Hardware Based Packet Filtering Using FPGAs

Submitted in partial fulfilment
of the requirements of the degree of

Bachelor of Science Honours in Computer Science

of Rhodes University

Timothy Whelan

Grahamstown, South Africa
01 November 2010

Abstract

This paper seeks to understand the operation of packet filters implemented on FPGAs as high speed platforms and evaluates the suitability of FPGAs as platforms for packet filter implementation. It must be remembered that packet filters need to operate at line speeds or else they become detrimental to the health of any network they may be placed upon by introducing significant delays. This task is approached by a novice to the world of FPGAs and hardware description languages and begins with a review of other implementations of packet filters on FPGAs. The design of a simple packet filter that filters packets based upon transport protocol, IP addresses and port numbers is presented. The packet filter in this work is not realised on a FPGA device but its logical design is verified with the use of ModelSim Starter Edition. The end product of this endeavour is a kernel upon which the filter can be expanded and the reaffirmation that FPGAs are successful candidates for packet filter platforms.

ACM Computing Classification System Classification

Categories and subject descriptors:

B.2.m [Arithmetic and Logic Structures]: FPGA Platform

C.3 [Special Purpose and Application-Based Systems]: Dedicated hardware

General Terms: Packet filtering, VHDL

Acknowledgements

I thank Shaun Bangay for his continued help throughout the year on matters relating to VHDL. My thanks also to the Rhodes University Departement of Physics and Electronics for the loan of their equipment and expertise on the subject of hardware interfaces. I would also like to thank my supervisor, Mr Barry Irwin, for putting himself up to the task of guiding me through the year. My parents deserve a huge thank you for continuing to support me in whatever I do. I thank Telkom SA, Comverse SA, Stortech, Tellabs, Easttel, Bright Ideas Projects 39 and THRIP who provided financial and technical support through the Telkcom Centre of Excellence at Rhodes University. I also wish to thank the National Research Foundation for their financial contribution to my project too.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As information systems increase in size and complexity the task of data acquisition and processing in and of such systems becomes ever more arduous and computationally expensive. To cope with the sheer size of the tasks required new and faster techniques and systems are required to be developed; often appearing as either software packages or primarily hardware based systems such as the systems described in this paper. The particular form of data processing discussed in this paper is that of packet filtering. In many network management areas it is advantageous to be able to leverage efficient techniques in packet filtering to monitor traffic behaviour at the boundary of a network. Other tools, such as network telescopes collect vast numbers of packets which can then be examined at a later stage for effects of events around the Internet such as back-scatter from attacks on other networks. However, typically such analysis will only find a small portion of the collected packets relevant but the entire set needs to be searched exhaustively to locate the relevant packets. Hence a tool is required to quickly match a sample packet against a set of criteria.

1.1 Problem Statement

The problem of packet classification resides in the known structure of packet headers from network and transport layers. Packet structure will be described in detail later in this paper but in short a packet header is comprised of multiple fields with varying values for each packet. Packets are classified according to the positive or negative matching of N fields of the packet (an N-tuple) to a series of values. Each N-tuple of values to be matched

against is termed a rule and the collection of rules is a rule set. Typically packets are classified according to a 5-tuple consisting of source and destination IP addresses, source and destination port numbers and the protocol number of the packet. Once a packet has been classified according to one or more rules a course of action can be determined for the classified packet.

1.2 Research Goal

The aim of this project is to produce a packet filter implemented on an FPGA that can accept an IP packet from an Ethernet connection. The FPGA must then match the packet to one or more rules in a rule set and record a count of how many times each rule in the rule set has been matched. A count of the rules matched will be sent to a pre-determined network node upon request via a serial RS-232 interface.

1.3 Motivation for Research

While software packages such as Snort IDS have been developed to perform packet filtering they require innovative algorithms to overcome time spent waiting for resources shared with other processes such as CPU's. On the other hand hardware systems can remain dedicated to the task for which they have been designed and therefore have a significant advantage over software systems in this regard. However hardware systems often require expensive design and production procedures so if a relatively cheap technology can be developed and used for data processing, and particularly designed for packet filtering, then the computing power of hardware can be focused on a single application while maintaining cost-effectiveness.

