from Remaining. If Remaining is greater than one, the SearchAgain signal is set to '1'. When Remaining dips below zero, SearchAgain is set to '0' to indicate that no more searches need to be performed on the current payload. Figure 14 is a visual overview of the internal design of the CASMA architecture.

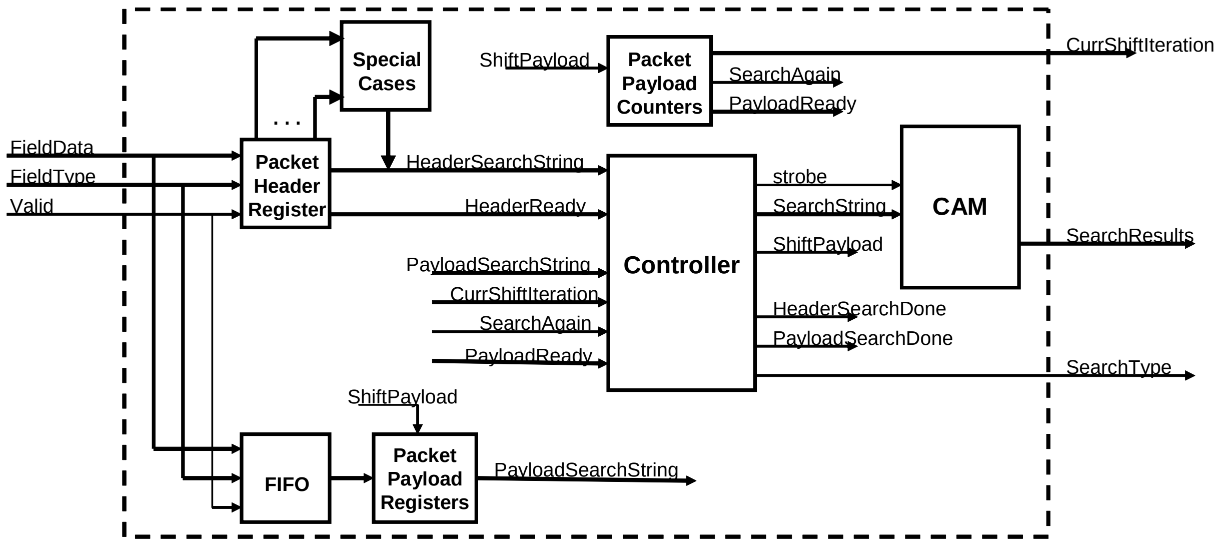


Figure 14. Internal Circuit Design of the CAM-Assisted Signature-Matching Architecture

The flow of the Controller state machine is shown in Figure 15. Each of the super-states shown contains a sub-hierarchy that performs the tasks associated with the super-state label. The

Controller remains dormant until the header information of the first packet received has been completely buffered. When this occurs, the PacketReady signal is asserted and the header search begins. If the search results in a match (HitAck='1'), the Controller proceeds to the "Scroll" state. The state machine within the hierarchical Scroll state will continue to output search matches until there are no more to report (MatchOut='0'). If a packet header search produces no hits, the Controller moves from "Header Search" to "Extra Cycles". This is because a tCAM search that produces no results requires a few extra cycles before it is able to perform a subsequent search.

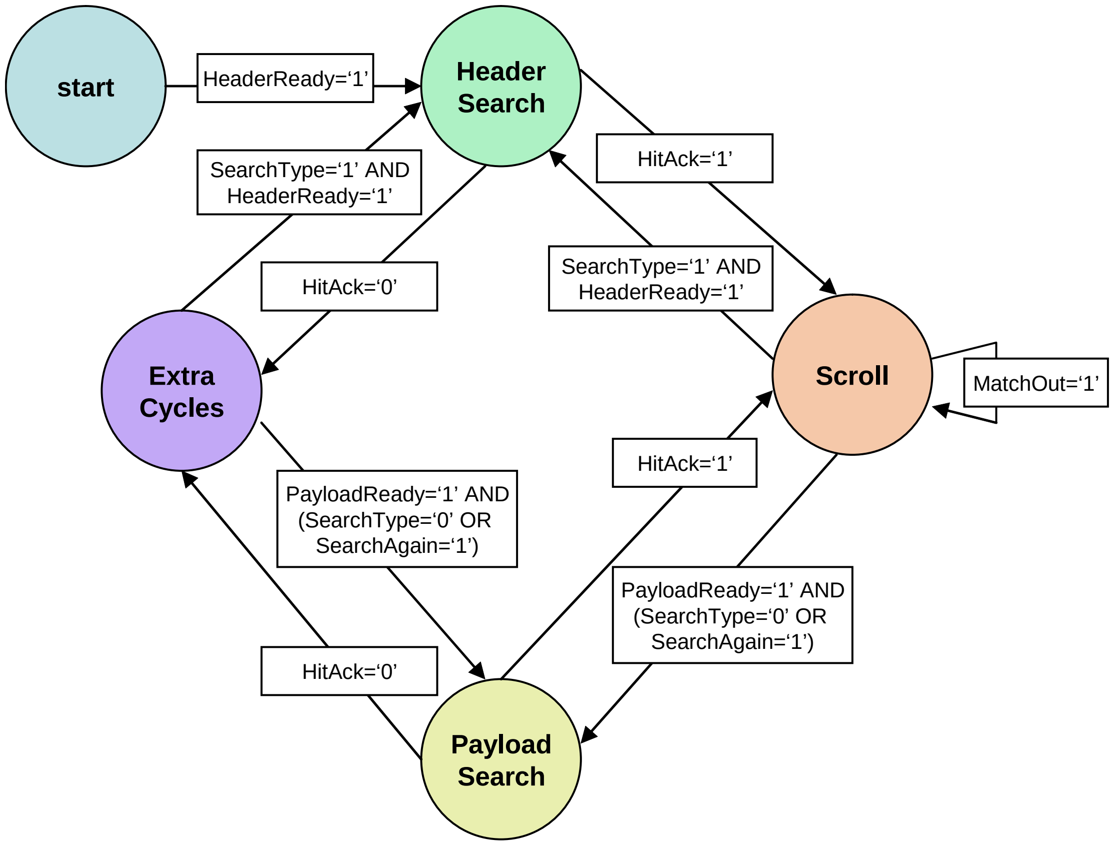

start
HeaderReady='1'
Header Search
SearchType='1' AND HeaderReady='1'
HitAck='1'
HitAck='0'
SearchType='1' AND HeaderReady='1'
Extra Cycles
Scroll
MatchOut='1'
HitAck='1'
PayloadReady='1' AND (SearchType='0' OR SearchAgain='1')
PayloadReady='1' AND (SearchType='0' OR SearchAgain='1')
HitAck='0'
Payload Search

Figure 15. Flow of the Controller State Machine

Regardless of whether a packet header search completes in the Extra Cycles or Scroll state, the same condition must apply to initiate packet payload searching: PacketReady must equal ‘1’. This indicates that enough payload information has been buffered to start a packet payload search. The “Payload Search” super-state works virtually the same as Header Search. The primary difference is that while only one header search can be performed per packet, multiple payload searches can be performed back to back. This is controlled by the “SearchAgain” signal. If SearchAgain is equal to ‘1’ at the conclusion of a packet payload search, the Controller returns to the Payload Search super-state and initiates a new CAM search.

4.6.2 CASMA Timing

This section discusses the timing and output that results for two consecutive search scenarios. These examples assume that the tCAM has been pre-loaded with the example rules discussed in Section 4.5. In total, mapping these three Snort rules to the tCAM required 24 tCAM entries and corresponding mask entries. Six of these entries are used for packet header searching, while the other 18 contain payload content strings to be matched.

To demonstrate the flow and associated timing of the CASMA architecture as it accepts packet data and performs CAM searching, two packets are sequentially passed to CASMA from ePAPP, and critical pieces of their simulation outputs are shown. The first packet is an example of benign network traffic, which should comprise a majority of the packets that pass through a switch on typical network. The second packet is a specially constructed malicious packet that matches all or part of each of the three rules.

Table 23 shows the relevant fields of the first packet that is being classified by the ePAPP unit and passed to CASMA. The packet, including the Ethernet preamble, is 103 bytes long. Since ePAPP produces packet data at a rate of 1 byte/cycle, it will take 103 clock cycles for CASMA to buffer the entire packet. This packet represents typical port 80 HTTP traffic and should not trigger any of the rules in the CAM and therefore will demonstrate minimal search times.

Table 23. Test Packet #1

Field Type	Value
Ethernet Type	0x0800 (IP)
IP Header Length	8 (in 32-bit words)
IP Type of Service	0x00
IP Total Length	77 bytes
Identification	0xA28A
IP Flags	None
IP Offset	0
IP Time-to-Live	0x80
Protocol	0x06 (TCP)
Source IP Address	136.142.42.14
Destination IP Address	64.233.161.99
IP Options	None
TCP Source Port	80
TCP Destination Port	80
TCP Sequence Number	0x41DA6AEE
TCP Acknowledge Number	0xBBB59568
TCP Header Length	5 (in 32-bit words)
TCP Flags	SYN
TCP Window Size	30
TCP Data (Payload)	"|01 02 03 04 05 06 07|Index of /cgi-bin/"
Payload Size	25 bytes

Consider that the cycle in which the first byte of a packet is received from ePAPP is cycle 0. It takes 74 cycles to read in all of the packet header information, which means that packet payload information begins arriving in cycle 74. Subsequently, since the entire packet header has been registered, packet header searching begins in cycle 75. Figure 16 shows the packet header search beginning. Note that the CAMStrobe signal goes high for one cycle, indicating that the first of eight 72-bit chunks of the search string is on CAMDataBus. The other seven chunks are placed on CAMDataBus over the next seven cycles. Note that CAMStrobe goes high one cycle after FieldType changes to “2C”, indicating that the payload is now being received and all packet header information has been registered. Eight cycles after the final piece of the search string is placed on CAMDataBus, SearchValid goes high for two cycles. When this signal goes high but SearchHit does not, it indicates that a search has completed successfully but returned no results.

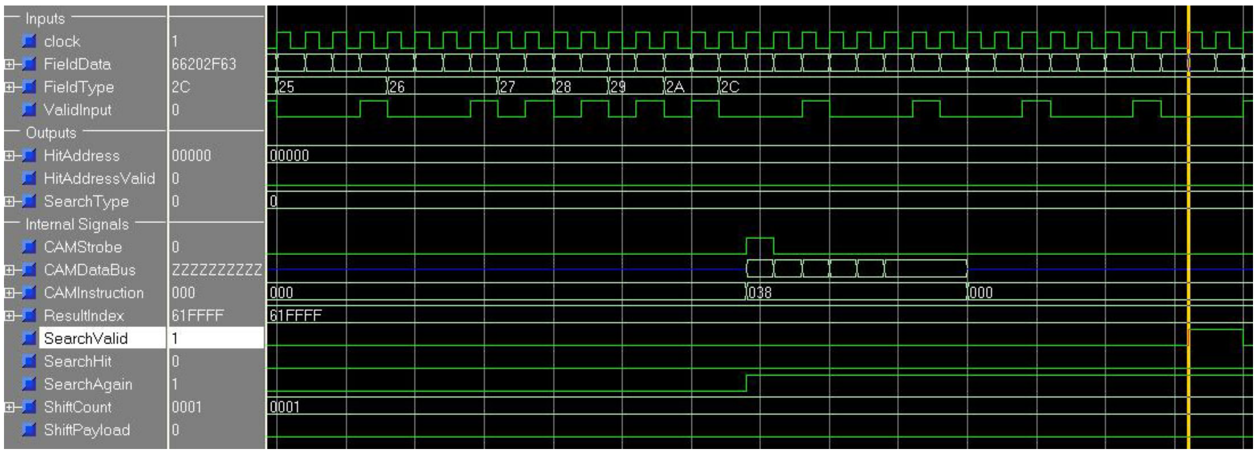


Figure 16. Header Searching Beginning for Test Packet #1

Thus, it takes eight cycles to feed the search data to the CAM, and eight cycles more to return a result. The SearchValid signal goes high for cycles 90 and 91. When a search returns no results, subsequent searches may not begin until 20 cycles after the first search began. Since the first search began at cycle 74, another CAM search could not begin until cycle 94.

At cycle 94, however, 70 bytes of payload have not yet been buffered, so payload searching is not ready to begin. Payload searching does not begin, in fact, until cycle 147. This is a current weakness of the design, that a short payload such as the one in the first test packet must still wait 70 cycles before payload searching can begin, even though it takes much less time than that to buffer the entire payload.

Figure 17 shows the beginning of a packet payload search. Again, the CAMStrobe signal goes high for a cycle and data is placed on CAMDataBus for eight consecutive cycles. Although not pictured, one can assume that the PacketReady signal has been asserted to allow the payload search to start. Note that the cycle after the payload search begins, SearchType changes from ‘0’ to ‘1’ to indicate that a packet payload search is taking place. Also, after the eight chunks of search string have been fed to the tCAM, ShiftPayload is asserted for one cycle to indicate that the payload should be shifted eight bytes to prepare for the next search iteration. Several cycles later ShiftCount is increased to reflect the shift in the payload. Again, it takes eight cycles to feed the search string to the CAM and another eight before results are displayed, so SearchValid goes high for cycles 163 and 164. A new search can begin at cycle 167, and indeed does, as there are more payload search iterations to perform.

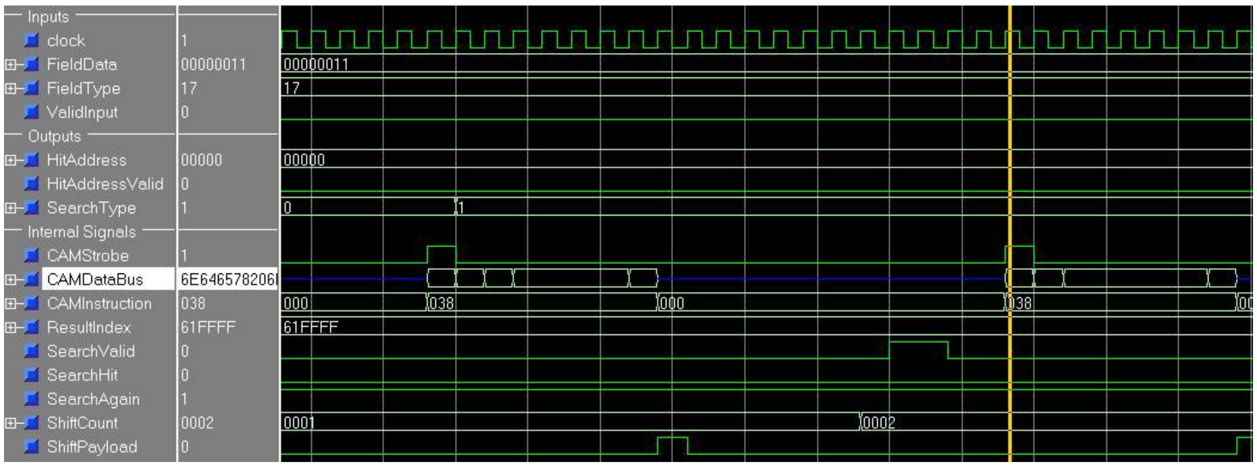


Figure 17. Payload Searching Beginning for Test Packet #1

In total, four iterations of payload searching are needed for this packet. Figure 18 shows the final search iteration of the packet payload. Observe that the SearchAgain signal goes low one cycle after the last piece of the search string has been placed on CAMDataBus. At the far right of the waveform, one can observe that the FieldType changes to “00”, which indicates that a new packet is incoming from the ePAPP. The final search iteration begins at cycle 207, and thus payload searching is complete and the CAM is ready to perform a new search 20 cycles later (cycle 227). Packet header searching began at cycle 75, and therefore it took 152 cycles from the beginning of the first packet header search to the end of the final packet payload search. As discussed previously, the packet was 103 bytes in total length and would require 103 cycles to be received from ePAPP. Thus, for this packet, 49 cycles would be required between packet arrivals to sustain searching of every packet. Again, this number could be lowered significantly by not requiring 70 cycles to buffer a payload that is much shorter than 70 bytes.

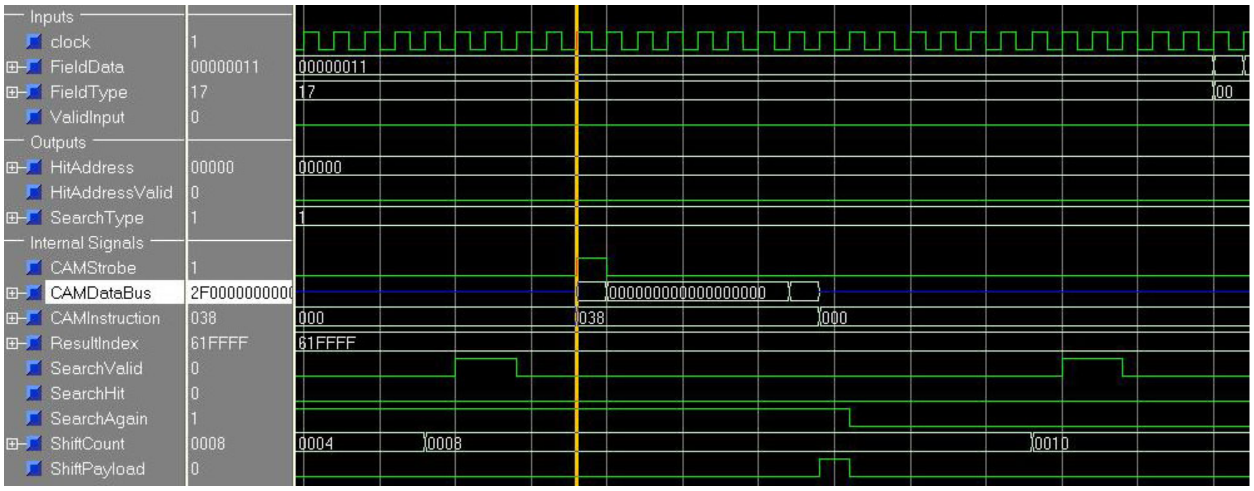


Figure 18. Payload Searching Concluding for Test Packet #1

Table 24 shows the relevant fields of the next packet that is being classified by the ePAPP unit and passed to CASMA. This packet is chock full of suspicious content and should match against several header and payload rules. Note that this packet has been specially crafted to match against several rules and may not make much sense as an actual network packet.

This packet will match Snort rule example #1, as described in Section 4.5.1. This is because the packet is of protocol type TCP, the SYN flag is set, and the destination IP address falls with the subnet 239.0.0.0/8. Since there is payload component to the first rule example, a header search match is enough to declare an entire rule match.

Table 24. Test Packet #2

Field Type	Value
Ethernet Type	0x0800 (IP)
IP Header Length	8 (in 32-bit words)
IP Type of Service	0x00
IP Total Length	79 bytes
Identification	0xA28A
IP Flags	None
IP Offset	0
IP Time-to-Live	0x80
Protocol	0x06 (TCP)
Source IP Address	123.123.16.8
Destination IP Address	239.24.16.8
IP Options	None
TCP Source Port	179
TCP Destination Port	80
TCP Sequence Number	0x41DA6AEE
TCP Acknowledge Number	0xBBB59568
TCP Header Length	5 (in 32-bit words)
TCP Flags	SYN
TCP Window Size	30
TCP Data (Payload)	“|00 01 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff 00 00 00|nc.exe”
Payload Size	27 bytes

This packet will also match Snort rule example #3, as described in Section 4.5.3. The header search will return a match because the packet is of protocol type TCP, the payload size (dsize) is greater than 1, the source port is 179, and the source IP address falls with the 123.123.0.0/16 subnet defined as $HOME_NET. The example rule also has two distance related payload content strings, both of which are found at acceptable depths within the payload of test packet #2.

Snort rule example #2, as described in Section 4.5.2, is a little more interesting. The header part of the rule does not match against the test packet, however both of the content strings specified by the rule are found in the packet payload. These are examples of CAM search hits that would have to be thrown out when they could not be correlated to a header search hit by an upstream processor.