These objectives together with the requirement that the system is capable of regular updates without significant effort leads one to examine FPGA's as a possible platform on which to develop such a system. FPGA's have been used before as platforms for rapid prototyping to test hardware configurations before creating the tested design as an application specific integrated circuit (ASIC), FPGA's behave similarly to ASIC's without eliminating the option of altering parts of the design after implementation – in the systems described in this paper such alterations correspond to changing the rule set of the packet filter. The development of such a system would be beneficial to any task requiring the rapid processing and filtering of packets in an affordable manner.

Chapter 2

Literature review

This literature review briefly describes the role of network telescopes before detailing the operation of FPGA's. Then a few algorithms for packet classification and deep packet inspection are described. Lastly related work describing previous projects that implement various packet processing tasks on FPGA's is reviewed to demonstrate the suitability of FPGA's to this field of research.

2.1 Description of Network Telescopes

A network telescope resides on a portion of IP space which should receive very little legitimate internet traffic [13]. Network telescopes passively collect data from networks and the data collected can be used to observe events on the Internet. As can be seen in [27], useful attributes of network traffic one might measure include source and destination IP addresses and destination port. Reasonably, one might wish to extend this list to include source port addresses and protocols used in the packet if they can be identified.

2.2 Description of FPGA's

As described by the Altera Corporation in [1] FPGA's can fulfil any programmable function that could be implemented on an application specific integrated circuit (ASIC) but also provide the ability to reconfigure the FPGA to provide altered functionality. Whilst

many other pieces of hardware exist that can be programmed to perform a certain function, and these hardware configurations can also be reprogrammed, FPGA's have become important in industry as the only field programmable logic devices that can provide a very high logic capacity on a single chip [4]. Simple programmable logic devices (PLD's) typically execute product and sum instructions on two levels of programmability; an AND-plane that AND's inputs together in an effective multiplication and an OR-plane that may or may not be programmable and which OR's its inputs [4, 10]. PLD's operate on two-state logic levels but FPGA's are designed to be able to operate using multi-level logic. The consequence of this design is that the complexity of the computational circuitry that can be programmed on an FPGA greatly exceeds that which can be placed on a PLD. Figure 2.1 shows a simplified structure of FPGAs.

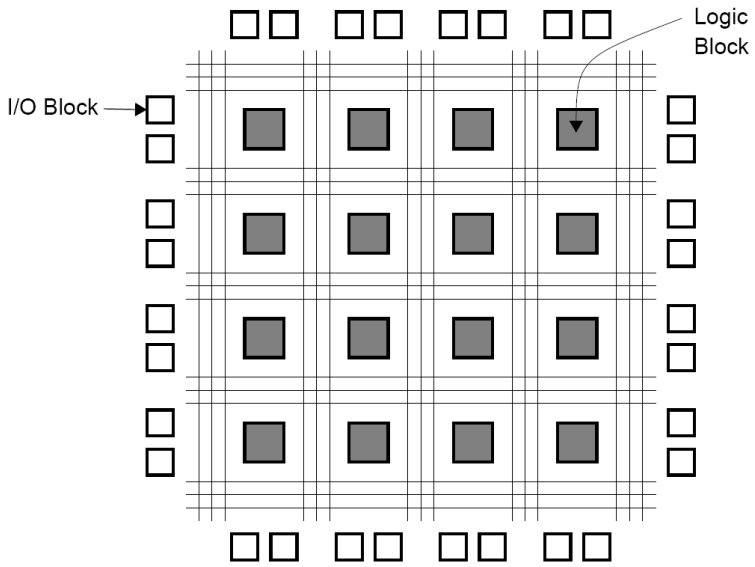

Figure 2.1: Structure of an FPGA [4]

2.2.1 Memory in FPGAs

The structure of an FPGA consists of configurable logic blocks with various interconnects between the configurable logic blocks as shown in figure 2.1 [10]. To provide functionality to an FPGA one specifies interconnects there by linking the different logic components. FPGA's retain their functionality in three ways; the instructions programmed on an FPGA are stored in antifuse configurations, EPROM/EEPROM or SRAM blocks [10]. What follows is a brief description of three memory technologies used in FPGA's as illustrated in [10].