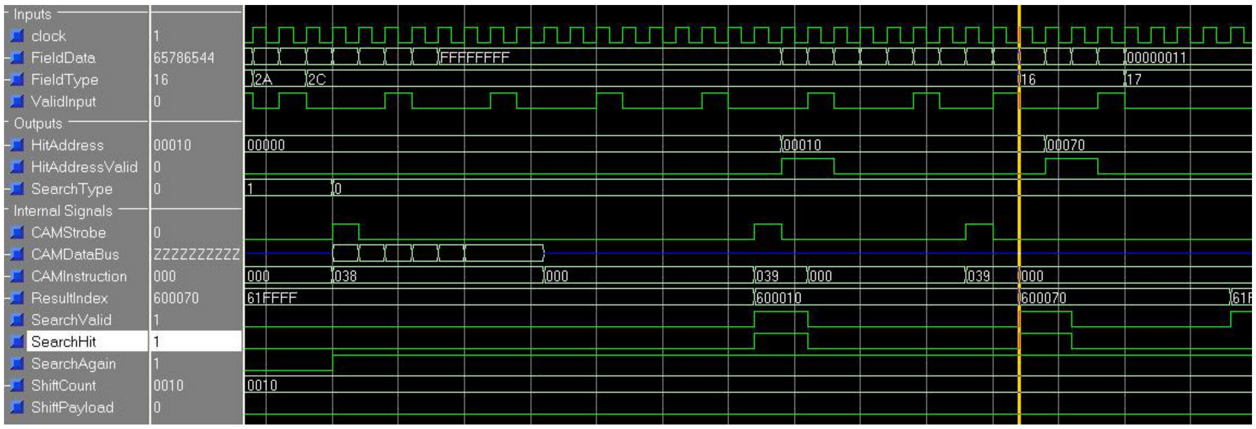


Figure 19. Header Searching Beginning for Test Packet #2

Figure 19 shows the next packet header search beginning. Again, we will consider cycle 0 to be the cycle in which the first byte of the packet arrives from ePAPP. This packet is two bytes longer than the first test packet, so the packet will require 105 cycles to be buffered. As in the previous example, the search begins a cycle after FieldType changes to “2C”, which is once again cycle 75. SearchType reverts to ‘0’ to indicate a header search. This time around, the packet header search produces two hits. The first hit is indicated by the SearchValid and SearchHit signals going high for two cycles. At the same time, CAMStrobe goes high for a cycle and a “scroll” instruction is sent to the tCAM to check for additional search hits. A new scroll instruction is issued every eight cycles until there are no more hits to output. The address that identifies the location of a hit within the CAM becomes available from the CAM when SearchHit goes high and is available on the HitAddress output a cycle later. As before it takes eight cycles to feed search data to the CAM and eight to wait for results. If there are matches, eight additional cycles are required for each match, plus eight additional cycles before the next search can begin. Therefore, a search with two matches requires:

$$8 + 8 + 2*8 + 6 = 38 \text{ cycles}$$

Thus, header searching begins in cycle 75 and completes in cycle 113, which is when the next search could take place.

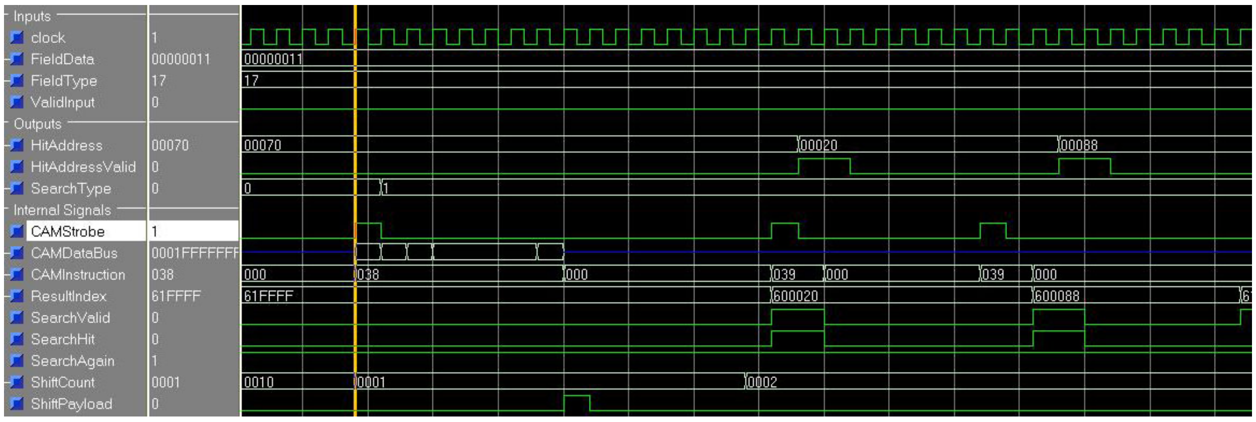


Figure 20. Payload Searching Beginning for Test Packet #2

The payload is ready to begin packet payload searching in cycle 147. Figure 20 shows the beginning of packet payload searching for the second packet. As shown in the waveform, the first search iteration produces two CAM hits. As before, a search with two CAM hits will require 38 cycles to complete, so the second search iteration will be able to begin at cycle 185. Figure 21 shows the third search iteration taking place. One can tell that it is the third iteration by observing SearchCount, which is one-hot encoded. The first iteration is represented by

“0x0001”, the second iteration by “0x0002”, the third iteration by “0x0004”, and so on. As is shown, the third iteration produces a single match. A CAM search that produces a single match will take 30 cycles to complete.

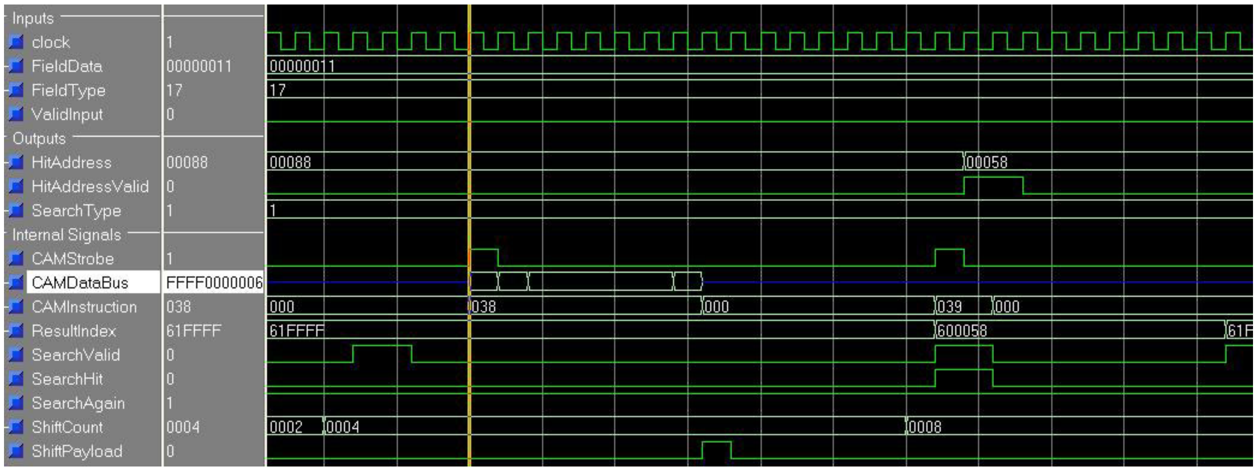


Figure 21. Payload Searching Concluding for Test Packet #2

As seen in Figures 20 and 20, there were two payload search CAM hits in the first search iteration and one in the third iteration. These three matches correspond to the payload content rules discussed earlier in this section. In total, this packet required four payload search iterations. The final iteration begins in cycle 235 and produces no hits, which means that searching for another packet could begin 20 cycles later at cycle 255. Packet header searching began at cycle 75, and therefore it took 180 cycles from the beginning of the first packet header search to the end of the final packet payload search. As discussed previously, the packet was 105 bytes in total length and would require 105 cycles to be received from ePAPP. Thus, for this packet, 75 cycles would be required between packet arrivals to sustain searching of every packet. As before, this number could be lowered significantly by not requiring 70 cycles to buffer a payload that is much shorter than 70 bytes.

4.6.3 CASMA Testing Methodology

The presented CASMA design was tested through simulation in conjunction with the ePAPP module. A behavior model of the 9Mb Network Search Engine provided by Integrated Device technologies was populated with a subset of diverse Snort rules, including the Snort rule examples described in Section 4.5. A simulation input file was created that fed a byte of packet data to the FIFO of the ePAPP module on the rising edge of every clock cycle. Use of the a simulation input file allowed for easy changes to packet lengths and protocol field values in order to perform diverse testing on the CASMA architecture.

4.6.4 CASMA Technology Mapping Results

The eventual target of the CASMA architecture is a standard ASIC platform. For the purposes of this thesis, however, the design was targeted at a 130 nm structured ASIC, the same target upon which the ePAPP architecture was synthesized. The results of design synthesis for CASMA are shown in Table 25.

Table 25. Performance Results After Synthesis

	130 nm Structured ASIC
Standard-cell Instance Count	99045 / 1.7M
Size as a % of Total Cells	5.9%
Speed in MHz	157.1 MHz
Throughput	1.25 Gb/s

The design is relatively expensive in terms of area due to the fact that entire incoming packets of up to 1500 bytes are buffered. This problem could be somewhat alleviated in several ways. Currently, for example, all header information is currently buffered upon input to CASMA, regardless of whether it is used in the header search string. Packet fields not used in intrusion detection could be ignored upon input. Another future optimization would be to significantly reduce the depth of the payload FIFO. As long payloads are buffered, packet searching will have already begun, thereby reducing the maximum amount of data the FIFO would be required to hold from the maximum payload length.

As discussed in Chapter 3, ePAPP would need to operate at a clock speed of 125 MHz to sustain 1 Gb/s traffic. Assuming that CASMA uses the same clock as ePAPP, this rate would also serve as the minimum requirement for CASMA. Synthesis results show that the architecture presented exceeds this requirement. Currently, the 9Mb Network Search Engine being used as the tCAM component supports a maximum clock frequency of 200 MHz. Future work involving design and synthesis optimization could yield significant increases in sustainable clock frequency.

5.0 CONCLUSIONS AND FUTURE DIRECTIONS

Increases in network transmission speeds and traffic loads have created a need for cost-effective, low-latency network security solutions. Packet protocol analysis and stateless intrusion detection signature-matching are critical components of this need. This thesis provides the following contributions to these areas:

- The Embedded Protocol Analyzer Pre-Processor (ePAPP) architecture, a pipelined design that classifies *every protocol field* of an incoming packet as the packet is captured.
- The ability to update the protocol information used by ePAPP in a loadable memory without changing the circuit implementation. Since frequent circuit redesign is not needed, ePAPP can be targeted for an ASIC implementation that could be feasibly added to a network device, such as a switch.
- The generic nature of the ePAPP output, allowing it to be used with any number of applications that require packets to be parsed and protocol fields to be identified. Network intrusion detection is one such application.
- The CAM-Assisted Signature-Matching Architecture (CASMA), a pipelined design that uses ternary content-addressable memory to aid in stateless intrusion detection signature-matching.
- A novel approach for encoding Snort intrusion detection rules into 576-bit CAM entries. This includes separate encoding techniques for entries pertaining to packet header information and entries pertaining to packet payload information.

- A novel approach for restricting the depth within a payload that a content string match is considered a rule hit. Without this, false positives would be generated for content matches that occur in a part of the payload that is not indicative of unwanted traffic as specified by a Snort rule.
- The ability to add new rules to the CAM without the frequent need for redesign of the CASMA circuit design. Since frequent circuit redesign is not needed, CASMA can be targeted for an ASIC implementation that could be feasibly added to a network device, such as a switch.

While ePAPP currently supports the IEEE 802.3 (Ethernet), IPv4, ARP, TCP, and UDP protocols, the design would benefit from the addition of support for more protocols, such as IPv6 and ICMP. Furthermore, internal calculations that use information found in protocol length fields to determine the lengths of other variable width fields are currently performed in combinational logic, and may not be adaptable to certain protocol changes. Future work should attempt to permit the nature of these calculations to be updated with other protocol changes.

The CASMA architecture was designed, for reasons of cost, to use only a single CAM component. As the cost for these devices falls, future efforts should consider multi-CAM implementations. Additional CAMs could speed the pipeline significantly by overlapping header and payload searching. Furthermore, since more CAM space would be available, payload searches could be performed at greater shift-increments. This thesis proposes a payload shift of eight bytes between every search. Increasing the shift amount to 16 bytes would cut the search time required in half, while doubling the amount of CAM entries required. If additional CAM space is not a restricting factor, this could be a worthwhile design adjustment.

The current CASMA design requires combinational logic to calculate special header search cases that require range or negation checking, functions not able to be performed by ternary content-addressable memory. The addition of a new rule to the CAM may require a special case calculation that has not yet been included in the circuit design. This is a design weakness, however traffic analysis may indicate key threshold values for which special case calculations may become necessary. Future attempts to predict relevant special cases could minimize the number of new rules that would require a circuit modification. Furthermore, the addition of programmable ALUs to the design could allow for table-based additions to the special case calculations, eliminating the need for circuit redesign.

The task of correlating CAM search results is not addressed within the scope of this work. Future work should address methodologies to assist with this problem. This work could include a method of restricting the subset of payload CAM entries searched based on the results of a header CAM search for the same packet.

Several Snort rule options, including *byte_test*, *byte_jump*, *isdataat*, and *pcre* are not compatible with a CAM-based approach. Future work should explore options that could use CAM searching in conjunction with pre- and/or post-processing to implement these options.

APPENDIX A

VHDL CODE FOR THE ANALYZER_TOP ENTITY

This appendix contains the VHDL code for the *analyzer_top* entity of ePAPP. This was the top-level entity used by ePAPP, and contains instances of the *assembler, protocol_memory, jump_tlb,* and *length_block* entities. Specific details about the code can be found as comments within the code itself.

```
-- VHDL Entity ProtocolAnalyzer.analyzer_top.symbol
-- Created by Jacob J. Repanshek
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY analyzer_top IS
  PORT(
    clock         : IN     std_logic;
    line_in       : IN     std_logic_vector (7 DOWNTO 0);
    line_in_valid  : IN     std_logic;
    reset         : IN     std_logic;
    field_data_out : OUT    std_logic_vector (31 DOWNTO 0);
    field_type_out : OUT    std_logic_vector (7 DOWNTO 0);
    valid_out      : OUT    std_logic
  );

END analyzer_top ;

-- VHDL Architecture ProtocolAnalyzer.analyzer_top.struct
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
```

```
LIBRARY ProtocolAnalyzer;

ARCHITECTURE struct OF analyzer_top IS

   -- Internal signal declarations
   SIGNAL aempty              : std_logic;
   SIGNAL almost_valid        : std_logic;
   SIGNAL branch_indicator    : std_logic;
   SIGNAL count_done          : std_logic;
   SIGNAL enable              : std_logic;
   SIGNAL ffout               : std_logic_vector(7 DOWNTO 0);
   SIGNAL field_data          : std_logic_vector(31 DOWNTO 0);
   SIGNAL field_data_early    : std_logic_vector(31 DOWNTO 0);
   SIGNAL field_data_valid    : std_logic;
   SIGNAL field_type          : std_logic_vector(7 DOWNTO 0);
   SIGNAL field_width         : std_logic_vector(15 DOWNTO 0);
   SIGNAL jump_addr           : std_logic_vector(7 DOWNTO 0);
   SIGNAL jump_addr_cur       : std_logic_vector(7 DOWNTO 0);
   SIGNAL jump_addr_n         : std_logic_vector(7 DOWNTO 0);
   SIGNAL jump_addr_next      : std_logic_vector(7 DOWNTO 0);
   SIGNAL jump_addr_next_reg  : std_logic_vector(7 DOWNTO 0);
   SIGNAL jump_addr_reg       : std_logic_vector(7 DOWNTO 0);
   SIGNAL len_in              : std_logic_vector(15 DOWNTO 0);
   SIGNAL len_indicator       : std_logic;
   SIGNAL load_addr           : std_logic;
   SIGNAL new_valid           : std_logic;
   SIGNAL no_optional         : std_logic;
   SIGNAL not_empty           : std_logic;
   SIGNAL not_empty_late      : std_logic;
   SIGNAL packet_done         : std_logic;
   SIGNAL protocol_addr       : std_logic_vector(7 DOWNTO 0);
   SIGNAL protocol_indicator  : std_logic;
   SIGNAL read_en             : std_logic;
   SIGNAL valid               : std_logic;
   SIGNAL valid_byte          : std_logic;
   SIGNAL var_len             : std_logic_vector(15 DOWNTO 0);
   SIGNAL var_len_field       : std_logic;


   -- ModuleWare signal declarations(v1.0) for instance 'I0' of 'fifo'
   TYPE mw_I0sreg IS ARRAY (8 DOWNTO 0) OF std_logic_vector(7 DOWNTO 0);
   SIGNAL mw_I0caddr : INTEGER RANGE 0 TO 8;
   SIGNAL mw_I0naddr : INTEGER RANGE 0 TO 8;
   SIGNAL mw_I0creg : mw_I0sreg;
   SIGNAL mw_I0nreg : mw_I0sreg;
```

```
-- ModuleWare signal declarations(v1.0) for instance 'I4' of 'mux'
SIGNAL mw_I4din0 : std_logic_vector(7 DOWNTO 0);
SIGNAL mw_I4din1 : std_logic_vector(7 DOWNTO 0);

-- ModuleWare signal declarations(v1.0) for instance 'I7' of 'mux'
SIGNAL mw_I7din0 : std_logic_vector(15 DOWNTO 0);
SIGNAL mw_I7din1 : std_logic_vector(15 DOWNTO 0);

-- ModuleWare signal declarations(v1.0) for instance 'I8' of 'mux'
SIGNAL mw_I8din0 : std_logic_vector(7 DOWNTO 0);
SIGNAL mw_I8din1 : std_logic_vector(7 DOWNTO 0);

-- Component Declarations

-- This is the component declaration for the assembler entity, described in Appendix B.
COMPONENT assembler
PORT (
   byte_in          : IN     std_logic_vector (7 DOWNTO 0);
   clock            : IN     std_logic ;
   enable           : IN     std_logic ;
   len_in           : IN     std_logic_vector (15 DOWNTO 0);
   load_len         : IN     std_logic ;
   reset            : IN     std_logic ;
   valid_in         : IN     std_logic ;
   almost_done      : OUT    std_logic ;
   count_done       : OUT    std_logic ;
   field_data       : OUT    std_logic_vector (31 DOWNTO 0);
   field_data_early : OUT    std_logic_vector (31 DOWNTO 0);
   field_data_valid : OUT    std_logic ;
   new_valid        : OUT    std_logic ;
   valid_byte       : OUT    std_logic
);
END COMPONENT;

-- This is the component declaration for the jump_tlb entity, described in Appendix C.
COMPONENT jump_tlb
PORT (
   clock              : IN     std_logic ;
   enable             : IN     std_logic ;
   field_data         : IN     std_logic_vector (31 DOWNTO 0);
   protocol           : IN     std_logic ;
   protocol_addr      : IN     std_logic_vector (7 DOWNTO 0);
   reset              : IN     std_logic ;
   jump_addr_cur      : OUT    std_logic_vector (7 DOWNTO 0);
   jump_addr_next     : OUT    std_logic_vector (7 DOWNTO 0);
   jump_addr_next_reg : OUT    std_logic_vector (7 DOWNTO 0);
```

```
  jump_addr_reg      : OUT    std_logic_vector (7 DOWNTO 0)
);
END COMPONENT;

-- This is the component declaration for the length_block entity, described in Appendix D.
COMPONENT length_block
PORT (
  clock         : IN     std_logic ;
  enable        : IN     std_logic ;
  field_data    : IN     std_logic_vector (31 DOWNTO 0);
  length        : IN     std_logic ;
  protocol_addr : IN     std_logic_vector (7 DOWNTO 0);
  reset         : IN     std_logic ;
  no_optional   : OUT    std_logic ;
  var_len_reg   : OUT    std_logic_vector (15 DOWNTO 0)
);
END COMPONENT;

-- This is the component declaration for the protocol_memory entity, described in Appendix E.
COMPONENT protocol_memory
PORT (
  clock              : IN     std_logic ;
  enable             : IN     std_logic ;
  incr_addr          : IN     std_logic ;
  jump_addr          : IN     std_logic_vector (7 DOWNTO 0);
  jump_addr_next     : IN     std_logic_vector (7 DOWNTO 0);
  load_addr          : IN     std_logic ;
  reset              : IN     std_logic ;
  branch_indicator   : OUT    std_logic ;
  field_width        : OUT    std_logic_vector (15 DOWNTO 0);
  len_indicator      : OUT    std_logic ;
  packet_done        : OUT    std_logic ;
  protocol_addr      : OUT    std_logic_vector (7 DOWNTO 0);
  protocol_addr_reg  : OUT    std_logic_vector (7 DOWNTO 0);
  protocol_indicator : OUT    std_logic ;
  var_len_field      : OUT    std_logic
);
END COMPONENT;

-- Optional embedded configurations
-- pragma synthesis_off
FOR ALL : assembler USE ENTITY ProtocolAnalyzer.assembler;
FOR ALL : jump_tlb USE ENTITY ProtocolAnalyzer.jump_tlb;
FOR ALL : length_block USE ENTITY ProtocolAnalyzer.length_block;
FOR ALL : protocol_memory USE ENTITY ProtocolAnalyzer.protocol_memory;
-- pragma synthesis_on
```

```
BEGIN
  -- Architecture concurrent statements

  -- HDL Embedded Text Block 1 enable_block

  -- This block is primarily used to monitor the state of the
  -- enable signal. It is assumed that once a packet begins to arrive,
  -- it can be streamed through the FIFO. Once a packet is complete,
  -- enable is turned off until a new packet arrives. Upon new packet
  -- arrival, the FIFO must be enable a cycle before the rest of the
  -- design.

  process(reset,packet_done,line_in_valid,not_empty,not_empty_late)
  begin
    if(reset='1') then
      enable<='0';
      read_en<='0';
    elsif(packet_done='1' and line_in_valid='0' and not_empty='0') then
      enable<='0';
      read_en<='0';
    elsif(not_empty_late='1' OR field_type="00010110") then
      enable<='1';
      read_en<='1';
    elsif(not_empty='1') then
      enable<='0';
      read_en<='1';
    else
      enable<='0';
      read_en<='0';
    end if;
  end process;

  -- HDL Embedded Text Block 2 empty_reg

  -- This block assigns the value of not_empty to not_empty_late
  -- on a rising clock edge, so that not_empty_late will always equal
  -- what not_empty did a cycle earlier.

  process(reset,clock)
  begin
    if(reset='1') then
      not_empty_late<='0';
    elsif(rising_edge(clock)) then
      not_empty_late<=not_empty;
    end if;
  end process;

  -- HDL Embedded Text Block 3 output_reg

  -- This process registers output values on the rising clock edge.

  process(clock,reset)
  begin
    if(reset='1') then
      field_data_out<="00000000000000000000000000000000";
      field_type_out<="00000000";
      valid_out<='0';
    elsif(rising_edge(clock)) then
      field_data_out<=field_data;
      field_type_out<=field_type;
      valid_out<=valid;
    end if;
  end process;


  -- The code below is ModuleWare code for a FIFO.

  -- ModuleWare code(v1.0) for instance 'I6' of 'and'
  load_addr <= branch_indicator AND no_optional;

  -- ModuleWare code(v1.0) for instance 'I10' of 'and'
  valid <= field_data_valid AND new_valid AND enable;

  -- ModuleWare code(v1.0) for instance 'I0' of 'fifo'
  ffout <= mw_I0creg(0);
  I0seq1: PROCESS (clock)
  BEGIN
    IF (clock'EVENT AND clock='1') THEN
      FOR i IN 0 TO 8 LOOP
        mw_I0creg(i)(7 DOWNTO 0) <= mw_I0nreg(i)(7 DOWNTO 0);
      END LOOP;
    END IF;
  END PROCESS I0seq1;

  I0seq2: PROCESS (clock, reset)
  BEGIN
    IF (reset = '1') THEN
      mw_I0caddr <= 0;
    ELSIF (clock'EVENT AND clock='1') THEN
      mw_I0caddr <= mw_I0naddr;
    END IF;
  END PROCESS I0seq2;

  I0combo: PROCESS (reset, read_en, line_in_valid, mw_I0caddr, mw_I0creg, line_in)
  VARIABLE trena : std_logic;
  VARIABLE twena : std_logic;
  VARIABLE tfull : std_logic;
  VARIABLE tempty : std_logic;
  BEGIN
    IF (mw_I0caddr = 8) THEN
      tfull := '1';
      tempty := '0';
    ELSIF (mw_I0caddr = 0) THEN
      tfull := '0';
      tempty := '1';
    ELSE
      tfull := '0';
      tempty := '0';
    END IF;
    trena := NOT(reset) AND read_en AND NOT(tempty);
    twena := NOT(reset) AND line_in_valid AND NOT(tfull);

    IF (twena = '1' OR twena = 'H') THEN
      IF (trena = '1' OR trena = 'H') THEN
        mw_I0naddr <= mw_I0caddr;
      ELSE
        mw_I0naddr <= mw_I0caddr + 1;
      END IF;
    ELSIF (trena = '1' OR trena = 'H') THEN
      mw_I0naddr <= mw_I0caddr - 1;
    ELSE
      mw_I0naddr <= mw_I0caddr;
    END IF;

    IF (twena = '1' OR twena = 'H') THEN
      IF (trena = '1' OR trena = 'H') THEN
        mw_I0nreg(8)(7 DOWNTO 0) <= mw_I0creg(8)(7 DOWNTO 0);
        FOR i IN 0 TO 7 LOOP
          IF (mw_I0caddr = i) THEN
            mw_I0nreg(i)(7 DOWNTO 0) <= line_in;
          ELSE
            mw_I0nreg(i)(7 DOWNTO 0) <= mw_I0creg(i+1)(7 DOWNTO 0);
          END IF;
        END LOOP;
      ELSIF (trena = '0' OR trena = 'L') THEN
        mw_I0nreg(0)(7 DOWNTO 0) <= mw_I0creg(0)(7 DOWNTO 0);
        FOR i IN 0 TO 7 LOOP
          IF (mw_I0caddr = i) THEN
            mw_I0nreg(i+1)(7 DOWNTO 0) <= line_in;
          ELSE
            mw_I0nreg(i+1)(7 DOWNTO 0) <= mw_I0creg(i+1)(7 DOWNTO 0);
          END IF;
        END LOOP;
      END IF;
    ELSIF (twena = '0' OR twena = 'L') THEN
      IF (trena = '1' OR trena = 'H') THEN
        FOR i IN 0 TO 7 LOOP
          mw_I0nreg(i)(7 DOWNTO 0) <= mw_I0creg(i+1)(7 DOWNTO 0);
        END LOOP;
          mw_I0nreg(8)(7 DOWNTO 0) <= mw_I0creg(8)(7 DOWNTO 0);
      ELSIF (trena = '0' OR trena = 'L') THEN
        FOR i IN 0 TO 8 LOOP
          mw_I0nreg(i)(7 DOWNTO 0) <= mw_I0creg(i)(7 DOWNTO 0);
        END LOOP;
      ELSE
        FOR i IN 0 TO 8 LOOP
          mw_I0nreg(i)(7 DOWNTO 0) <= (OTHERS => 'X');
        END LOOP;
      END IF;
    ELSE
      FOR i IN 0 TO 8 LOOP
        mw_I0nreg(i)(7 DOWNTO 0) <= (OTHERS => 'X');
      END LOOP;
    END IF;
    not_empty <= NOT(tempty);
    aempty <= NOT(tempty);
  END PROCESS I0combo;

  -- ModuleWare code(v1.0) for instance 'I4' of 'mux'
  I4combo: PROCESS(mw_I4din0, mw_I4din1, protocol_indicator)
  VARIABLE dtemp : std_logic_vector(7 DOWNTO 0);
  BEGIN
    CASE protocol_indicator IS
    WHEN '0'|'L' => dtemp := mw_I4din0;
    WHEN '1'|'H' => dtemp := mw_I4din1;
    WHEN OTHERS => dtemp := (OTHERS => 'X');
    END CASE;
    jump_addr <= dtemp;
  END PROCESS I4combo;
  mw_I4din0 <= jump_addr_reg;
  mw_I4din1 <= jump_addr_cur;
```

```
-- ModuleWare code(v1.0) for instance 'I7' of 'mux'
I7combo: PROCESS(mw_I7din0, mw_I7din1, var_len_field)
VARIABLE dtemp : std_logic_vector(15 DOWNTO 0);
BEGIN
  CASE var_len_field IS
  WHEN '0'|'L' => dtemp := mw_I7din0;
  WHEN '1'|'H' => dtemp := mw_I7din1;
  WHEN OTHERS => dtemp := (OTHERS => 'X');
  END CASE;
  len_in <= dtemp;
END PROCESS I7combo;
mw_I7din0 <= field_width;
mw_I7din1 <= var_len;

-- ModuleWare code(v1.0) for instance 'I8' of 'mux'
I8combo: PROCESS(mw_I8din0, mw_I8din1, protocol_indicator)
VARIABLE dtemp : std_logic_vector(7 DOWNTO 0);
BEGIN
  CASE protocol_indicator IS
  WHEN '0'|'L' => dtemp := mw_I8din0;
  WHEN '1'|'H' => dtemp := mw_I8din1;
  WHEN OTHERS => dtemp := (OTHERS => 'X');
  END CASE;
  jump_addr_n <= dtemp;
END PROCESS I8combo;
mw_I8din0 <= jump_addr_next_reg;
mw_I8din1 <= jump_addr_next;

-- Instance port mappings.
I1 : assembler
  PORT MAP (
    byte_in          => ffout,
    clock            => clock,
    enable           => enable,
    len_in           => len_in,
    load_len         => count_done,
    reset            => reset,
    valid_in         => line_in_valid,
    almost_done      => almost_valid,
    count_done       => count_done,
    field_data       => field_data,
    field_data_early => field_data_early,
    field_data_valid => field_data_valid,
    new_valid        => new_valid,
    valid_byte       => valid_byte
  );
  I3 : jump_tlb
    PORT MAP (
      clock             => clock,
      enable            => enable,
      field_data        => field_data_early,
      protocol          => protocol_indicator,
      protocol_addr     => protocol_addr,
      reset             => reset,
      jump_addr_cur     => jump_addr_cur,
      jump_addr_next    => jump_addr_next,
      jump_addr_next_reg => jump_addr_next_reg,
      jump_addr_reg     => jump_addr_reg
    );
  I5 : length_block
    PORT MAP (
      clock        => clock,
      enable       => enable,
      field_data   => field_data_early,
      length       => len_indicator,
      protocol_addr => protocol_addr,
      reset        => reset,
      no_optional  => no_optional,
      var_len_reg  => var_len
    );
  I2 : protocol_memory
    PORT MAP (
      clock             => clock,
      enable            => enable,
      incr_addr         => almost_valid,
      jump_addr         => jump_addr,
      jump_addr_next    => jump_addr_n,
      load_addr         => load_addr,
      reset             => reset,
      branch_indicator  => branch_indicator,
      field_width       => field_width,
      len_indicator     => len_indicator,
      packet_done       => packet_done,
      protocol_addr     => protocol_addr,
      protocol_addr_reg  => field_type,
      protocol_indicator => protocol_indicator,
      var_len_field     => var_len_field
    );

END struct;
```

APPENDIX B

VHDL CODE FOR THE ASSEMBLER ENTITY

This appendix contains the VHDL code for the *assembler* entity of ePAPP. As described in Section 3.3, this component is responsible for receiving bytes from a FIFO and assembling them into packet fields as specified by the *protocol_memory*. Specific details about the code can be found as comments within the code itself.

```
-- VHDL Entity ProtocolAnalyzer.assembler.symbol
-- Created by Jacob J. Repanshek
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY assembler IS
  PORT(
    byte_in          : IN     std_logic_vector (7 DOWNTO 0);
    clock            : IN     std_logic;
    enable           : IN     std_logic;
    len_in           : IN     std_logic_vector (15 DOWNTO 0);
    load_len         : IN     std_logic;
    reset            : IN     std_logic;
    valid_in         : IN     std_logic;
    almost_done      : OUT    std_logic;
    count_done       : OUT    std_logic;
    field_data       : OUT    std_logic_vector (31 DOWNTO 0);
    field_data_early : OUT    std_logic_vector (31 DOWNTO 0);
    field_data_valid : OUT    std_logic;
    new_valid        : OUT    std_logic;
    valid_byte       : OUT    std_logic
  );

END assembler ;

-- VHDL Architecture ProtocolAnalyzer.assembler.struct
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;


ARCHITECTURE struct OF assembler IS

   -- Architecture declarations

   -- Internal signal declarations
   SIGNAL count       : std_logic_vector(15 DOWNTO 0);
   SIGNAL data_temp   : std_logic_vector(31 DOWNTO 0);
   SIGNAL full        : std_logic;
   SIGNAL len_delayed : std_logic_vector(15 DOWNTO 0);
   SIGNAL max         : std_logic;
   SIGNAL q0          : std_logic_vector(7 DOWNTO 0);
   SIGNAL q1          : std_logic_vector(7 DOWNTO 0);
   SIGNAL q2          : std_logic_vector(7 DOWNTO 0);
   SIGNAL q3          : std_logic_vector(7 DOWNTO 0);
   SIGNAL valid1      : std_logic;
   SIGNAL valid2      : std_logic;
   SIGNAL valid3      : std_logic;

   -- Implicit buffer signal declarations
   SIGNAL count_done_internal : std_logic;


   -- ModuleWare signal declarations(v1.0) for instance 'I5' of 'cntr'
   SIGNAL mw_I5n_cnt : std_logic_vector(15 DOWNTO 0);
   SIGNAL mw_I5c_cnt : std_logic_vector(15 DOWNTO 0);


BEGIN
   -- Architecture concurrent statements

   -- HDL Embedded Text Block 1 data_out_select

   -- This block determines how to output field_data based on the
   -- specified width of the protocol field. This is done by observing
   -- the value on the count bus. If more than four bytes remain,
   -- a payload or extended options field is being output and four bytes
   -- of data are valid. If less than four bytes are specified, that number
   -- of bytes are put on the field_data bus, padded with zeros accordingly.
```

```
process(len_delayed, data_temp, count)
begin
  if(count(15 downto 2)="00000000000000" AND
    not(count(1)='1' AND len_delayed(1)='0') AND
    not(count(1 downto 0)="11" AND len_delayed(1 downto 0)="10") AND
    not(count(1 downto 0)="01" AND len_delayed(1 downto 0)="00")) then
    case len_delayed(1 downto 0) is
      -- field length is 1 byte
      when "00" =>
        field_data <= "000000000000000000000000" & data_temp(7 downto 0);
      -- field length is 2 bytes
      when "01" =>
        field_data <= "0000000000000000" & data_temp(15 downto 0);
      -- field length is 3 bytes
      when "10" =>
        field_data <= "00000000" & data_temp(23 downto 0);
      -- field length is 4 bytes
      when "11" =>
        field_data <= data_temp;
      when others =>
        field_data <= data_temp;
    end case;
  else
    -- remaining field length is more than 4 bytes -> indicates payload or optional field
    field_data <= data_temp;
  end if;
end process;

-- The “reg” blocks that follow are a chain of registers that hold
-- the bytes incoming from the FIFO. When the registers are enabled,
-- value from the FIFO is read into the first register and all of the
-- register values are shifted one register to the right.

-- HDL Embedded Text Block 2 reg
process(reset,clock)
begin
  if(reset='1') then
    q3<="00000000";
    valid3<='0';
  elsif (rising_edge(clock)) then
    if(enable='1') then
      q3<=byte_in;
      valid3<=valid_in;
    end if;
  end if;
end process;
```

```
-- HDL Embedded Text Block 3 reg1
process(reset,clock)
begin
  if(reset='1') then
    q2<="00000000";
    valid2<='0';
  elsif (rising_edge(clock)) then
    if(enable='1') then
      q2<=q3;
      valid2<=valid3;
    end if;
  end if;
end process;

-- HDL Embedded Text Block 4 reg2
process(reset,clock)
begin
  if(reset='1') then
    q1<="00000000";
    valid1<='0';
  elsif (rising_edge(clock)) then
    if(enable='1') then
      q1<=q2;
      valid1<=valid2;
    end if;
  end if;
end process;

-- HDL Embedded Text Block 5 reg3
process(reset,clock)
begin
  if(reset='1') then
    q0<="00000000";
    valid_byte<='0';
  elsif (rising_edge(clock)) then
    if(enable='1') then
      q0<=q1;
      valid_byte<=valid1;
    end if;
  end if;
end process;

-- HDL Embedded Text Block 6 len_delay
-- The len_delayed signal is always equal to the value of len_in the cycle before.

process(reset,clock)
begin
  if(reset='1') then
    len_delayed<="0000000000000000";
  elsif (rising_edge(clock)) then
    if(enable='1') then
      len_delayed<=len_in;
    end if;
  end if;
end process;

-- HDL Embedded Text Block 7 almost

-- If a packet field is one cycle from being complete, almost_done is asserted.

process(count,len_in,load_len)
begin
  if(count="0000000000000001" OR (len_in="0000000000000000" and load_len='1')) then
    almost_done<='1';
  else
    almost_done<='0';
  end if;
end process;

new_valid<=valid1;

-- HDL Embedded Text Block 10 full_count

-- For fields of more than 32 bits, there must be
-- a way to indicate that a 32-bit chunk is ready to
-- be put in the output FIFO. That is what this block
-- does.
--
process(clock,reset)
variable cnt : std_logic_vector(1 downto 0);
variable go : std_logic;
begin
  if(reset='1') then
    full <= '0';
    cnt := "11";
    go := '0';
  elsif(rising_edge(clock)) then
    if(enable='1') then
```

```
         if(load_len='1') then
            cnt := "10";
            full<='0';
            go := '1';
         elsif(go='1') then
            if(cnt = "00") then
               full<='1';
            else
               full<='0';
            end if;
            if(cnt = "00") then
               cnt := "11";
            else
               cnt := (unsigned(cnt) - '1');
            end if;
         end if;
      end if;
   end if;
end process;


-- All of the code below is inserted by the use of ModuleWare components.

-- ModuleWare code(v1.0) for instance 'I0' of 'and'
count_done_internal <= max AND enable;

-- ModuleWare code(v1.0) for instance 'I5' of 'cntr'
count <= mw_I5c_cnt;
I5clock: PROCESS (clock, reset, enable, mw_I5n_cnt)
BEGIN
   IF (reset = '1' OR reset = 'H') THEN
      mw_I5c_cnt <= "0000000000000100";
   ELSIF (clock'EVENT AND clock='1') THEN
      IF (enable = '1' OR enable = 'H') THEN
         mw_I5c_cnt <= mw_I5n_cnt;
      END IF;
   END IF;
END PROCESS I5clock;
I5combo: PROCESS (load_len, len_in, mw_I5c_cnt)
BEGIN
IF (load_len = '1' OR load_len = 'H') THEN
   mw_I5n_cnt <= len_in;
ELSE
   IF (mw_I5c_cnt = "0000000000000000") THEN
   mw_I5n_cnt <= "1111111111111111";
   ELSE
```

```
        mw_I5n_cnt <= (unsigned(mw_I5c_cnt) - '1');
      END IF;
   END IF;
   END PROCESS I5combo;
   I5max_drive: PROCESS (mw_I5c_cnt)
   VARIABLE temp : std_logic;
   BEGIN
   temp := '0';
   IF (mw_I5c_cnt = "0000000000000000") THEN
      temp := '1';
   END IF;
   max <= temp;
   END PROCESS I5max_drive;

   -- ModuleWare code(v1.0) for instance 'I4' of 'merge'
   data_temp <= q0 & q1 & q2 & q3;

   -- ModuleWare code(v1.0) for instance 'I6' of 'merge'
   field_data_early <= q1 & q2 & q3 & byte_in;

   -- ModuleWare code(v1.0) for instance 'I1' of 'or'
   field_data_valid <= full OR count_done_internal;

   -- Instance port mappings.

   -- Implicit buffered output assignments
   count_done <= count_done_internal;

END struct;
```