2.2.1.1 Antifuses

Antifuses are semi-conductor components that provide electronic insulation until a sufficiently high voltage is placed across the device at which point the insulating ability of the components of the device is destroyed – the antifuse behaves as a typical fuse but is meant to be 'fused' to allow current to flow through the device. Antifuses however cannot be 'unfused' hence once the antifuse block of an FPGA has been programmed it cannot be reprogrammed [10]. Figure 2.2 provides a diagrammatic sketch of an antifuse.

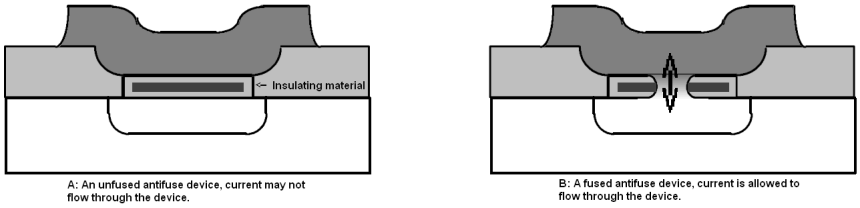

Figure 2.2: An antifuse semiconducting device (adapted from [10]).

2.2.1.2 Static Random Access Memory

To allow the FPGA to be reprogrammed it may be provided with blocks of static random access memory, or SRAM. SRAM bit construction is shown in figure 2.3. When a write instruction, logic 1, is placed on the gate of the control transistor data can flow into the inverter system which maintains any logic level placed in it. When logic 0 is placed on the gate of the control transistor then no change can occur to the logic placed in the inverter system while it is supplied with power. Should power to the SRAM bit be lost however then the data written to the SRAM bit will be lost. Therefore a weakness of FPGA's that are built using SRAM blocks of memory is that each time the device is powered on it must be reprogrammed.

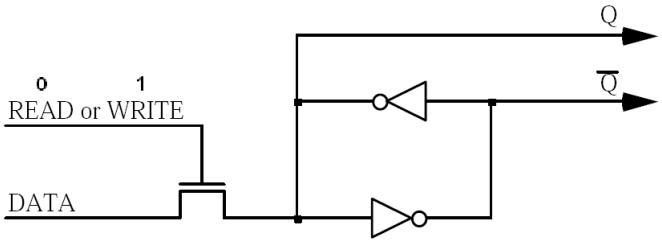

Figure 2.3: A single bit of SRAM [10]

2.2.1.3 Erasable Programmable Read Only Memory (EEPROM)

The weakness of using volatile SRAM in FPGA's can be mitigated by making use of EPROM. In short, a voltage is placed across the gate structure of an EPROM bit thereby inducing a change in the internal structure of the semi-conductor. This change in structure can be reversed by shining UV light on the device which causes the device to revert back to its original state. The advantage of using EPROM over SRAM as the memory that stores the functionality of the FPGA is that the change in the device's structure does not need to be maintained by supplying power to the device which means that the device need not be programmed upon powering up but rather only once when the program logic is first transferred to the device.

2.2.2 General FPGA Structure

The general structure of FPGA's is that of many configurable logic blocks (CLB's) that are interconnected by routing lines of different types that perform slightly different routing functions [29, 10]. Figure 2.4 shows the structure of a Xilinx Spartan-II FPGA's CLB slice, a portion of the CLB. There are two such slices on a CLB [29]. What follows is the structure of the Xilinx Spartan-II family of FPGAs taken from the Spartan-II databook [29]. This description, while for a specific class of FPGA, serves to demonstrate the general principles on which FPGA's operate.