APPENDIX C

VHDL CODE FOR THE JUMP_TLB ENTITY

This appendix contains the VHDL code for the *jump_tlb* entity of ePAPP. As described in Section 3.2.2, this component is responsible for determining which higher-layer protocol is encapsulated by the current protocol layer and to inform the *protocol_memory* to branch accordingly. Specific details about the code can be found as comments within the code itself.

```
-- VHDL Entity ProtocolAnalyzer.jump_tlb.symbol
-- Created by Jacob J. Repanshek
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY jump_tlb IS
  PORT(
    clock             : IN     std_logic;
    enable            : IN     std_logic;
    field_data        : IN     std_logic_vector (31 DOWNTO 0);
    protocol          : IN     std_logic;
    protocol_addr     : IN     std_logic_vector (7 DOWNTO 0);
    reset             : IN     std_logic;
    jump_addr_cur     : OUT    std_logic_vector (7 DOWNTO 0);
    jump_addr_next    : OUT    std_logic_vector (7 DOWNTO 0);
    jump_addr_next_reg : OUT   std_logic_vector (7 DOWNTO 0);
    jump_addr_reg     : OUT    std_logic_vector (7 DOWNTO 0)
  );

END jump_tlb ;

-- VHDL Architecture ProtocolAnalyzer.jump_tlb.struct
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;


ARCHITECTURE struct OF jump_tlb IS

   -- Architecture declarations

   -- Internal signal declarations
   SIGNAL jump_addr   : std_logic_vector(7 DOWNTO 0);
   SIGNAL jump_addr_n : std_logic_vector(7 DOWNTO 0);

BEGIN
   -- Architecture concurrent statements

   -- HDL Embedded Block 1 jump_table
   -- Non hierarchical truthtable

   -- This truth table specifies the protocol_memory address to jump
   -- to based on information found within a field of the current packet.
   -- If certain data is found on the field_data bus while a certain field type
   -- is being specified by the protocol_memory, jump_addr and jump_addr_n
   -- are assigned. jump_addr_n is always jump_addr plus one.

   ---------------------------------------------------------------------------
   jump_table_truth_process: PROCESS(field_data, protocol_addr)
   ---------------------------------------------------------------------------
   BEGIN
      -- Block 1
      IF (field_data(15 downto 8) = "00001000") AND (field_data(7 downto 0) = "00000000")
AND (protocol_addr = "00000110") THEN
         jump_addr <= "00011000";
         jump_addr_n <= "00011001";
      ELSIF (field_data(15 downto 8) = "00001000") AND (field_data(7 downto 0) =
"00000110") AND (protocol_addr = "00000110") THEN
         jump_addr <= "00000111";
         jump_addr_n <= "00001000";
      ELSIF (field_data(7 downto 0) = "00000110") AND (protocol_addr = "00011110") THEN
         jump_addr <= "00100011";
         jump_addr_n <= "00100100";
      ELSIF (field_data(7 downto 0) = "00010001") AND (protocol_addr = "00011110") THEN
         jump_addr <= "00101101";
         jump_addr_n <= "00101110";
      ELSIF (field_data(7 downto 0) = "00000001") AND (protocol_addr = "00011110") THEN
         jump_addr <= "00110010";
         jump_addr_n <= "00110011";
      ELSIF (protocol_addr = "00101010") THEN
```

```
  jump_addr <= "00101100";
  jump_addr_n <= "00101101";
ELSIF (field_data(7 downto 0) = "00000000") AND (protocol_addr = "00110010") THEN
  jump_addr <= "00110101";
  jump_addr_n <= "00110110";
ELSIF (field_data(7 downto 0) = "00001000") AND (protocol_addr = "00110010") THEN
  jump_addr <= "00110101";
  jump_addr_n <= "00110110";
ELSIF (field_data(7 downto 0) = "00000011") AND (protocol_addr = "00110010") THEN
  jump_addr <= "00111000";
  jump_addr_n <= "00111001";
ELSIF (field_data(7 downto 0) = "00000100") AND (protocol_addr = "00110010") THEN
  jump_addr <= "00111000";
  jump_addr_n <= "00111001";
ELSIF (field_data(7 downto 0) = "00001011") AND (protocol_addr = "00110010") THEN
  jump_addr <= "00111000";
  jump_addr_n <= "00111001";
ELSIF (field_data(7 downto 0) = "00000101") AND (protocol_addr = "00110010") THEN
  jump_addr <= "00111011";
  jump_addr_n <= "00111100";
ELSIF (field_data(7 downto 0) = "00001001") AND (protocol_addr = "00110010") THEN
  jump_addr <= "00111101";
  jump_addr_n <= "00111110";
ELSIF (field_data(7 downto 0) = "00001010") AND (protocol_addr = "00110010") THEN
  jump_addr <= "01000001";
  jump_addr_n <= "01000010";
ELSIF (field_data(7 downto 0) = "00001100") AND (protocol_addr = "00110010") THEN
  jump_addr <= "01000010";
  jump_addr_n <= "01000011";
ELSIF (field_data(7 downto 0) = "00001101") AND (protocol_addr = "00110010") THEN
  jump_addr <= "01000101";
  jump_addr_n <= "01000110";
ELSIF (field_data(7 downto 0) = "00001110") AND (protocol_addr = "00110010") THEN
  jump_addr <= "01000101";
  jump_addr_n <= "01000110";
ELSIF (field_data(7 downto 0) = "00001111") AND (protocol_addr = "00110010") THEN
  jump_addr <= "01001010";
  jump_addr_n <= "01001011";
ELSIF (field_data(7 downto 0) = "00010000") AND (protocol_addr = "00110010") THEN
  jump_addr <= "01001010";
  jump_addr_n <= "01001011";
ELSIF (field_data(7 downto 0) = "00010001") AND (protocol_addr = "00110010") THEN
  jump_addr <= "01001100";
  jump_addr_n <= "01001101";
ELSIF (field_data(7 downto 0) = "00010010") AND (protocol_addr = "00110010") THEN
  jump_addr <= "01001100";
```

```
      jump_addr_n <= "01001101";
    ELSIF (protocol_addr = "00101100") THEN
      jump_addr <= "00010110";
      jump_addr_n <= "00010111";
    ELSIF (protocol_addr = "00110001") THEN
      jump_addr <= "00010110";
      jump_addr_n <= "00010111";
    ELSE
      jump_addr <= "00000000";
      jump_addr_n <= "00000001";
    END IF;

  END PROCESS jump_table_truth_process;


  -- HDL Embedded Text Block 2 jump_addr_reg

  -- This block registers the jump_addr and jump_addr_n signals
  -- set in the above truth table, so that they can be used when a
  -- branch is to occur.

  jump_addr_cur<=jump_addr;
  jump_addr_next<=jump_addr_n;
  process(reset,clock)
  begin
    if(reset='1') then
      jump_addr_reg<="00000000";
    elsif (rising_edge(clock)) then
      if(enable='1' AND protocol='1') then
        jump_addr_reg<=jump_addr;
        jump_addr_next_reg<=jump_addr_n;
      end if;
    end if;
  end process;


END struct;
```

APPENDIX D

VHDL CODE FOR THE LENGTH_BLOCK ENTITY

This appendix contains the VHDL code for the *length_block* entity of ePAPP. As described in Section 3.2.3, this component is responsible for calculating dynamic field lengths based on information contained within packet header fields (i.e. payload lengths). Specific details about the code can be found as comments within the code itself.

```
-- VHDL Entity ProtocolAnalyzer.length_block.symbol
-- Created by Jacob J. Repanshek
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY length_block IS
  PORT(
    clock         : IN     std_logic;
    enable        : IN     std_logic;
    field_data    : IN     std_logic_vector (31 DOWNTO 0);
    length        : IN     std_logic;
    protocol_addr : IN     std_logic_vector (7 DOWNTO 0);
    reset         : IN     std_logic;
    no_optional   : OUT    std_logic;
    var_len_reg   : OUT    std_logic_vector (15 DOWNTO 0)
  );

END length_block ;

-- VHDL Architecture ProtocolAnalyzer.length_block.struct
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
```

```
ARCHITECTURE struct OF length_block IS

   -- Architecture declarations

   -- Internal signal declarations
   SIGNAL IP_header_len     : std_logic_vector(5 DOWNTO 0);
   SIGNAL IP_header_len_out : std_logic_vector(5 DOWNTO 0);
   SIGNAL IP_header_len_reg : std_logic_vector(5 DOWNTO 0);
   SIGNAL IP_len_reg        : std_logic_vector(15 DOWNTO 0);
   SIGNAL IP_len_tmp        : std_logic_vector(15 DOWNTO 0);
   SIGNAL a                 : std_logic_vector(15 DOWNTO 0);
   SIGNAL a_out             : std_logic_vector(15 DOWNTO 0);
   SIGNAL b                 : std_logic_vector(15 DOWNTO 0);
   SIGNAL b_out             : std_logic_vector(15 DOWNTO 0);
   SIGNAL minus             : std_logic;
   SIGNAL minus_out         : std_logic;
   SIGNAL not_zero          : std_logic;
   SIGNAL opt               : std_logic;
   SIGNAL opt_next          : std_logic;
   SIGNAL opt_next_out      : std_logic;
   SIGNAL opt_out           : std_logic;
   SIGNAL opt_reg           : std_logic;
   SIGNAL var_len           : std_logic_vector(15 DOWNTO 0);
   SIGNAL var_len_out       : std_logic_vector(15 DOWNTO 0);
   SIGNAL var_len_reg_tmp   : std_logic_vector(15 DOWNTO 0);

BEGIN
   -- Architecture concurrent statements

   -- HDL Embedded Text Block 1 len_calc

   -- The protocol_addr bus indicates what packet field is currently coming
   -- in on field_data. If the field contains information that will be used to
   -- calculate a dynamic field length, the information is registered and/or used
   -- in calculations accordingly.

   process(field_data, protocol_addr)
   begin
      case protocol_addr is
         when "00011000" =>
            -- subtract 5 from IP header length (in 32 bit words, so add "00")
            var_len <= "0000000000"&(unsigned(field_data(3 downto 0)) - 5)&"00";
            IP_header_len <= field_data(3 downto 0)&"00";
            opt <= '1';
            opt_next <= '0';
```

```
  a <= a_out;
  b <= b_out;
  minus <= minus_out;
when "00011010" =>
  -- subtract IP header length from total IP packet length
  a <= field_data(15 downto 0);
  b <= "0000000000"&IP_header_len_reg;
  minus <= '1';
  var_len <= var_len_out;
  IP_header_len <= IP_header_len_out;
  opt <= opt_out;
  opt_next <= opt_next_out;
when "00100001" =>
  -- optional IP field is next
  var_len <= "0000000000000000";
  opt <= '0';
  opt_next <= '1';
  IP_header_len <= IP_header_len_out;
  a <= a_out;
  b <= b_out;
  minus <= minus_out;
when "00100111" =>
  -- subtract 5 from TCP header length (in 32 bits words, so add "00")
  var_len <= "0000000000"&(unsigned(field_data(15 downto 12)) - 5)&"00";
  a <= IP_len_reg;
  b <= "0000000000"&field_data(15 downto 12)&"00";
  minus <= '1';
  opt <= '1';
  opt_next <= '0';
  IP_header_len <= IP_header_len_out;
when "00101010" =>
  -- optional TCP field is next
  var_len <= "0000000000000000";
  opt <= '0';
  opt_next <= '1';
  IP_header_len <= IP_header_len_out;
  a <= a_out;
  b <= b_out;
  minus <= minus_out;
when "00101111" =>
  -- subtract 8 from the UDP length (in bytes)
  var_len <= "00000000"&(unsigned(field_data(7 downto 0)) - 8);
  opt <= '0';
  opt_next <= '0';
  IP_header_len <= IP_header_len_out;
  a <= a_out;
```

```
        b <= b_out;
        minus <= minus_out;
      when others =>
        var_len <= "0000000000000000";
        opt <= '0';
        opt_next <= '0';
        minus <= '0';
        IP_header_len <= IP_header_len_out;
        a <= a_out;
        b <= b_out;
  end case;
end process;

process(clock,reset)
begin
  if(rising_edge(clock)) then
    a_out<=a;
    b_out<=b;
    minus_out<=minus;
    IP_header_len_out<=IP_header_len;
    opt_out<=opt;
    opt_next_out<=opt_next;
    var_len_out<=var_len;
  end if;
end process;

-- HDL Embedded Text Block 2 length_reg

-- This process registers a dynamic field width value to be output
-- from length_block. Also, this block identifies whether the dynamic
-- length is a non-zero value.

process(reset,clock)
begin
  if(reset='1') then
    var_len_reg_tmp<="0000000000000000";
    not_zero<='0';
    opt_reg<='0';
  elsif (rising_edge(clock)) then
    if(enable='1' AND length='1' and protocol_addr/="00011010") then
      var_len_reg_tmp<=(unsigned(var_len)-'1');
      opt_reg<=opt;
      if(var_len/="0000000000000000") then
        not_zero<='1';
      else
        not_zero<='0';
      end if;
    end if;
  end if;
end process;

-- HDL Embedded Text Block 3 rename

-- This blocks specifies the output of length_block.

process(protocol_addr)
begin
  if(protocol_addr="00101100") then
    var_len_reg<=(unsigned(IP_len_tmp)-1);
  else
    var_len_reg<=var_len_reg_tmp;
  end if;
end process;

-- HDL Embedded Text Block 4 IP_len_reg

-- This process registers the length of the IP portion of packet.

process(reset,clock)
begin
  if(reset='1') then
    IP_len_reg<="0000000000000000";
  elsif (rising_edge(clock)) then
    if(enable='1' AND protocol_addr="00011011") then
      IP_len_reg<=IP_len_tmp;
    end if;
  end if;
end process;

-- HDL Embedded Text Block 5 IP_header_len

-- This process registers the value of the IP header length.

process(reset,clock)
begin
  if(reset='1') then
    IP_header_len_reg<="000000";
  elsif (rising_edge(clock)) then
    if(enable='1' AND length='1') then
      IP_header_len_reg<=IP_header_len;
    end if;
  end if;
```

```
   end process;

   -- HDL Embedded Text Block 6 subtractor

   -- This block performs subtraction on two 16-bit inputs.

   process(reset,clock)
   begin
      if(reset='1') then
         IP_len_tmp <= "0000000000000000";
      elsif(rising_edge(clock)) then
         if(minus='1') then
            IP_len_tmp <= (unsigned(a) - unsigned(b));
         end if;
      end if;
   end process;


   -- ModuleWare code(v1.0) for instance 'I0' of 'nand'
   no_optional <= NOT(not_zero AND opt_reg AND opt_next);

   -- Instance port mappings.

END struct;
```

APPENDIX E

VHDL CODE FOR THE PROTOCOL_MEMORY ENTITY

This appendix contains the VHDL code for the *protocol_memory* entity of ePAPP. As described in Section 3.2.1, this component keeps track of the current part of a packet being classified and uses a truth table to determine what field type is next. Specific details about the code can be found as comments within the code itself.

```
-- VHDL Entity ProtocolAnalyzer.protocol_memory.symbol
-- Created by Jacob J. Repanshek
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY protocol_memory IS
  PORT(
    clock              : IN     std_logic;
    enable             : IN     std_logic;
    incr_addr          : IN     std_logic;
    jump_addr          : IN     std_logic_vector (7 DOWNTO 0);
    jump_addr_next     : IN     std_logic_vector (7 DOWNTO 0);
    load_addr          : IN     std_logic;
    reset              : IN     std_logic;
    branch_indicator   : OUT    std_logic;
    field_width        : OUT    std_logic_vector (15 DOWNTO 0);
    len_indicator      : OUT    std_logic;
    packet_done        : OUT    std_logic;
    protocol_addr      : OUT    std_logic_vector (7 DOWNTO 0);
    protocol_addr_reg  : OUT    std_logic_vector (7 DOWNTO 0);
    protocol_indicator : OUT    std_logic;
    var_len_field      : OUT    std_logic
  );

END protocol_memory ;
```

```
-- VHDL Architecture ProtocolAnalyzer.protocol_memory.struct
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ARCHITECTURE struct OF protocol_memory IS

   -- Architecture declarations

   -- Internal signal declarations
   SIGNAL clk_en : std_logic;
   SIGNAL protocol_addr_int : std_logic_vector(7 DOWNTO 0);


BEGIN
   -- Architecture concurrent statements
   -- HDL Embedded Block 1 protocol_table
   -- Non hierarchical truthtable

   -- This is the truth table which contains information about all
   -- of the protocol fields supported by ePAPP. This information is
   -- used not only to identify the current part of a packet being read in,
   -- but also the width of the packet field and whether or not it contains
   -- information useful to classifying future fields, such as length information.

   ---------------------------------------------------------------------------
   protocol_table_truth_process: PROCESS(protocol_addr_int)
   ---------------------------------------------------------------------------
   BEGIN
      -- Block 1
      CASE protocol_addr_int IS
      WHEN "00000000" =>
         branch_indicator <= '0';
         field_width <= "0000000000000011";
         len_indicator <= '0';
         var_len_field <= '0';
         protocol_indicator <= '0';
         packet_done <= '1';
      WHEN "00000001" =>
         branch_indicator <= '0';
         field_width <= "0000000000000011";
         len_indicator <= '0';
         var_len_field <= '0';
```

```
  protocol_indicator <= '0';
  packet_done <= '0';
WHEN "00000010" =>
  branch_indicator <= '0';
  field_width <= "0000000000000011";
  len_indicator <= '0';
  var_len_field <= '0';
  protocol_indicator <= '0';
  packet_done <= '0';
WHEN "00000011" =>
  branch_indicator <= '0';
  field_width <= "0000000000000001";
  len_indicator <= '0';
  var_len_field <= '0';
  protocol_indicator <= '0';
  packet_done <= '0';
WHEN "00000100" =>
  branch_indicator <= '0';
  field_width <= "0000000000000011";
  len_indicator <= '0';
  var_len_field <= '0';
  protocol_indicator <= '0';
  packet_done <= '0';
WHEN "00000101" =>
  branch_indicator <= '0';
  field_width <= "0000000000000001";
  len_indicator <= '0';
  var_len_field <= '0';
  protocol_indicator <= '0';
  packet_done <= '0';
WHEN "00000110" =>
  branch_indicator <= '1';
  field_width <= "0000000000000001";
  len_indicator <= '0';
  var_len_field <= '0';
  protocol_indicator <= '1';
  packet_done <= '0';
WHEN "00000111" =>
  branch_indicator <= '0';
  field_width <= "0000000000000001";
  len_indicator <= '0';
  var_len_field <= '0';
  protocol_indicator <= '0';
  packet_done <= '0';
WHEN "00001000" =>
  branch_indicator <= '0';
```

```
    field_width <= "0000000000000001";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00001001" =>
    branch_indicator <= '0';
    field_width <= "0000000000000000";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00001010" =>
    branch_indicator <= '0';
    field_width <= "0000000000000000";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00001011" =>
    branch_indicator <= '0';
    field_width <= "0000000000000001";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00001100" =>
    branch_indicator <= '0';
    field_width <= "0000000000000011";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00001101" =>
    branch_indicator <= '0';
    field_width <= "0000000000000001";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00001110" =>
    branch_indicator <= '0';
    field_width <= "0000000000000011";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00001111" =>
    branch_indicator <= '0';
    field_width <= "0000000000000011";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00010000" =>
    branch_indicator <= '0';
    field_width <= "0000000000000001";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00010001" =>
    branch_indicator <= '0';
    field_width <= "0000000000000011";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00010010" =>
    branch_indicator <= '0';
    field_width <= "0000000000000011";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00010011" =>
    branch_indicator <= '0';
    field_width <= "0000000000000011";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00010100" =>
    branch_indicator <= '0';
    field_width <= "0000000000000011";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00010101" =>
    branch_indicator <= '0';
    field_width <= "0000000000000001";
```

```
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00010110" =>
    branch_indicator <= '0';
    field_width <= "0000000000000011";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00010111" =>
    branch_indicator <= '1';
    field_width <= "0000000000000000";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '1';
    packet_done <= '0';
  WHEN "00011000" =>
    branch_indicator <= '0';
    field_width <= "0000000000000000";
    len_indicator <= '1';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00011001" =>
    branch_indicator <= '0';
    field_width <= "0000000000000000";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00011010" =>
    branch_indicator <= '0';
    field_width <= "0000000000000001";
    len_indicator <= '1';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00011011" =>
    branch_indicator <= '0';
    field_width <= "0000000000000001";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
```

```
WHEN "00011100" =>
  branch_indicator <= '0';
  field_width <= "0000000000000001";
  len_indicator <= '0';
  var_len_field <= '0';
  protocol_indicator <= '0';
  packet_done <= '0';
WHEN "00011101" =>
  branch_indicator <= '0';
  field_width <= "0000000000000000";
  len_indicator <= '0';
  var_len_field <= '0';
  protocol_indicator <= '0';
  packet_done <= '0';
WHEN "00011110" =>
  branch_indicator <= '0';
  field_width <= "0000000000000000";
  len_indicator <= '0';
  var_len_field <= '0';
  protocol_indicator <= '1';
  packet_done <= '0';
WHEN "00011111" =>
  branch_indicator <= '0';
  field_width <= "0000000000000001";
  len_indicator <= '0';
  var_len_field <= '0';
  protocol_indicator <= '0';
  packet_done <= '0';
WHEN "00100000" =>
  branch_indicator <= '0';
  field_width <= "0000000000000011";
  len_indicator <= '0';
  var_len_field <= '0';
  protocol_indicator <= '0';
  packet_done <= '0';
WHEN "00100001" =>
  branch_indicator <= '1';
  field_width <= "0000000000000011";
  len_indicator <= '0';
  var_len_field <= '0';
  protocol_indicator <= '0';
  packet_done <= '0';
WHEN "00100010" =>
  branch_indicator <= '1';
  field_width <= "1111111111111111";
  len_indicator <= '0';
```

```
    var_len_field <= '1';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00100011" =>
    branch_indicator <= '0';
    field_width <= "0000000000000001";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00100100" =>
    branch_indicator <= '0';
    field_width <= "0000000000000001";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00100101" =>
    branch_indicator <= '0';
    field_width <= "0000000000000011";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00100110" =>
    branch_indicator <= '0';
    field_width <= "0000000000000011";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00100111" =>
    branch_indicator <= '0';
    field_width <= "0000000000000001";
    len_indicator <= '1';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00101000" =>
    branch_indicator <= '0';
    field_width <= "0000000000000001";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00101001" =>
```

```
    branch_indicator <= '0';
    field_width <= "0000000000000001";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00101010" =>
    branch_indicator <= '1';
    field_width <= "0000000000000001";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '1';
    packet_done <= '0';
  WHEN "00101011" =>
    branch_indicator <= '0';
    field_width <= "1111111111111111";
    len_indicator <= '0';
    var_len_field <= '1';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00101100" =>
    branch_indicator <= '1';
    field_width <= "1111111111111111";
    len_indicator <= '0';
    var_len_field <= '1';
    protocol_indicator <= '1';
    packet_done <= '0';
  WHEN "00101101" =>
    branch_indicator <= '0';
    field_width <= "0000000000000001";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00101110" =>
    branch_indicator <= '0';
    field_width <= "0000000000000001";
    len_indicator <= '0';
    var_len_field <= '0';
    protocol_indicator <= '0';
    packet_done <= '0';
  WHEN "00101111" =>
    branch_indicator <= '0';
    field_width <= "0000000000000001";
    len_indicator <= '1';
    var_len_field <= '0';
```

```
      protocol_indicator <= '0';
      packet_done <= '0';
   WHEN "00110000" =>
      branch_indicator <= '0';
      field_width <= "0000000000000001";
      len_indicator <= '0';
      var_len_field <= '0';
      protocol_indicator <= '0';
      packet_done <= '0';
   WHEN "00110001" =>
      branch_indicator <= '1';
      field_width <= "1111111111111111";
      len_indicator <= '0';
      var_len_field <= '1';
      protocol_indicator <= '1';
      packet_done <= '0';
   WHEN OTHERS =>
      branch_indicator <= '0';
      field_width <= "1111111111111111";
      len_indicator <= '0';
      var_len_field <= '0';
      protocol_indicator <= '0';
      packet_done <= '0';
   END CASE;

END PROCESS protocol_table_truth_process;

-- Architecture concurrent statements

-- HDL Embedded Text Block 2 addr_reg

-- This process registers the current protocol address.

protocol_addr<=protocol_addr_int;
process(reset,clock)
begin
   if(reset='1') then
      protocol_addr_reg<="00000000";
   elsif(rising_edge(clock)) then
      if(enable='1') then
         protocol_addr_reg<=protocol_addr_int;
      end if;
   end if;
end process;

-- HDL Embedded Text Block 3 counter
```

```
   -- This process keeps a loadable counter that keeps track of
   -- the current location in the protocol_memory. A branch is enabled
   -- by loading a value from jump_tlb into the counter.

   process(clock,reset)
   variable cnt : std_logic_vector(7 downto 0);
   begin
      if(reset='1') then
         protocol_addr_int <= "00000000";
         cnt := "00000001";
      elsif(rising_edge(clock)) then
         if(clk_en='1') then
            if(load_addr='1') then
               protocol_addr_int <= jump_addr;
               cnt := jump_addr_next;
            else
               protocol_addr_int <= cnt;
               if(cnt = "11111111") then
                  cnt := "00000000";
               else
                  cnt := (unsigned(cnt) + '1');
               end if;
            end if;
         end if;
      end if;
   end process;


   -- ModuleWare code(v1.0) for instance 'I1' of 'and'
   clk_en <= enable AND incr_addr;

   -- Instance port mappings.

END struct;
```