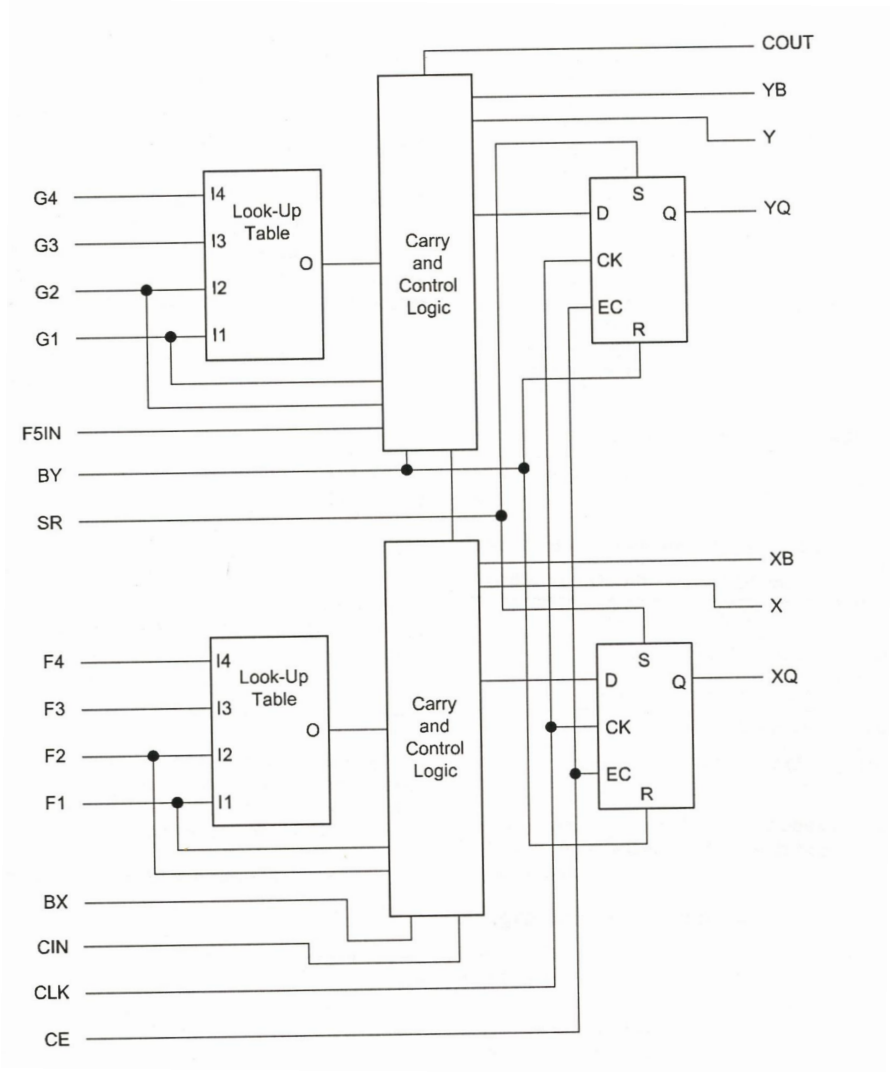

Figure 2.4: Spartan-II CLB slice

As can be seen in figure 2.4, a CLB slice is composed of a look up table (LUT), carry and control circuitry and a D-type flip-flop which can store the output from the carry and control circuitry. This configuration is termed a logic cell (LC) and a CLB may have one or more LC's that together make up a slice of a CLB. Multiple slices then make up a CLB. To program the FPGA with certain functionality the FPGA design software, normally supplied by the FPGA vendor, maps the high level logic designed by the FPGA programmer into multiple single-step functions and these functions are then mapped onto the LUT. The LUT configurations are stored using one of the memory technologies discussed above. LUT's function by specifying the required output for all inputs to the LUT. The results from CLB's are then routed around the FPGA using the available routing paths placed on the FPGA. Routing on the FPGA is provided to

perform various different functions such as:

- routing between LC's in a CLB to minimise routing delays for functions requiring more than one LUT to describe,
- direct routing paths between adjacent CLB's for fast binary arithmetic between adjacent CLB's,
- routing channels through general routing matrices that form meeting points for horizontal and vertical routing paths that also connect adjacent routing matrices – general routing matrices specify whether or not connections exist between the lines entering the general routing matrix,
- prioritised routing paths for clock signals to decrease clock skew and clock delays.