APPENDIX F

VHDL CODE FOR THE SNORT_CAM ENTITY

This appendix contains the VHDL code for the *snort_cam* entity of CASMA. This entity contains all of the VHDL code responsible for everything from receiving data from ePAPP to sending data to the CAM to retrieving results from the CAM and outputting them. Specific details about the code can be found as comments within the code itself.

```
-- VHDL Entity CAMmodel.snort_CAM.symbol
-- Created by Jacob J. Repanshek
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY snort_CAM IS
  PORT(
    Gmask_cam      : OUT   std_logic_vector (5 DOWNTO 0);
    Crb_init       : IN    std_logic_vector (2 DOWNTO 0);
    Burst_b_cam    : OUT   std_logic;
    Burst_b_init   : IN    std_logic;
    CeOe_b         : IN    std_logic;
    Crb_cam        : OUT   std_logic_vector (2 DOWNTO 0);
    Dev_cam        : OUT   std_logic_vector (2 DOWNTO 0);
    Dev_init       : IN    std_logic_vector (2 DOWNTO 0);
    Gmask_init     : IN    std_logic_vector (5 DOWNTO 0);
    HitAck         : IN    std_logic;
    InitDone       : IN    std_logic;
    Inst_cam       : OUT   std_logic_vector (10 DOWNTO 0);
    Inst_init      : IN    std_logic_vector (10 DOWNTO 0);
    LC_cam         : OUT   std_logic;
    LC_init        : IN    std_logic;
    LS_cam         : OUT   std_logic;
    LS_init        : IN    std_logic;
    MMOut_b        : IN    std_logic;
    MatchIn_cam    : OUT   std_logic_vector (6 DOWNTO 0);
```

```
      MatchIn_init    : IN     std_logic_vector (6 DOWNTO 0);
      MatchOut        : IN     std_logic;
      ParErr          : IN     std_logic;
      Phasen_cam      : OUT    std_logic;
      Phasen_init     : IN     std_logic;
      RdAck           : IN     std_logic;
      ReqData         : INOUT  std_logic_vector (71 DOWNTO 0);
      ReqStb_cam      : OUT    std_logic;
      ReqStb_init     : IN     std_logic;
      Rst_b_cam       : OUT    std_logic;
      Rst_b_init      : IN     std_logic;
      Tck_cam         : OUT    std_logic;
      Tck_init        : IN     std_logic;
      Tdi_cam         : OUT    std_logic;
      Tdi_init        : IN     std_logic;
      Tdo             : IN     std_logic;
      Tms_cam         : OUT    std_logic;
      Tms_init        : IN     std_logic;
      Trst_b_cam      : OUT    std_logic;
      Trst_b_init     : IN     std_logic;
      We_b            : IN     std_logic;
      clock_cam       : OUT    std_logic;
      clock_init      : IN     std_logic;
      indx            : IN     std_logic_vector (23 DOWNTO 0);
      valid           : IN     std_logic;
      Phasen          : IN     std_logic;
      clock           : IN     std_logic;
      field_data      : IN     std_logic_vector (31 DOWNTO 0);
      field_type      : IN     std_logic_vector (7 DOWNTO 0);
      reset           : IN     std_logic;
      valid_in        : IN     std_logic;
      ReqDataOut      : OUT    std_logic_vector (71 DOWNTO 0);
      address_valid   : OUT    std_logic;
      hit_address     : OUT    std_logic_vector (16 DOWNTO 0);
      search_type_out : OUT    std_logic_vector (1 DOWNTO 0)
   );

END snort_CAM ;


-- VHDL Architecture CAMmodel.snort_CAM.struct
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.STD_LOGIC_UNSIGNED.all;

LIBRARY CAMmodel;

ARCHITECTURE struct OF snort_CAM IS

   -- Architecture declarations
-- Non hierarchical state machine declarations
TYPE CSM1_CURRENT_STATE_STATE_TYPE IS (
      s0,
      extra_start,
      s48,
      s49,
      s50,
      header_start1,
      h2,
      s6,
      s7,
      s8,
      s9,
      s10,
      s11,
      s12,
      s3,
      s4,
      s43,
      s44,
      s45,
      s46,
      s47,
      s51,
      s29,
      s27,
      s26,
      s25,
      s24,
      s23,
      s22,
      s21,
      s20,
      s19,
      s18,
      s17,
      s16,
      s15,
      s14,
      payload_start1,
      s42,
      s41,
      s40,
      s39,
      s38,
      s30,
      s37,
      s36,
      s35,
      s34,
      s33,
      s32,
      s31,
      scroll_start
   );

-- State vector declaration
ATTRIBUTE state_vector : string;
ATTRIBUTE state_vector OF struct : ARCHITECTURE IS "csm1_current_state" ;


-- Declare current and next state signals
SIGNAL csm1_current_state : CSM1_CURRENT_STATE_STATE_TYPE ;
SIGNAL csm1_next_state : CSM1_CURRENT_STATE_STATE_TYPE ;


   -- Internal signal declarations
   -- Internal signal declarations
   SIGNAL Burst_b                    : std_logic;
   SIGNAL Crb                        : std_logic_vector(2 DOWNTO 0);
   SIGNAL Dev                        : std_logic_vector(2 DOWNTO 0);
   SIGNAL Gmask                      : std_logic_vector(5 DOWNTO 0);
   SIGNAL Inst                       : std_logic_vector(10 DOWNTO 0);
   SIGNAL LC                         : std_logic;
   SIGNAL LS                         : std_logic;
   SIGNAL MatchIn                    : std_logic_vector(6 DOWNTO 0);
   SIGNAL ReqStb                     : std_logic;
   SIGNAL Rst_b                      : std_logic;
   SIGNAL Tck                        : std_logic;
   SIGNAL Tdi                        : std_logic;
   SIGNAL Tms                        : std_logic;
   SIGNAL Trst_b                     : std_logic;
   SIGNAL arp_hardware               : std_logic_vector(15 DOWNTO 0);
   SIGNAL arp_hw_dest                : std_logic_vector(47 DOWNTO 0);
   SIGNAL arp_hw_len                 : std_logic_vector(7 DOWNTO 0);
   SIGNAL arp_hw_src                  : std_logic_vector(47 DOWNTO 0);
   SIGNAL arp_opcode                  : std_logic_vector(15 DOWNTO 0);
   SIGNAL arp_prot_len                : std_logic_vector(7 DOWNTO 0);
   SIGNAL arp_protocol                : std_logic_vector(15 DOWNTO 0);
   SIGNAL count_up                    : std_logic_vector(10 DOWNTO 0);
   SIGNAL ctrl_bus                    : std_logic_vector(40 DOWNTO 0);
   SIGNAL curr_shift                  : std_logic_vector(7 DOWNTO 0);
   SIGNAL dest_port                   : std_logic_vector(15 DOWNTO 0);
   SIGNAL dest_port_between_1000_1300   : std_logic;
   SIGNAL dest_port_between_3127_3199   : std_logic;
   SIGNAL dest_port_between_32771_34000 : std_logic;
   SIGNAL dest_port_between_32772_34000 : std_logic;
   SIGNAL dest_port_between_6666_7000   : std_logic;
   SIGNAL dest_port_gt_1023            : std_logic;
   SIGNAL dest_port_gt_499             : std_logic;
   SIGNAL dest_port_not_80             : std_logic;
   SIGNAL dsize_gt_1                  : std_logic;
   SIGNAL dsize_gt_100                : std_logic;
   SIGNAL dsize_gt_1000               : std_logic;
   SIGNAL dsize_gt_1023               : std_logic;
   SIGNAL dsize_gt_128                : std_logic;
   SIGNAL dsize_gt_1445               : std_logic;
   SIGNAL dsize_gt_156                : std_logic;
   SIGNAL dsize_gt_200                : std_logic;
   SIGNAL dsize_gt_500                : std_logic;
   SIGNAL dsize_gt_512                : std_logic;
   SIGNAL dsize_gt_6                  : std_logic;
   SIGNAL dsize_gt_720                : std_logic;
   SIGNAL dsize_gt_800                : std_logic;
   SIGNAL dsize_lt_25                 : std_logic;
   SIGNAL ethernet_checksum            : std_logic_vector(31 DOWNTO 0);
   SIGNAL ethernet_dest               : std_logic_vector(47 DOWNTO 0);
   SIGNAL ethernet_src                : std_logic_vector(47 DOWNTO 0);
   SIGNAL ethernet_type               : std_logic_vector(15 DOWNTO 0);
   SIGNAL header_ready                : std_logic;
   SIGNAL header_search_done           : std_logic;
   SIGNAL header_search_string         : std_logic_vector(575 DOWNTO 0);
   SIGNAL icmp_code                   : std_logic_vector(7 DOWNTO 0);
   SIGNAL icmp_code_gt_0              : std_logic;
   SIGNAL icmp_code_gt_1              : std_logic;
   SIGNAL icmp_code_gt_15             : std_logic;
   SIGNAL icmp_code_gt_2              : std_logic;
   SIGNAL icmp_code_gt_3              : std_logic;
   SIGNAL icmp_id                     : std_logic_vector(15 DOWNTO 0);
   SIGNAL icmp_seq                    : std_logic_vector(15 DOWNTO 0);
   SIGNAL icmp_type                   : std_logic_vector(7 DOWNTO 0);
   SIGNAL init_bus                    : std_logic_vector(40 DOWNTO 0);
   SIGNAL ip_dest                     : std_logic_vector(31 DOWNTO 0);
   SIGNAL ip_flags                    : std_logic_vector(2 DOWNTO 0);
   SIGNAL ip_header_checksum          : std_logic_vector(15 DOWNTO 0);
   SIGNAL ip_header_len               : std_logic_vector(3 DOWNTO 0);
   SIGNAL ip_identification           : std_logic_vector(15 DOWNTO 0);
   SIGNAL ip_offset                   : std_logic_vector(12 DOWNTO 0);
   SIGNAL ip_options                  : std_logic_vector(7 DOWNTO 0);
   SIGNAL ip_proto_gt_134             : std_logic;
   SIGNAL ip_protocol                 : std_logic_vector(7 DOWNTO 0);
   SIGNAL ip_src                      : std_logic_vector(31 DOWNTO 0);
   SIGNAL ip_total_len                : std_logic_vector(15 DOWNTO 0);
   SIGNAL ip_ttl                      : std_logic_vector(7 DOWNTO 0);
   SIGNAL ip_ttl_gt_220               : std_logic;
   SIGNAL ip_type                     : std_logic_vector(7 DOWNTO 0);
   SIGNAL ip_version                  : std_logic_vector(3 DOWNTO 0);
   SIGNAL max_shift                   : std_logic;
   SIGNAL padding                     : std_logic_vector(217 DOWNTO 0);
   SIGNAL payload_count               : std_logic_vector(15 DOWNTO 0);
   SIGNAL payload_len                 : std_logic_vector(15 DOWNTO 0);
   SIGNAL payload_ready               : std_logic;
   SIGNAL payload_search_done         : std_logic;
   SIGNAL payload_search_string       : std_logic_vector(575 DOWNTO 0);
   SIGNAL same_ip                     : std_logic;
   SIGNAL search_again                : std_logic;
   SIGNAL search_type                 : std_logic_vector(1 DOWNTO 0);
   SIGNAL shift_count                 : std_logic_vector(13 DOWNTO 0);
   SIGNAL shift_payload               : std_logic;
   SIGNAL src_port                    : std_logic_vector(15 DOWNTO 0);
   SIGNAL src_port_between_1000_1300  : std_logic;
   SIGNAL src_port_between_6666_7000  : std_logic;
   SIGNAL src_port_gt_1023            : std_logic;
   SIGNAL src_port_not_21_to_23       : std_logic;
   SIGNAL src_port_not_80             : std_logic;
   SIGNAL tcp_ack_num                 : std_logic_vector(31 DOWNTO 0);
   SIGNAL tcp_checksum                : std_logic_vector(15 DOWNTO 0);
   SIGNAL tcp_flags                   : std_logic_vector(7 DOWNTO 0);
   SIGNAL tcp_header_len              : std_logic_vector(3 DOWNTO 0);
   SIGNAL tcp_seq_num                 : std_logic_vector(31 DOWNTO 0);
   SIGNAL tcp_urgent_pointer          : std_logic_vector(15 DOWNTO 0);
   SIGNAL tcp_window_size             : std_logic_vector(15 DOWNTO 0);
   SIGNAL udp_checksum                : std_logic_vector(15 DOWNTO 0);
   SIGNAL udp_len                     : std_logic_vector(15 DOWNTO 0);
   SIGNAL wena                        : std_logic;
   SIGNAL ffout                       : std_logic_vector(7 DOWNTO 0);
   SIGNAL empty                       : std_logic;
```

```
   SIGNAL rena                        : std_logic;
   SIGNAL full                        : std_logic;
   SIGNAL aempty                         : std_logic;


   -- ModuleWare signal declarations(v1.0) for instance 'I1' of 'fifo'
   TYPE mw_I1sreg IS ARRAY (1500 DOWNTO 0) OF std_logic_vector(7 DOWNTO 0);
   SIGNAL mw_I1caddr : INTEGER RANGE 0 TO 1500;
   SIGNAL mw_I1naddr : INTEGER RANGE 0 TO 1500;
   SIGNAL mw_I1creg : mw_I1sreg;
   SIGNAL mw_I1nreg : mw_I1sreg;

signal temp_count : std_logic_vector(13 downto 0);


BEGIN
   -- Architecture concurrent statements
   -- HDL Embedded Block 1 controller
   -- Non hierarchical state machine

   -- This state machine flows from one search type to the other as packets are buffered. A packet
   -- header search begins when the first header information of a packet is completely registered.
   -- If the header search returns results, the flow shifts to the scroll super-state, in which all of the
   -- search matches are output. When all matches have been specified and enough of a payload
   -- has been buffered to start payload searching, the flow moves to the payload search super-
   -- state. When all the results for one payload search have been output, the flow either goes to
   -- another payload search if there is payload left to be searched, or back to header searching if a
   -- new packet header has been buffered.

   -----------------------------------------------------------------------------
   csm1_current_state_clocked : PROCESS(
      clock,
      reset
   )
   -----------------------------------------------------------------------------
   BEGIN
      IF (reset = '1') THEN
         csm1_current_state <= s0;
         -- Reset Values
      ELSIF (clock'EVENT AND clock = '1') THEN
         csm1_current_state <= csm1_next_state;
         -- Default Assignment To Internals

      END IF;

   END PROCESS csm1_current_state_clocked;
```

```
---------------------------------------------------------------------------
csm1_current_state_nextstate : PROCESS (
  HitAck,
  MatchOut,
  csm1_current_state,
  header_ready,
  payload_ready,
  search_again,
  search_type
)
---------------------------------------------------------------------------
BEGIN
  CASE csm1_current_state IS
  WHEN s0 =>
      csm1_next_state <= header_start1;
  WHEN extra_start =>
      csm1_next_state <= s48;
  WHEN s48 =>
      csm1_next_state <= s49;
  WHEN s49 =>
      csm1_next_state <= s50;
  WHEN s50 =>
    IF (((search_type="00") AND (payload_ready='1'))) THEN
      csm1_next_state <= payload_start1;
    ELSIF (search_again='1' AND payload_ready='1') THEN
      csm1_next_state <= payload_start1;
    ELSIF ((search_type="01")) THEN
      csm1_next_state <= header_start1;
    ELSE
      csm1_next_state <= s50;
    END IF;
  WHEN header_start1 =>
    IF ((header_ready='1')) THEN
      csm1_next_state <= h2;
    ELSE
      csm1_next_state <= header_start1;
    END IF;
  WHEN h2 =>
      csm1_next_state <= s6;
  WHEN s6 =>
      csm1_next_state <= s7;
  WHEN s7 =>
      csm1_next_state <= s8;
  WHEN s8 =>
      csm1_next_state <= s9;
```

```
WHEN s9 =>
    csm1_next_state <= s10;
WHEN s10 =>
    csm1_next_state <= s11;
WHEN s11 =>
    csm1_next_state <= s12;
WHEN s12 =>
    csm1_next_state <= s3;
WHEN s3 =>
    csm1_next_state <= s4;
WHEN s4 =>
    csm1_next_state <= s43;
WHEN s43 =>
    csm1_next_state <= s44;
WHEN s44 =>
    csm1_next_state <= s45;
WHEN s45 =>
    csm1_next_state <= s46;
WHEN s46 =>
    csm1_next_state <= s47;
WHEN s47 =>
    csm1_next_state <= s51;
WHEN s51 =>
  IF (HitAck='1') THEN
    csm1_next_state <= scroll_start;
  ELSE
    csm1_next_state <= extra_start;
  END IF;
WHEN s29 =>
  IF (HitAck='1') THEN
    csm1_next_state <= scroll_start;
  ELSIF (search_again='1') THEN
    csm1_next_state <= extra_start;
  ELSE
    csm1_next_state <= extra_start;
  END IF;
WHEN s27 =>
    csm1_next_state <= s29;
WHEN s26 =>
    csm1_next_state <= s27;
WHEN s25 =>
    csm1_next_state <= s26;
WHEN s24 =>
    csm1_next_state <= s25;
WHEN s23 =>
    csm1_next_state <= s24;
```

```
WHEN s22 =>
    csm1_next_state <= s23;
WHEN s21 =>
    csm1_next_state <= s22;
WHEN s20 =>
    csm1_next_state <= s21;
WHEN s19 =>
    csm1_next_state <= s20;
WHEN s18 =>
    csm1_next_state <= s19;
WHEN s17 =>
    csm1_next_state <= s18;
WHEN s16 =>
    csm1_next_state <= s17;
WHEN s15 =>
    csm1_next_state <= s16;
WHEN s14 =>
    csm1_next_state <= s15;
WHEN payload_start1 =>
    csm1_next_state <= s14;
WHEN s42 =>
  IF (((search_type="01" AND search_again='1') AND (payload_ready='1'))) THEN
    csm1_next_state <= payload_start1;
  ELSIF ((search_type="01")) THEN
    csm1_next_state <= header_start1;
  ELSIF (((search_type="00") AND (payload_ready='1'))) THEN
    csm1_next_state <= payload_start1;
  ELSE
    csm1_next_state <= s42;
  END IF;
WHEN s41 =>
    csm1_next_state <= s42;
WHEN s40 =>
    csm1_next_state <= s41;
WHEN s39 =>
    csm1_next_state <= s40;
WHEN s38 =>
    csm1_next_state <= s39;
WHEN s30 =>
    csm1_next_state <= s38;
WHEN s37 =>
  IF ((MatchOut='1')) THEN
    csm1_next_state <= scroll_start;
  ELSE
    csm1_next_state <= s30;
  END IF;
```

```
   WHEN s36 =>
         csm1_next_state <= s37;
   WHEN s35 =>
         csm1_next_state <= s36;
   WHEN s34 =>
         csm1_next_state <= s35;
   WHEN s33 =>
         csm1_next_state <= s34;
   WHEN s32 =>
         csm1_next_state <= s33;
   WHEN s31 =>
         csm1_next_state <= s32;
   WHEN scroll_start =>
         csm1_next_state <= s31;
   WHEN OTHERS =>
      csm1_next_state <= s0;
   END CASE;

END PROCESS csm1_current_state_nextstate;

-----------------------------------------------------------------------------
csm1_current_state_output : PROCESS (
   HitAck,
   MatchOut,
   csm1_current_state,
   header_ready,
   header_search_string,
   max_shift,
   payload_ready,
   payload_search_string,
   search_again,
   search_type,
   shift_count,
   temp_count
)
-----------------------------------------------------------------------------
BEGIN
   -- Default Assignment
   Burst_b <= '1';
   Crb <= "000";
   Dev <= "000";
   Gmask <= "000000";
   Inst <= "00000000000";
   LC <= '1';
   LS <= '1';
   MatchIn <= "0000000";
```

```
    ReqData <=
"ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
ZZZZZZZZZ";
    ReqStb <= '0';
    Rst_b <= '1';
    Tck <= '0';
    Tdi <= '0';
    Tms <= '0';
    Trst_b <= '0';
    header_search_done <= '0';
    payload_search_done <= '0';
    shift_payload <= '0';
    -- Default Assignment To Internals

    -- Combined Actions
    CASE csm1_current_state IS
    WHEN s0 =>
      shift_count<="00000000000001";
      search_type<="00";
    WHEN s50 =>
      IF (((search_type="00") AND (payload_ready='1'))) THEN
        shift_count<="00000000000001";
        header_search_done<='1';
        -- perform 576-bit lookup on a snort payload rule
        ReqStb<='1';
        Inst<="00000111000";
        ReqData<=payload_search_string(575 downto 504);
      ELSIF (search_again='1' AND payload_ready='1') THEN
        -- perform 576-bit lookup on a snort payload rule
        ReqStb<='1';
        Inst<="00000111000";
        ReqData<=payload_search_string(575 downto 504);
      END IF;
    WHEN header_start1 =>
      IF ((header_ready='1')) THEN
        -- perform 576-bit lookup on header information
                ReqStb<='1';
        Inst<="00000111000";
        ReqData<=header_search_string(575 downto 504);
        search_type<="00";
      END IF;
    WHEN h2 =>
      Inst<="00000111000";
      ReqData<=header_search_string(503 downto 432);
    WHEN s6 =>
      Inst<="00000111000";
```

```
  ReqData<=header_search_string(431 downto 360);
WHEN s7 =>
  Inst<="00000111000";
  ReqData<=header_search_string(359 downto 288);
WHEN s8 =>
  Inst<="00000111000";
  ReqData<=header_search_string(287 downto 216);
WHEN s9 =>
  Inst<="00000111000";
  ReqData<=header_search_string(215 downto 144);
WHEN s10 =>
  Inst<="00000111000";
  ReqData<=header_search_string(143 downto 72);
WHEN s11 =>
  Inst<="00000111000";
  ReqData<=header_search_string(71 downto 0);
WHEN s51 =>
  IF (HitAck='1') THEN
    ReqStb<='1';
    Inst<="00000111001";
  END IF;
WHEN s29 =>
  IF (HitAck='1') THEN
    ReqStb<='1';
    Inst<="00000111001";
  ELSIF (search_again='1') THEN
  ELSE
    payload_search_done<='1';
  END IF;
WHEN s27 =>
  shift_count<=temp_count(12 downto 0)&'0';
WHEN s26 =>
  temp_count<=shift_count;
WHEN s20 =>
  shift_payload<='1';
WHEN s19 =>
  Inst<="00000111000";
  ReqData<=payload_search_string(71 downto 16)&max_shift&shift_count&'1';
WHEN s18 =>
  Inst<="00000111000";
  ReqData<=payload_search_string(143 downto 72);
WHEN s17 =>
  Inst<="00000111000";
  ReqData<=payload_search_string(215 downto 144);
WHEN s16 =>
  Inst<="00000111000";
```

```
      ReqData<=payload_search_string(287 downto 216);
    WHEN s15 =>
      Inst<="00000111000";
      ReqData<=payload_search_string(359 downto 288);
    WHEN s14 =>
      Inst<="00000111000";
      ReqData<=payload_search_string(431 downto 360);
    WHEN payload_start1 =>
      Inst<="00000111000";
      ReqData<=payload_search_string(503 downto 432);
      search_type<="01";
    WHEN s42 =>
      IF (((search_type="01" AND search_again='1') AND (payload_ready='1'))) THEN
        -- perform 576-bit lookup on a snort payload rule
        ReqStb<='1';
        Inst<="00000111000";
        ReqData<=payload_search_string(575 downto 504);
      ELSIF ((search_type="01")) THEN
        payload_search_done<='1';
      ELSIF (((search_type="00") AND (payload_ready='1'))) THEN
        header_search_done<='1';
        shift_count<="00000000000001";
        -- perform 576-bit lookup on a snort payload rule
        ReqStb<='1';
        Inst<="00000111000";
        ReqData<=payload_search_string(575 downto 504);
      END IF;
    WHEN s37 =>
      IF ((MatchOut='1')) THEN
        ReqStb<='1';
        Inst<="00000111001";
      END IF;
    WHEN scroll_start =>
      Inst<="00000111001";
    WHEN OTHERS =>
      NULL;
    END CASE;

  END PROCESS csm1_current_state_output;

  -- Concurrent Statements

  ReqDataOut<=ReqData;

  -- HDL Embedded Text Block 4 CAM_mux
```

```
-- This process serves as a mux that controls what signals reach the CAM:
-- signals from the initialization unit, or signals from within the snort_CAM entity.

  process(InitDone,ctrl_bus,init_bus)
  begin

    case InitDone is
      when '1' =>
        clock_cam <= ctrl_bus(0);
        Phasen_cam <= ctrl_bus(1);
        Dev_cam <= ctrl_bus(4 downto 2);
        Rst_b_cam <= ctrl_bus(5);
        Burst_b_cam <= ctrl_bus(6);
        ReqStb_cam <= ctrl_bus(7);
        Inst_cam <= ctrl_bus(18 downto 8);
        MatchIn_cam <= ctrl_bus(25 downto 19);
        Gmask_cam <= ctrl_bus(31 downto 26);
        Crb_cam <= ctrl_bus(34 downto 32);
        Tms_cam <= ctrl_bus(35);
        Tdi_cam <= ctrl_bus(36);
        Tck_cam <= ctrl_bus(37);
        Trst_b_cam <= ctrl_bus(38);
        LC_cam <= ctrl_bus(39);
        LS_cam <= ctrl_bus(40);
      when others =>
        clock_cam <= init_bus(0);
        Phasen_cam <= init_bus(1);
        Dev_cam <= init_bus(4 downto 2);
        Rst_b_cam <= init_bus(5);
        Burst_b_cam <= init_bus(6);
        ReqStb_cam <= init_bus(7);
        Inst_cam <= init_bus(18 downto 8);
        MatchIn_cam <= init_bus(25 downto 19);
        Gmask_cam <= init_bus(31 downto 26);
        Crb_cam <= init_bus(34 downto 32);
        Tms_cam <= init_bus(35);
        Tdi_cam <= init_bus(36);
        Tck_cam <= init_bus(37);
        Trst_b_cam <= init_bus(38);
        LC_cam <= init_bus(39);
        LS_cam <= init_bus(40);
    end case;

  end process;
```

```
-- HDL Embedded Text Block 4 packet_field_reg

-- This process registes packet header information as it is read in on the field_data bus.

process(clock,reset)
  begin
    if(reset='1') then
      ethernet_dest <= "000000000000000000000000000000000000000000000000";
      ethernet_src <= "000000000000000000000000000000000000000000000000";
      ethernet_type <= "0000000000000000";
      arp_hardware <= "0000000000000000";
      arp_protocol <= "0000000000000000";
      arp_hw_len <= "00000000";
      arp_prot_len <= "00000000";
      arp_opcode <= "0000000000000000";
      arp_hw_src <= "000000000000000000000000000000000000000000000000";
      arp_hw_dest <= "000000000000000000000000000000000000000000000000";
      ip_src <= "00000000000000000000000000000000";
      ip_dest <= "00000000000000000000000000000000";
      ethernet_checksum <= "00000000000000000000000000000000";
      ip_version <= "0000";
      ip_header_len <= "0000";
      ip_type <= "00000000";
      ip_total_len <= "0000000000000000";
      ip_identification <= "0000000000000000";
      ip_flags <= "000";
      ip_offset <= "0000000000000";
      ip_ttl <= "00000000";
      ip_protocol <= "00000000";
      ip_header_checksum <= "0000000000000000";
      src_port <= "0000000000000000";
      dest_port <= "0000000000000000";
      tcp_seq_num <= "00000000000000000000000000000000";
      tcp_ack_num <= "00000000000000000000000000000000";
      tcp_header_len <= "0000";
      tcp_flags <= "00000000";
      tcp_window_size <= "0000000000000000";
      tcp_checksum <= "0000000000000000";
      tcp_urgent_pointer <= "0000000000000000";
      udp_len <= "0000000000000000";
      udp_checksum <= "0000000000000000";
      ip_options <= "00000000";
      icmp_type <= "00000000";
      icmp_code <= "00000000";
      icmp_id <= "0000000000000000";
      icmp_seq <= "0000000000000000";
```

```
elsif(rising_edge(clock)) then
  if(header_search_done='1') then
    ethernet_dest <= "000000000000000000000000000000000000000000000000";
    ethernet_src <= "000000000000000000000000000000000000000000000000";
    ethernet_type <= "0000000000000000";
    arp_hardware <= "0000000000000000";
    arp_protocol <= "0000000000000000";
    arp_hw_len <= "00000000";
    arp_prot_len <= "00000000";
    arp_opcode <= "0000000000000000";
    arp_hw_src <= "000000000000000000000000000000000000000000000000";
    arp_hw_dest <= "000000000000000000000000000000000000000000000000";
    ip_src <= "00000000000000000000000000000000";
    ip_dest <= "00000000000000000000000000000000";
    ethernet_checksum <= "00000000000000000000000000000000";
    ip_version <= "0000";
    ip_header_len <= "0000";
    ip_type <= "00000000";
    ip_total_len <= "0000000000000000";
    ip_identification <= "0000000000000000";
    ip_flags <= "000";
    ip_offset <= "0000000000000";
    ip_ttl <= "00000000";
    ip_protocol <= "00000000";
    ip_header_checksum <= "0000000000000000";
    src_port <= "0000000000000000";
    dest_port <= "0000000000000000";
    tcp_seq_num <= "00000000000000000000000000000000";
    tcp_ack_num <= "00000000000000000000000000000000";
    tcp_header_len <= "0000";
    tcp_flags <= "00000000";
    tcp_window_size <= "0000000000000000";
    tcp_checksum <= "0000000000000000";
    tcp_urgent_pointer <= "0000000000000000";
    udp_len <= "0000000000000000";
    udp_checksum <= "0000000000000000";
    ip_options <= "00000000";
    icmp_type <= "00000000";
    icmp_code <= "00000000";
    icmp_id <= "0000000000000000";
    icmp_seq <= "0000000000000000";
  elsif(valid_in='1') then
    case field_type is
      when "00000010" =>
        ethernet_dest(47 downto 16) <= field_data;
      when "00000011" =>
  ethernet_dest(15 downto 0) <= field_data(15 downto 0);
when "00000100" =>
  ethernet_src(47 downto 16) <= field_data;
when "00000101" =>
  ethernet_src(15 downto 0) <= field_data(15 downto 0);
when "00000110" =>
  ethernet_type <= field_data(15 downto 0);
when "00000111" =>
  arp_hardware <= field_data(15 downto 0);
when "00001000" =>
  arp_protocol <= field_data(15 downto 0);
when "00001001" =>
  arp_hw_len <= field_data(7 downto 0);
when "00001010" =>
  arp_prot_len <= field_data(7 downto 0);
when "00001011" =>
  arp_opcode <= field_data(15 downto 0);
when "00001100" =>
  arp_hw_src(47 downto 16) <= field_data;
when "00001101" =>
  arp_hw_src(15 downto 0) <= field_data(15 downto 0);
when "00001110" =>
  ip_src <= field_data;
when "00001111" =>
  arp_hw_dest(47 downto 16) <= field_data;
when "00010000" =>
  arp_hw_dest(15 downto 0) <= field_data(15 downto 0);
when "00010001" =>
  ip_dest <= field_data;
when "00010110" =>
  ethernet_checksum <= field_data;
when "00011000" =>
  ip_version <= field_data(7 downto 4);
  ip_header_len <= field_data(3 downto 0);
when "00011001" =>
  ip_type <= field_data(7 downto 0);
when "00011010" =>
  ip_total_len <= field_data(15 downto 0);
when "00011011" =>
  ip_identification <= field_data(15 downto 0);
when "00011100" =>
  ip_flags <= field_data(15 downto 13);
  ip_offset <= field_data(12 downto 0);
when "00011101" =>
  ip_ttl <= field_data(7 downto 0);
when "00011110" =>
```

```
            ip_protocol <= field_data(7 downto 0);
          when "00011111" =>
            ip_header_checksum <= field_data(15 downto 0);
          when "00100000" =>
            ip_src <= field_data;
          when "00100001" =>
            ip_dest <= field_data;
          when "00100010" =>
            if(ip_options = "00000000") then
              ip_options <= field_data(31 downto 24);
            end if;
          when "00100011" =>
            src_port <= field_data(15 downto 0);
          when "00100100" =>
            dest_port <= field_data(15 downto 0);
          when "00100101" =>
            tcp_seq_num <= field_data;
          when "00100110" =>
            tcp_ack_num <= field_data;
          when "00100111" =>
            tcp_header_len <= field_data(15 downto 12);
            tcp_flags <= field_data(7 downto 0);
          when "00101000" =>
            tcp_window_size <= field_data(15 downto 0);
          when "00101001" =>
            tcp_checksum <= field_data(15 downto 0);
          when "00101010" =>
            tcp_urgent_pointer <= field_data(15 downto 0);
          when "00101101" =>
            src_port <= field_data(15 downto 0);
          when "00101110" =>
            dest_port <= field_data(15 downto 0);
          when "00101111" =>
            udp_len <= field_data(15 downto 0);
          when "00110000" =>
            udp_checksum <= field_data(15 downto 0);
          when others =>
        end case;
      end if;
    end if;
  end process;

-- HDL Embedded Text Block 5 payload_len_reg

-- The process calculates the length of the payload by subtracting the IP header length
-- and TCP or UDP header length from the total IP packet length.

process(clock,reset)
  variable iplen : std_logic_vector(5 downto 0);
  variable tcplen : std_logic_vector(5 downto 0);
begin
  if(reset='1') then
    payload_len <= "0000000000000000";
  elsif(rising_edge(clock)) then
    if(ip_protocol="00000110") then
      iplen := ip_header_len&"00";
      tcplen := tcp_header_len&"00";
      payload_len <= unsigned(ip_total_len) - (unsigned(iplen) + unsigned(tcplen));
    end if;
    if(ip_protocol="00010001") then
      payload_len <= unsigned(udp_len) - 8;
    end if;
  end if;
end process;

-- HDL Embedded Text Block 6 search_status

-- This process keeps track of the current search status, which is to say what kind of search is
-- occurring or about to occur. Once payload information begins to arrive from ePAPP, one
-- knows that the header must be done registering, and header_ready is asserted. When 70 bytes
-- of payload have been buffered (or, in the case of shorter payloads, the payload has been
-- buffered and padded with enough zeros to fill 70 bytes), payload_ready is asserted. These
-- signals are turned off when a header search or payload search completes, respectively.

  process(clock,reset)
    variable curr_payload : std_logic;
  begin
    if(reset='1') then
      header_ready <= '0';
      payload_ready <= '0';
      curr_payload := '0';
    elsif(rising_edge(clock)) then

      -- if currently receiving payload information
      if(field_type="00110001" OR field_type="00101100") then

        -- set header_ready flag
        if(curr_payload='0') then
          header_ready<='1';
        else
          header_ready<='0';
        end if;
      end if;

      -- signal when 72 bytes of payload have been buffered or payload information is done
      if(count_up="00001000111") then
        payload_ready<='1';
      end if;

      -- header search has completed
      if(header_search_done='1') then
        header_ready<='0';
        curr_payload := '1';
      end if;

      -- payload search has completed
      if(payload_search_done='1') then
        payload_ready<='0';
        curr_payload := '0';
      end if;

    end if;
  end process;

-- HDL Embedded Text Block 8 logic_block

-- This block uses data registered in the packet_field_reg block to calculate
-- all the special cases needed for packet header searching.

  process(clock,reset)
  begin
    if(reset='1') then
      src_port_not_21_to_23 <= '0';
      src_port_gt_1023 <= '0';
      dest_port_gt_1023 <= '0';
      src_port_between_1000_1300 <= '0';
      dest_port_between_1000_1300 <= '0';
      dest_port_between_3127_3199 <= '0';
      src_port_between_6666_7000 <= '0';
      dest_port_between_6666_7000 <= '0';
      dest_port_between_32772_34000 <= '0';
      dest_port_between_32771_34000 <= '0';
      src_port_not_80 <= '0';
      dest_port_not_80 <= '0';
      dest_port_gt_499 <= '0';
      ip_ttl_gt_220 <= '0';
      dsize_gt_1 <= '0';
```

```
            dsize_gt_6 <= '0';
            dsize_gt_1445 <= '0';
            dsize_gt_1023 <= '0';
            dsize_gt_1000 <= '0';
            dsize_gt_512 <= '0';
            dsize_gt_128 <= '0';
            dsize_gt_720 <= '0';
            dsize_gt_100 <= '0';
            dsize_gt_800 <= '0';
            dsize_lt_25 <= '0';
            dsize_gt_500 <= '0';
            dsize_gt_156 <= '0';
            dsize_gt_200 <= '0';
            ip_proto_gt_134 <= '0';
            same_ip <= '0';
            icmp_code_gt_0 <= '0';
            icmp_code_gt_15 <= '0';
            icmp_code_gt_2 <= '0';
            icmp_code_gt_3 <= '0';
            icmp_code_gt_1 <= '0';

          elsif(rising_edge(clock)) then
            -- trigger if the source port is not 21, 22, or 23
            if(not(src_port="0000000000010101" OR src_port="0000000000010110" OR
src_port="0000000000010111")) then
              src_port_not_21_to_23 <= '1';
            else
              src_port_not_21_to_23 <= '0';
            end if;

            -- trigger if the source port is greater than 1023
            if(unsigned(src_port) > 1023) then
              src_port_gt_1023 <= '1';
            else
              src_port_gt_1023 <= '0';
            end if;

            -- trigger if the destination port is greater than 1023
            if(unsigned(dest_port) > 1023) then
              dest_port_gt_1023 <= '1';
            else
              dest_port_gt_1023 <= '0';
            end if;

            -- trigger if the source port is between 1000 and 1300 (inclusive)
            if((unsigned(src_port) > 999) AND (unsigned(src_port) < 1301)) then
```

```
  src_port_between_1000_1300 <= '1';
else
  src_port_between_1000_1300 <= '0';
end if;

-- trigger if the destination port is between 1000 and 1300 (inclusive)
if((unsigned(dest_port) > 999) AND (unsigned(dest_port) < 1301)) then
  dest_port_between_1000_1300 <= '1';
else
  dest_port_between_1000_1300 <= '0';
end if;

-- trigger if the destination port is between 3127 and 3199 (inclusive)
if((unsigned(dest_port) > 3126) AND (unsigned(dest_port) < 3200)) then
  dest_port_between_3127_3199 <= '1';
else
  dest_port_between_3127_3199 <= '0';
end if;

-- trigger if the source port is between 6666 and 7000 (inclusive)
if((unsigned(src_port) > 6665) AND (unsigned(src_port) < 7001)) then
  src_port_between_6666_7000 <= '1';
else
  src_port_between_6666_7000 <= '0';
end if;

-- trigger if the destination port is between 6666 and 7000 (inclusive)
if((unsigned(dest_port) > 6665) AND (unsigned(dest_port) < 7001)) then
  dest_port_between_6666_7000 <= '1';
else
  dest_port_between_6666_7000 <= '0';
end if;

-- trigger if the destination port is between 32772 and 34000 (inclusive)
if((unsigned(dest_port) > 32771) AND (unsigned(dest_port) < 34001)) then
  dest_port_between_32772_34000 <= '1';
else
  dest_port_between_32772_34000 <= '0';
end if;

-- trigger if the destination port is between 32771 and 34000 (inclusive)
if((unsigned(dest_port) > 32770) AND (unsigned(dest_port) < 34001)) then
  dest_port_between_32771_34000 <= '1';
else
  dest_port_between_32771_34000 <= '0';
end if;
```

```
-- trigger if the source port is not port 80
if(src_port /= "0000000001010000") then
  src_port_not_80 <= '1';
else
  src_port_not_80 <= '0';
end if;

-- trigger if the destination port is not port 80
if(dest_port /= "0000000001010000") then
  dest_port_not_80 <= '1';
else
  dest_port_not_80 <= '0';
end if;

-- trigger if the destination port is greater than 499
if(unsigned(dest_port) > 499) then
  dest_port_gt_499 <= '1';
else
  dest_port_gt_499 <= '0';
end if;

-- trigger if the IP TTL is greater than 220
if(unsigned(ip_ttl) > 220) then
  ip_ttl_gt_220 <= '1';
else
  ip_ttl_gt_220 <= '0';
end if;

-- trigger if DSIZE is greater than 1
if(unsigned(payload_len) > 1) then
  dsize_gt_1 <= '1';
else
  dsize_gt_1 <= '0';
end if;

-- trigger if DSIZE is greater than 6
if(unsigned(payload_len) > 6) then
  dsize_gt_6 <= '1';
else
  dsize_gt_6 <= '0';
end if;

-- trigger if DSIZE is greater than 1445
if(unsigned(payload_len) > 1445) then
  dsize_gt_1445 <= '1';
```

```
else
  dsize_gt_1445 <= '0';
end if;

-- trigger if DSIZE is greater than 1023
if(unsigned(payload_len) > 1023) then
  dsize_gt_1023 <= '1';
else
  dsize_gt_1023 <= '0';
end if;

-- trigger if DSIZE is greater than 1000
if(unsigned(payload_len) > 1000) then
  dsize_gt_1000 <= '1';
else
  dsize_gt_1000 <= '0';
end if;

-- trigger if DSIZE is greater than 512
if(unsigned(payload_len) > 512) then
  dsize_gt_512 <= '1';
else
  dsize_gt_512 <= '0';
end if;

-- trigger if DSIZE is greater than 128
if(unsigned(payload_len) > 128) then
  dsize_gt_128 <= '1';
else
  dsize_gt_128 <= '0';
end if;

-- trigger if DSIZE is greater than 720
if(unsigned(payload_len) > 720) then
  dsize_gt_720 <= '1';
else
  dsize_gt_720 <= '0';
end if;

-- trigger if DSIZE is greater than 100
if(unsigned(payload_len) > 100) then
  dsize_gt_100 <= '1';
else
  dsize_gt_100 <= '0';
end if;
```

```
-- trigger if DSIZE is greater than 800
if(unsigned(payload_len) > 800) then
  dsize_gt_800 <= '1';
else
  dsize_gt_800 <= '0';
end if;

-- trigger if DSIZE is less than 25
if(unsigned(payload_len) < 25) then
  dsize_lt_25 <= '1';
else
  dsize_lt_25 <= '0';
end if;

-- trigger if DSIZE is greater than 500
if(unsigned(payload_len) > 500) then
  dsize_gt_500 <= '1';
else
  dsize_gt_500 <= '0';
end if;

-- trigger if DSIZE is greater than 156
if(unsigned(payload_len) > 156) then
  dsize_gt_156 <= '1';
else
  dsize_gt_156 <= '0';
end if;

-- trigger if DSIZE is greater than 200
if(unsigned(payload_len) > 200) then
  dsize_gt_200 <= '1';
else
  dsize_gt_200 <= '0';
end if;

-- trigger if IP protocol is greater than 134
if(unsigned(ip_protocol) > 134) then
  ip_proto_gt_134 <= '1';
else
  ip_proto_gt_134 <= '0';
end if;

-- trigger if source and dest IPs are the same
if(ip_src=ip_dest) then
  same_ip <= '1';
else
```

```
          same_ip <= '0';
        end if;

        -- trigger if icmp code is greater than 0
        if(unsigned(icmp_code) > 0) then
          icmp_code_gt_0 <= '1';
        else
          icmp_code_gt_0 <= '0';
        end if;

        -- trigger if icmp code is greater than 15
        if(unsigned(icmp_code) > 15) then
          icmp_code_gt_15 <= '1';
        else
          icmp_code_gt_15 <= '0';
        end if;

        -- trigger if icmp code is greater than 2
        if(unsigned(icmp_code) > 2) then
          icmp_code_gt_2 <= '1';
        else
          icmp_code_gt_2 <= '0';
        end if;

        -- trigger if icmp code is greater than 3
        if(unsigned(icmp_code) > 3) then
          icmp_code_gt_3 <= '1';
        else
          icmp_code_gt_3 <= '0';
        end if;

        -- trigger if icmp code is greater than 1
        if(unsigned(icmp_code) > 1) then
          icmp_code_gt_1 <= '1';
        else
          icmp_code_gt_1 <= '0';
        end if;

      end if;
    end process;

-- HDL Embedded Text Block 8 payload_counter

-- This block keeps track of how much payload has been registers and/or searched at a given
-- time. payload_count is initialized to the size of the payload and counts down to zero to
-- indicate that the entire payload has been buffered. count_up is initialized to zero and counts
-- up with every new payload byte registered. remaining is initialized to the size of the payload
-- is decremented by eight every time an eight-byte payload shift occurs.

    process(clock,reset)
      variable active : std_logic;
      variable remaining : std_logic_vector(15 downto 0);
      variable start_count : std_logic;

    begin
      if(reset='1') then
        payload_count<="0000000000000000";
        count_up<="00000000000";
        active := '0';
        remaining := "0000000000000000";
        search_again<='0';
        max_shift<='0';
        start_count := '0';

      elsif(rising_edge(clock)) then
        -- if first byte of payload information is arriving, start counting
        if(active='0' AND (field_type="00101100" OR field_type="00110001")) then
          payload_count<=payload_len-2;
          count_up<="00000000000";
          active:='1';
          remaining := payload_len+8;
          start_count:='1';
        -- continue to count
        else
          if(active='1') then
            if(payload_count="0000000000000000") then
              active:='0';
            else
              payload_count<=payload_count-1;
            end if;

            count_up<=count_up+1;

          end if;

          if(start_count='1') then
            count_up<=count_up+1;
          end if;

        end if;

        -- decrement payload length remaining by 8 bytes
        if(shift_payload='1') then
          remaining := (remaining - 8);
        end if;

        -- if some payload remains, search again
        if(remaining(15 downto 3)="0000000000000") then
          search_again<='0';
        else
          search_again<='1';
        end if;

        -- trigger if max number of offset or depth shifts has occured
        if(shift_count="00000000000000") then
          max_shift<='1';
        else
          max_shift<='0';
        end if;
      end if;
    end process;

-- HDL Embedded Text Block 9 payload_check

-- This process enables the payload FIFO to read incoming bytes of payload data.

process(reset,field_type)
begin
    if(reset='1') then
      wena<='0';
    elsif(field_type="00110001" OR field_type="00101100") then
      wena<='1';
    else
      wena<='0';
    end if;
end process;

-- HDL Embedded Text Block 10 result_reg

-- This process registers outputs from the CASMA unit.

    process(clock,reset)
    begin
      if(reset='1') then
        hit_address<="0000000000000000";
        search_type_out<="00";
        address_valid<='0';
      elsif(rising_edge(clock)) then
```

```
      address_valid<=HitAck;
      if(HitAck='1') then
        hit_address<=indx(16 downto 0);
        search_type_out<=search_type;
      end if;
    end if;
  end process;

-- HDL Embedded Text Block 12 payload_reg

-- The process contains 70 1-byte registers that each hold a byte of payload information. As a
-- payload is buffered, bytes of data are shifted into these registers until they are all full, at
-- which point payload searching begins. There is a valid bit associated with each of these
-- registers to specify whether the register holds valid data. When an eight-byte shift is to
-- occur, the last eight valid bits are cleared so that eight more bytes will be shifted out of the
-- FIFO into the registers.

process(clock,reset)
  variable cnt : std_logic_vector(1 downto 0);
  variable empty_late : std_logic;

  variable valid1 : std_logic;
  variable valid2 : std_logic;
  variable valid3 : std_logic;
  variable valid4 : std_logic;
  variable valid5 : std_logic;
  variable valid6 : std_logic;
  variable valid7 : std_logic;
  variable valid8 : std_logic;
  variable valid9 : std_logic;
  variable valid10 : std_logic;
  variable valid11 : std_logic;
  variable valid12 : std_logic;
  variable valid13 : std_logic;
  variable valid14 : std_logic;
  variable valid15 : std_logic;
  variable valid16 : std_logic;
  variable valid17 : std_logic;
  variable valid18 : std_logic;
  variable valid19 : std_logic;
  variable valid20 : std_logic;
  variable valid21 : std_logic;
  variable valid22 : std_logic;
  variable valid23 : std_logic;
  variable valid24 : std_logic;
  variable valid25 : std_logic;
  variable valid26 : std_logic;
  variable valid27 : std_logic;
  variable valid28 : std_logic;
  variable valid29 : std_logic;
  variable valid30 : std_logic;
  variable valid31 : std_logic;
  variable valid32 : std_logic;
  variable valid33 : std_logic;
  variable valid34 : std_logic;
  variable valid35 : std_logic;
  variable valid36 : std_logic;
  variable valid37 : std_logic;
  variable valid38 : std_logic;
  variable valid39 : std_logic;
  variable valid40 : std_logic;
  variable valid41 : std_logic;
  variable valid42 : std_logic;
  variable valid43 : std_logic;
  variable valid44 : std_logic;
  variable valid45 : std_logic;
  variable valid46 : std_logic;
  variable valid47 : std_logic;
  variable valid48 : std_logic;
  variable valid49 : std_logic;
  variable valid50 : std_logic;
  variable valid51 : std_logic;
  variable valid52 : std_logic;
  variable valid53 : std_logic;
  variable valid54 : std_logic;
  variable valid55 : std_logic;
  variable valid56 : std_logic;
  variable valid57 : std_logic;
  variable valid58 : std_logic;
  variable valid59 : std_logic;
  variable valid60 : std_logic;
  variable valid61 : std_logic;
  variable valid62 : std_logic;
  variable valid63 : std_logic;
  variable valid64 : std_logic;
  variable valid65 : std_logic;
  variable valid66 : std_logic;
  variable valid67 : std_logic;
  variable valid68 : std_logic;
  variable valid69 : std_logic;
  variable valid70 : std_logic;
  variable valid71 : std_logic;
  variable valid72 : std_logic;

  variable reg1 : std_logic_vector(7 downto 0);
  variable reg2 : std_logic_vector(7 downto 0);
  variable reg3 : std_logic_vector(7 downto 0);
  variable reg4 : std_logic_vector(7 downto 0);
  variable reg5 : std_logic_vector(7 downto 0);
  variable reg6 : std_logic_vector(7 downto 0);
  variable reg7 : std_logic_vector(7 downto 0);
  variable reg8 : std_logic_vector(7 downto 0);
  variable reg9 : std_logic_vector(7 downto 0);
  variable reg10 : std_logic_vector(7 downto 0);
  variable reg11 : std_logic_vector(7 downto 0);
  variable reg12 : std_logic_vector(7 downto 0);
  variable reg13 : std_logic_vector(7 downto 0);
  variable reg14 : std_logic_vector(7 downto 0);
  variable reg15 : std_logic_vector(7 downto 0);
  variable reg16 : std_logic_vector(7 downto 0);
  variable reg17 : std_logic_vector(7 downto 0);
  variable reg18 : std_logic_vector(7 downto 0);
  variable reg19 : std_logic_vector(7 downto 0);
  variable reg20 : std_logic_vector(7 downto 0);
  variable reg21 : std_logic_vector(7 downto 0);
  variable reg22 : std_logic_vector(7 downto 0);
  variable reg23 : std_logic_vector(7 downto 0);
  variable reg24 : std_logic_vector(7 downto 0);
  variable reg25 : std_logic_vector(7 downto 0);
  variable reg26 : std_logic_vector(7 downto 0);
  variable reg27 : std_logic_vector(7 downto 0);
  variable reg28 : std_logic_vector(7 downto 0);
  variable reg29 : std_logic_vector(7 downto 0);
  variable reg30 : std_logic_vector(7 downto 0);
  variable reg31 : std_logic_vector(7 downto 0);
  variable reg32 : std_logic_vector(7 downto 0);
  variable reg33 : std_logic_vector(7 downto 0);
  variable reg34 : std_logic_vector(7 downto 0);
  variable reg35 : std_logic_vector(7 downto 0);
  variable reg36 : std_logic_vector(7 downto 0);
  variable reg37 : std_logic_vector(7 downto 0);
  variable reg38 : std_logic_vector(7 downto 0);
  variable reg39 : std_logic_vector(7 downto 0);
  variable reg40 : std_logic_vector(7 downto 0);
  variable reg41 : std_logic_vector(7 downto 0);
  variable reg42 : std_logic_vector(7 downto 0);
  variable reg43 : std_logic_vector(7 downto 0);
  variable reg44 : std_logic_vector(7 downto 0);
```

```
  variable reg45 : std_logic_vector(7 downto 0);
  variable reg46 : std_logic_vector(7 downto 0);
  variable reg47 : std_logic_vector(7 downto 0);
  variable reg48 : std_logic_vector(7 downto 0);
  variable reg49 : std_logic_vector(7 downto 0);
  variable reg50 : std_logic_vector(7 downto 0);
  variable reg51 : std_logic_vector(7 downto 0);
  variable reg52 : std_logic_vector(7 downto 0);
  variable reg53 : std_logic_vector(7 downto 0);
  variable reg54 : std_logic_vector(7 downto 0);
  variable reg55 : std_logic_vector(7 downto 0);
  variable reg56 : std_logic_vector(7 downto 0);
  variable reg57 : std_logic_vector(7 downto 0);
  variable reg58 : std_logic_vector(7 downto 0);
  variable reg59 : std_logic_vector(7 downto 0);
  variable reg60 : std_logic_vector(7 downto 0);
  variable reg61 : std_logic_vector(7 downto 0);
  variable reg62 : std_logic_vector(7 downto 0);
  variable reg63 : std_logic_vector(7 downto 0);
  variable reg64 : std_logic_vector(7 downto 0);
  variable reg65 : std_logic_vector(7 downto 0);
  variable reg66 : std_logic_vector(7 downto 0);
  variable reg67 : std_logic_vector(7 downto 0);
  variable reg68 : std_logic_vector(7 downto 0);
  variable reg69 : std_logic_vector(7 downto 0);
  variable reg70 : std_logic_vector(7 downto 0);
  variable reg71 : std_logic_vector(7 downto 0);
  variable reg72 : std_logic_vector(7 downto 0);

begin
  if(reset='1') then
    payload_search_string <= (others => '0');
    rena <= '0';
    cnt := "00";
    empty_late := '0';

    valid1 := '0';
    valid2 := '0';
    valid3 := '0';
    valid4 := '0';
    valid5 := '0';
    valid6 := '0';
    valid7 := '0';
    valid8 := '0';
    valid9 := '0';
    valid10 := '0';
```

```
valid11 := '0';
valid12 := '0';
valid13 := '0';
valid14 := '0';
valid15 := '0';
valid16 := '0';
valid17 := '0';
valid18 := '0';
valid19 := '0';
valid20 := '0';
valid21 := '0';
valid22 := '0';
valid23 := '0';
valid24 := '0';
valid25 := '0';
valid26 := '0';
valid27 := '0';
valid28 := '0';
valid29 := '0';
valid30 := '0';
valid31 := '0';
valid32 := '0';
valid33 := '0';
valid34 := '0';
valid35 := '0';
valid36 := '0';
valid37 := '0';
valid38 := '0';
valid39 := '0';
valid40 := '0';
valid41 := '0';
valid42 := '0';
valid43 := '0';
valid44 := '0';
valid45 := '0';
valid46 := '0';
valid47 := '0';
valid48 := '0';
valid49 := '0';
valid50 := '0';
valid51 := '0';
valid52 := '0';
valid53 := '0';
valid54 := '0';
valid55 := '0';
valid56 := '0';
```

```
    valid57 := '0';
    valid58 := '0';
    valid59 := '0';
    valid60 := '0';
    valid61 := '0';
    valid62 := '0';
    valid63 := '0';
    valid64 := '0';
    valid65 := '0';
    valid66 := '0';
    valid67 := '0';
    valid68 := '0';
    valid69 := '0';
    valid70 := '0';
    valid71 := '0';
    valid72 := '0';

  elsif(rising_edge(clock)) then
    if(shift_payload = '1') then
      valid70 := '0';
      valid69 := '0';
      valid68 := '0';
      valid67 := '0';
      valid66 := '0';
      valid65 := '0';
      valid64 := '0';
      valid63 := '0';
      rena <= '1';


    elsif(valid70='0' and cnt="10") then
      reg72 := reg71;
      valid72 := valid71;
      reg71 := reg70;
      valid71 := valid70;
      reg70 := reg69;
      valid70 := valid69;
      reg69 := reg68;
      valid69 := valid68;
      reg68 := reg67;
      valid68 := valid67;
      reg67 := reg66;
      valid67:= valid66;
      reg66 := reg65;
      valid66 := valid65;
      reg65 := reg64;
```

```
valid65 := valid64;
reg64 := reg63;
valid64 := valid63;
reg63 := reg62;
valid63 := valid62;
reg62 := reg61;
valid62 := valid61;
reg61 := reg60;
valid61 := valid60;
reg60 := reg59;
valid60 := valid59;
reg59 := reg58;
valid59 := valid58;
reg58 := reg57;
valid58 := valid57;
reg57 := reg56;
valid57:= valid56;
reg56 := reg55;
valid56 := valid55;
reg55 := reg54;
valid55 := valid54;
reg54 := reg53;
valid54 := valid53;
reg53 := reg52;
valid53 := valid52;
reg52 := reg51;
valid52 := valid51;
reg51 := reg50;
valid51 := valid50;
reg50 := reg49;
valid50 := valid49;
reg49 := reg48;
valid49 := valid48;
reg48 := reg47;
valid48 := valid47;
reg47 := reg46;
valid47:= valid46;
reg46 := reg45;
valid46 := valid45;
reg45 := reg44;
valid45 := valid44;
reg44 := reg43;
valid44 := valid43;
reg43 := reg42;
valid43 := valid42;
reg42 := reg41;
```

```
valid42 := valid41;
reg41 := reg40;
valid41 := valid40;
reg40 := reg39;
valid40 := valid39;
reg39 := reg38;
valid39 := valid38;
reg38 := reg37;
valid38 := valid37;
reg37 := reg36;
valid37:= valid36;
reg36 := reg35;
valid36 := valid35;
reg35 := reg34;
valid35 := valid34;
reg34 := reg33;
valid34 := valid33;
reg33 := reg32;
valid33 := valid32;
reg32 := reg31;
valid32 := valid31;
reg31 := reg30;
valid31 := valid30;
reg30 := reg29;
valid30 := valid29;
reg29 := reg28;
valid29 := valid28;
reg28 := reg27;
valid28 := valid27;
reg27 := reg26;
valid27:= valid26;
reg26 := reg25;
valid26 := valid25;
reg25 := reg24;
valid25 := valid24;
reg24 := reg23;
valid24 := valid23;
reg23 := reg22;
valid23 := valid22;
reg22 := reg21;
valid22 := valid21;
reg21 := reg20;
valid21 := valid20;
reg20 := reg19;
valid20 := valid19;
reg19 := reg18;
```

```
        valid19 := valid18;
        reg18 := reg17;
        valid18 := valid17;
        reg17 := reg16;
        valid17:= valid16;
        reg16 := reg15;
        valid16 := valid15;
        reg15 := reg14;
        valid15 := valid14;
        reg14 := reg13;
        valid14 := valid13;
        reg13 := reg12;
        valid13 := valid12;
        reg12 := reg11;
        valid12 := valid11;
        reg11 := reg10;
        valid11 := valid10;
        reg10 := reg9;
        valid10 := valid9;
        reg9 := reg8;
        valid9 := valid8;
        reg8 := reg7;
        valid8 := valid7;
        reg7 := reg6;
        valid7:= valid6;
        reg6 := reg5;
        valid6 := valid5;
        reg5 := reg4;
        valid5 := valid4;
        reg4 := reg3;
        valid4 := valid3;
        reg3 := reg2;
        valid3 := valid2;
        reg2 := reg1;
        valid2 := valid1;

        if(empty_late='1') then
          reg1 := ffout;
          valid1 := '1';
        else
          reg1 := "00000000";
          valid1 := '0';
        end if;

        payload_search_string <= reg70 & reg69 & reg68 & reg67 & reg66 & reg65 & reg64 &
reg63 & reg62 & reg61 & reg60 & reg59 & reg58 & reg57 & reg56 & reg55 & reg54 & reg53 &
```

```
reg52 & reg51 & reg50 & reg49 & reg48 & reg47 & reg46 & reg45 & reg44 & reg43 & reg42 &
reg41 & reg40 & reg39 & reg38 & reg37 & reg36 & reg35 & reg34 & reg33 & reg32 & reg31 &
reg30 & reg29 & reg28 & reg27 & reg26 & reg25 & reg24 & reg23 & reg22 & reg21 & reg20 &
reg19 & reg18 & reg17 & reg16 & reg15 & reg14 & reg13 & reg12 & reg11 & reg10 & reg9 &
reg8 & reg7 & reg6 & reg5 & reg4 & reg3 & reg2 & reg1 & "00000000" & "00000000";

        if(valid70='0') then
          rena <= '1';
        else
          rena <= '0';
        end if;

      elsif(empty='1' and valid70='0') then
        rena <= '1';
        cnt := cnt+1;

      else
        rena <= '0';
      end if;

      empty_late := empty;
    end if;
  end process;

  -- HDL Embedded Text Block 13 header_cam_mux

  init_bus <= LS_init & LC_init & Trst_b_init & Tck_init & Tdi_init & Tms_init & Crb_init &
Gmask_init & MatchIn_init & Inst_init & ReqStb_init & Burst_b_init & Rst_b_init & Dev_init
& Phasen_init & clock_init;

  ctrl_bus <= LS & LC & Trst_b & Tck & Tdi & Tms & Crb & Gmask & MatchIn & Inst &
ReqStb & Burst_b & Rst_b & Dev & Phasen & clock;

  -- This concurrent statement assembles information from the packet_field_register and all of
the special case output into a search string that will be used for header searching.

  header_search_string <= ip_src & ip_dest & src_port & dest_port & ip_type &
ip_identification & ip_flags & ip_offset & ip_ttl & ip_protocol & ip_options & tcp_seq_num &
tcp_ack_num & tcp_flags & tcp_window_size & icmp_type & icmp_code & icmp_id &
icmp_seq & ethernet_type & payload_len(10 downto 0) & src_port_not_21_to_23 &
src_port_not_80 & dest_port_not_80 & dest_port_gt_499 & src_port_gt_1023 &
dest_port_gt_1023 & src_port_between_1000_1300 & dest_port_between_1000_1300 &
dest_port_between_3127_3199 & src_port_between_6666_7000 &
dest_port_between_6666_7000 & dest_port_between_32772_34000 &
dest_port_between_32771_34000 & ip_ttl_gt_220 & ip_proto_gt_134 & same_ip & dsize_gt_1
& dsize_gt_6 & dsize_lt_25 & dsize_gt_100 & dsize_gt_128 & dsize_gt_156 & dsize_gt_200 &
```

```
dsize_gt_500 & dsize_gt_512 & dsize_gt_720 & dsize_gt_800 & dsize_gt_1000 &
dsize_gt_1023 & dsize_gt_1445 & icmp_code_gt_0 & icmp_code_gt_1 & icmp_code_gt_2 &
icmp_code_gt_3 & icmp_code_gt_15 & padding;


  -- ModuleWare code(v1.0) for instance 'I1' of 'fifo'
  ffout <= mw_I1creg(0);
  I1seq1: PROCESS (clock)
  BEGIN
    IF (clock'EVENT AND clock='1') THEN
      FOR i IN 0 TO 1500 LOOP
        mw_I1creg(i)(7 DOWNTO 0) <= mw_I1nreg(i)(7 DOWNTO 0);
      END LOOP;
    END IF;
  END PROCESS I1seq1;

  I1seq2: PROCESS (clock, reset)
  BEGIN
    IF (reset = '1') THEN
      mw_I1caddr <= 0;
    ELSIF (clock'EVENT AND clock='1') THEN
      mw_I1caddr <= mw_I1naddr;
    END IF;
  END PROCESS I1seq2;

  I1combo: PROCESS (reset, rena, wena, mw_I1caddr, mw_I1creg, field_data(7 DOWNTO 0))
  VARIABLE trena : std_logic;
  VARIABLE twena : std_logic;
  VARIABLE tfull : std_logic;
  VARIABLE tempty : std_logic;
  BEGIN
    IF (mw_I1caddr = 1500) THEN
      tfull := '1';
      tempty := '0';
    ELSIF (mw_I1caddr = 0) THEN
      tfull := '0';
      tempty := '1';
    ELSE
      tfull := '0';
      tempty := '0';
    END IF;
    trena := NOT(reset) AND rena AND NOT(tempty);
    twena := NOT(reset) AND wena AND NOT(tfull);

    IF (twena = '1' OR twena = 'H') THEN
      IF (trena = '1' OR trena = 'H') THEN
        mw_I1naddr <= mw_I1caddr;
      ELSE
        mw_I1naddr <= mw_I1caddr + 1;
      END IF;
    ELSIF (trena = '1' OR trena = 'H') THEN
      mw_I1naddr <= mw_I1caddr - 1;
    ELSE
      mw_I1naddr <= mw_I1caddr;
    END IF;

    IF (twena = '1' OR twena = 'H') THEN
      IF (trena = '1' OR trena = 'H') THEN
        mw_I1nreg(1500)(7 DOWNTO 0) <= mw_I1creg(1500)(7 DOWNTO 0);
        FOR i IN 0 TO 1499 LOOP
          IF (mw_I1caddr = i) THEN
            mw_I1nreg(i)(7 DOWNTO 0) <= field_data(7 DOWNTO 0);
          ELSE
            mw_I1nreg(i)(7 DOWNTO 0) <= mw_I1creg(i+1)(7 DOWNTO 0);
          END IF;
        END LOOP;
      ELSIF (trena = '0' OR trena = 'L') THEN
        mw_I1nreg(0)(7 DOWNTO 0) <= mw_I1creg(0)(7 DOWNTO 0);
        FOR i IN 0 TO 1499 LOOP
          IF (mw_I1caddr = i) THEN
            mw_I1nreg(i+1)(7 DOWNTO 0) <= field_data(7 DOWNTO 0);
          ELSE
            mw_I1nreg(i+1)(7 DOWNTO 0) <= mw_I1creg(i+1)(7 DOWNTO 0);
          END IF;
        END LOOP;
      END IF;
    ELSIF (twena = '0' OR twena = 'L') THEN
      IF (trena = '1' OR trena = 'H') THEN
        FOR i IN 0 TO 1499 LOOP
          mw_I1nreg(i)(7 DOWNTO 0) <= mw_I1creg(i+1)(7 DOWNTO 0);
        END LOOP;
          mw_I1nreg(1500)(7 DOWNTO 0) <= mw_I1creg(1500)(7 DOWNTO 0);
      ELSIF (trena = '0' OR trena = 'L') THEN
        FOR i IN 0 TO 1500 LOOP
          mw_I1nreg(i)(7 DOWNTO 0) <= mw_I1creg(i)(7 DOWNTO 0);
        END LOOP;
    ELSE
        FOR i IN 0 TO 1500 LOOP
          mw_I1nreg(i)(7 DOWNTO 0) <= (OTHERS => 'X');
        END LOOP;
    END IF;
    ELSE
```

```
      FOR i IN 0 TO 1500 LOOP
        mw_I1nreg(i)(7 DOWNTO 0) <= (OTHERS => 'X');
      END LOOP;
    END IF;
    full <= tfull;
    empty <= NOT(tempty);
    aempty <= NOT(tempty);
  END PROCESS I1combo;

  -- ModuleWare code(v1.0) for instance 'I4' of 'gnd'
  padding <= (OTHERS => '0');

  -- Instance port mappings.

END struct;
```