2.2.3 Example FPGA Applications

FPGA's are suitable for a number of high speed processing applications, especially ones where components can operate in parallel. What follows is a partial list of applications of FPGA's:

1. FPGA's have been successful platforms on which to implement artificial neural networks [35].
2. The authors of [7] demonstrate a system developed to implement the Advanced Encryption Standard encryption algorithm at a rate of 150Mbs.
3. FPGA's have proved a viable replacement for dedicated digital signal processing (DSP) chips and ASIC's [9].
4. Central to the motivation for this literature review is the work done in using FPGA's to perform packet classification and deep packet inspection [6, 3, 14, 22, 34]. As can be seen in the sources mentioned, deep packet inspection also often involves string matching within the payloads of packets.

The focus of the rest of this literature review will focus on FPGA based methods for packet processing such as packet classification and deep packet inspection.

2.3 FPGA Methods for Packet Processing

As mentioned before, the motivation for this literature review is to examine and assess work carried out in the field on network packet processing particularly on FPGA platforms. The literature reviewed decomposes packet processing into two separate actions: packet classification, typically based on packet headers, and deep packet inspection, typically string matching within packet payloads to identify data being transferred in the packet [21, 22].

2.3.1 Packet Classification

To understand the desire to research improved methods for processing of packet headers one can examine the uses of packet headers and the decisions that can be taken pertaining to the handling of a packet once its headers have been inspected. Song and Lockwood mention the use of network packet headers in NIDS in [21]. The rules used in the Snort IDS typically include a 5-tuple of packet headers; source and destination IP addresses and port numbers and the protocol being used [21]. The IDS also uses pattern matching, as described later in this review, when matching rules but uses the header processing to provide a context in which the pattern matching results can be better utilized [21]. Ravindran *et al* describe packet routing and forwarding as another application that benefits from improved methods in header processing [18].

There are many techniques for packet classification based upon multiple header fields of a packet and a description of these techniques can be found in [17, 26]. A brief range of these methods is related below.

Linear Search Suppose that each filter in a set of filters were sequentially compared with the headers of an inspected packet until either a filter is found that matches the packet's header fields or all filters are unsuccessfully matched to the filter; this would constitute an exhaustive, linear search for filters that match the packet headers [17]. This approach to packet classification is straight-forward and reliable [17, 26] but provides poor performance [26] and makes poor use of memory [17, 26] with O(N) memory requirements and performance decreasing proportionally to the size of the filter set [26]. Taylor does demonstrate a means of reducing the memory requirements of a filter set when using exhaustive search techniques but the computational behaviour of exhaustive

linear searches still make it prohibitively slow for applications requiring high throughput though it can be used in conjunction with other algorithms in a more complex system [17, 26]. Linear search techniques do lend themselves well to systems that can provide parallel processing [17]. Taylor suggests that if a linear search is at one end of a spectrum of searching techniques, with linear memory and search computational complexities, then ternary content addressable memory would be at the opposite end of the spectrum with O(1) computational complexity. Ternary content addressable memory is discussed further below.

Ternary Content Addressable Memory (TCAM) TCAM is a type of memory that can store a "don't care" state as well as a 1 or 0 bit value in the memory [21] hence it is a tri-state memory as described by the word "ternary" in its name. The operation of CAM memory is practically the converse of conventional memory; a desired piece of data contained in memory is specified and what is returned is the address or a list of multiple addresses where the content resides in memory [30]. TCAM is also optimised to search for content in memory at multiple locations in parallel [26, 30] and all these features of TCAM memory make it an obvious tool in packet classification based on header processing as indicated in the related work mentioned in [21] which will be discussed later to illustrate a hybrid algorithmic approach to the problem of packet classification. However, TCAM technology has significant drawbacks as described by Taylor [26]:

- TCAM memory has a greater cost per bit than other memory technologies.
- TCAM memory suffers from inefficient use of storage space because it cannot store arbitrary ranges of numbers but instead such ranges need to be converted into '<x' and '>y' descriptions (where x and y are bit string prefixes) and extra resources are required to store the "don't care" value.
- TCAM memory makes use of more transistors than other memories leading to greater power consumption.
- TCAM memory does not scale very well when input to the TCAM is long.