BIBLIOGRAPHY

[1] W. Bux, W.E. Denzel, T. Engbersen, A. Herkersdorf, R.P. Luijten, "Technologies and Building Blocks for Fast Packet Forwarding," *IEEE Communications Magazine*, Jan. 2001, pp. 70-77.

[2] B. Acohido. "Agency raises the bar on tech security," *USA TODAY*, [Online document] Feb. 2002, Available at HTTP: http://www.usatoday.com/life/cyber/tech/2002/02/27/ security.htm.

[3] "IDC Names ISS Worldwide Leader in IDS," *Help Net Security*, [Online document] June 28, 2001, Available at HTTP: http://www.net-security.org/text/press/ 984080961,95550,.shtml.

[4] "Security software to total billions by 2004," [Online document] Apr.19, 2001, Available at HTTP: http://www.landfield.com/isn/mail-archive/2001/Apr/0123.html.

[5] "IDC Forecasts Worldwide Intrusion Detection and Vulnerability Assessment Software Revenues Will Exceed $1 billion in 2003," *Fitzgerald Communications Inc.*, [Online document] Apr. 18, 2001, Available at HTTP: http://www.bindview.com/news/images/ IDCRelease041801.pdf.

[6] Computer Economics, [Online], [2004 Aug 31], Available at HTTP: www.computereconomics.com.

[7] "CSI/FBI Computer Crime and Security Survey," *SI Computer Security Issues & Trends*, 2002.

[8] S. Lodin. "Intrusion detection Product Evaluation Criteria," *Ernst & Young LLP*, [Online document] 1998, Available at HTTP: docshow.net/ids.htm.

[9] T. Heberlein, K. Levitt, B. Mukherjee. "A Method to Detect Intrusive Activity in a Networked Environment," Proceedings of the 14th National Computer Security Conference, 1991, pp. 362-372.

[10] T. Heberlien, B. Mukherjee, K.N. Levitt, G. Dias, D..Mansur. "Towards Detecting Intrusions in a Networked Environment," Proceedings of the Fourteenth Department of Energy Computer Security Group Conference, 1991, pp. 47-66.

[11] R. Zalenski. "Firewall technologies," *IEEE Potentials*, Feb-Mar 2002, pp. 24-29.

[12] P. Glaskowsky. “Network Processors Multiply,” *Microprocessor Report*, Jan 2001, pp. 37.

[13] S. Cobb. “NCSA Firewall Policy Guide, Version 2.0,” [Online document], Available at HTTP: http://cobb.com/firewalls/.

[14] T. Sato, M. Fukase. “Reconfigurable hardware implementation of host-based IDS,” Sept 2003, pp. 849-853.

[15] “Network Processor Designs for Next-Generation Networking Equipment,” White Paper, [Online document], Available at HTTP: http://www.ezchip.com/html/ tech_nsppaper.html.

[16] M. S. Sheshadri, J. Bent, T. Kosar. “Intelligent Routing using Network Processors: Guiding Design through Analysis,” Technical Report CS-TR-2003-1480, Computer Sciences Department, University of Wisconsin, April 2003.

[17] F. Gong. “Next Generation Intrusion Detection Systems (IDS),” Network Associates White Paper, March 2002.

[18] C.J. Coit, S. Staniford, J. McAlerney. “Towards faster string matching for intrusion detection or exceeding the speed of Snort,” DARPA Information Survivability Conference & Exposition II, 2001, DISCEX '01 Proceedings, Volume 1, 2001, pp. 367-373.

[19] B. Yuebin, H. Kobayashi. “New string matching technology for network security,” Advanced Information Networking and Applications, 2003, 17th International Conference, March 27-29, 2003, pp. 198-201.

[20] “75K62100_NSE Datasheet Brief, Integrated Device Technologies”, [Online document] June 24, 2003, Available at HTTP: http://www1.idt.com/pcms/tempDocs/ 75K62100_DS_34483.pdf.

[21] Snort, [Online], [2004 Aug 31], Available at HTTP: http://www.snort.org.

[22] D. V. Schuehler, J. Moscola, J. Lockwood. “Architecture for a Hardware Based, TCP/IP Content Scanning System,” Proceedings of the 11th Symposium on High Performance Interconnects, 2003.

[23] J. Moscola, J. Lockwood, R.P. Loui, and M. Pachos. “Implementation of a content-scanning module for an internet firewall,” IEEE Symposium on Field-Programmable Custom Computing machines (FCCM), Napa, CA, April 2003.

[24] B.L. Hutchings, R. Franklin, D. Carver. “Assisting Network Intrusion Detection with Reconfigurable Hardware,” Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 20002.

[25] R. Sidhu, V.K. Prasanna. “Fast Regular Expression Matching Using FPGAs,” Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, April 2001.

[26] S. Li, J. Torresen, O. Soraasen. “Exploiting Reconfigurable Hardware for Network Security,” Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2003.

[27] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, V. Hogsett. “Granidt: Towards Gigabit Rate Network Intrusion Detection Technolog,” FPL 2002, LNCS 2438, pp. 404-413, 2002.

[28] Y.H. Cho, S. Navab, W.H. Mangione-Smith. “Specialized Hardware for Deep Network Packet Filtering,” FPL 2002, LNCS 2438, pp. 452-461, 2002.

[29] N. Huang, W. Chen, J. Luo, J. Chen. “Design of Multi-field IPv6 Packet Classifiers Using Ternary CAMs,” IEEE 2001.

[30] TCPDUMP Public Repository, [Online], [2004 Aug 31], Available at HTTP: http://www.tcpdump.org.

[31] Talisker Switch Port Mirroring, [Online], [2004 Aug 31], Available at HTTP: http://www.networkintrusion.co.uk/switch.htm.