The above two techniques perform exhaustive searches over the entire filter set, albeit with vastly differing memory and time requirements. The following techniques perform restructuring of the filters and/or the incoming packet's fields to transform the packet classification problem into another class of problem and leverage observations that can be made about filter sets.

Grid-of-tries This approach is described more formally in [24] but a brief description is provided below. Suppose an example filter set is as in figure **??** (note that this initial set represents only source and destination addresses).

Table 2.1: Example filter rule set (from Srinivasan *et al [24]*)

Filter	Destination	Source
F1	0*	10*
F2	0*	01*
F3	0*	1*
F4	00*	1*
F5	00*	11*
F6	10*	1*
F7	*	00*

Firstly the desitination addresses of the filter are used to construct a tree structure as in figure 2.5. This enables any incoming packet to be tracked to a node in this tree (termed the dest-trie) except for the open circle which represents an invalid destination address prefix. The leaf nodes of the dest-trie contain pointers to similar trees that represent decision trees for source addresses relevant to the destination address represented at the originating leaf nodes of the dest-trie as shown in figure 2.6. where the reader can see the pointers from dest-trie to relevant source-trie filters. Note that source address filters are repeated when one dest-trie filter is a prefix of another dest-trie filter.

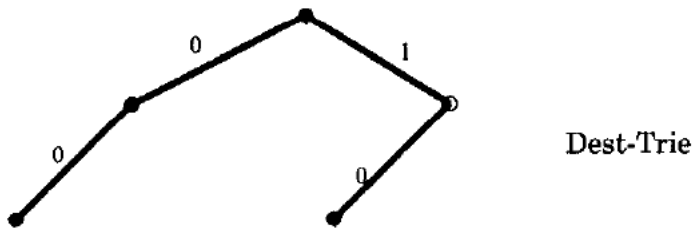

Figure 2.5: Dest-trie representing the source addresses in the filter set. [24]

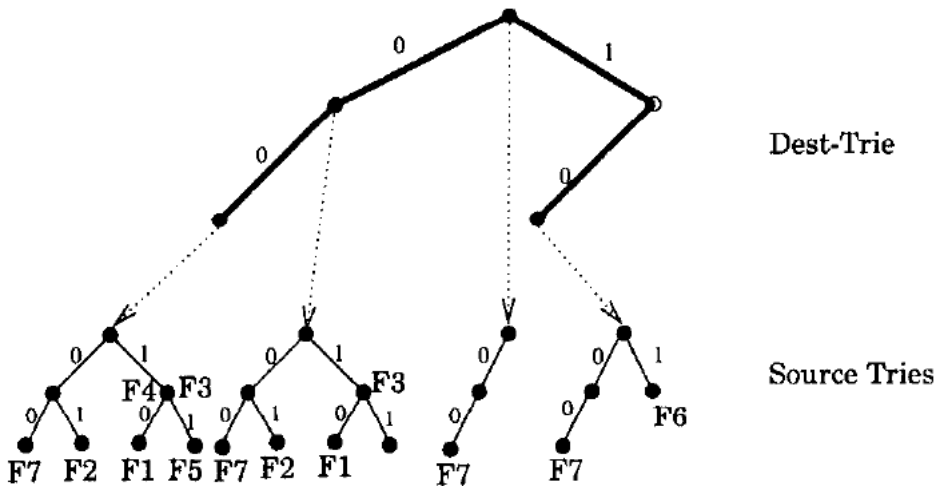

Figure 2.6: The general trie structure. [24]

The grid-of-tries structure described above can be optimised in many ways and Srinivasan *et al* go on to describe some of these optimisations. Notably, the memory requirements of this structure can be reduced by not repeating filters in multiple source-tries and search costs can be reduced by placing pointers from one source-trie node to nodes in other source-tries so that when a search down one source-trie branch fails a pointer can be followed to another source-trie and the search can continue without having to perform redundant searches along higher branches before progressing beyond what was already processed at the point of failure [24].

Tries however only work well for 2-dimensional filters [17, 26, 24]. The structure of tries also means that they are suitable for IP address masks but are not suitable for arbitrary port ranges without changing the ranges specifications into prefixes, a task which can significantly increase the number of tries required to adequately describe the desired filters [17, 26, 24]. As will be discussed later, the cost of using a trie structure to perform port filtering can be made acceptable provided other filter optimisation techniques are employed and tries have indeed been used in conjunction with other techniques such as cross-producting (to be discussed in the next section) in [24] and TCAM's in [21].

Cross-producting Srinivasan *et al* introduced another technique in [24] as well as the grid-of-tries approach to packet classification: cross-producting. Cross-producting operates by performing 'best matching prefix' searches on each field in the filter, possibly in parallel [17] given that these searches can be done independently [26], and then combing the results of these independent searches into one more search to provide a final overall

classification. While searching each field independently for a best matching prefix is relatively simple an efficient method is required to combine the search results into a final usable result [24]. The scheme proposed by Srinivasan *et al* proposes dividing filters into columns which group the different field values by field type. Then a cross-product is formed of these sets to create a list of all possible combinations of field values that can be made from the field values in the original filter set. An example to illustrate this process is in figure 2.7.

Cross-producting requires a prioritisation of fields to enable the final cross-products to be more closely aligned with one filter than another filter as once the cross-product of filters has been performed each cross-product is labelled with a filter that best matches the cross-product. When a packet is received and the closest field prefix for each field has been located independently then this combination of individual field filter prefixes is used as a hash to locate the closest matching filter from the cross-product table.

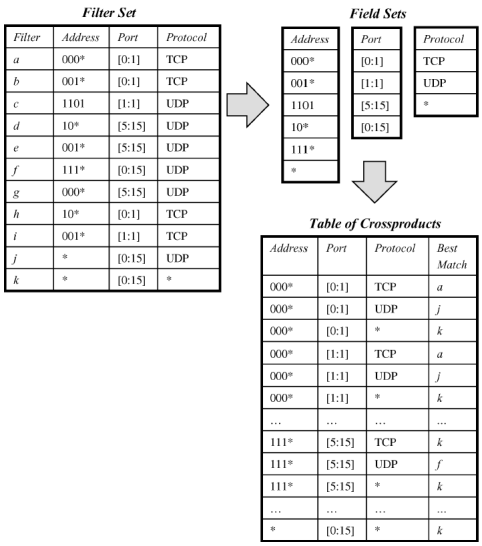

Figure 2.7: The process of forming cross-products (from Taylor[26]).

Cross-producting, while providing a high throughput, has exponential memory requirements, O(Nd) [17, 26]. Srinivasan *et al* therefore proposed a hybrid approach to packet classification using their grid-of-tries technique to classify based on destination and then source matching and cross-producting to match ports and flags [17, 24, 26].

Bit Vector The bit vector classification technique views the packet classification problem as a geometric problem with each D-field filter specifying a region in D-dimensional

space and the classification equates to locating which named region a packet can be said to be placed in [17, 26], an illustration of this view is given in figure 2.8. Filtering on two fields (Port number and address) results in a two dimensional filter. The shaded regions with letters represent filters and filters may overlap (darker regions).

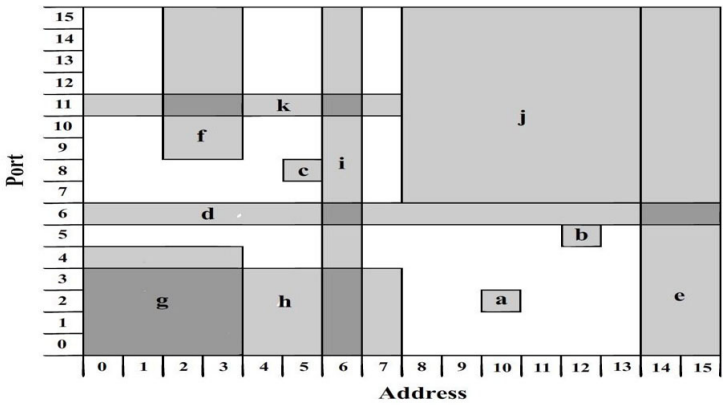

Figure 2.8: A geometric view of the packet classification problem. [17]

A brief, simplified, algorithm is adapted from [17] and [26] and described below. Note that there are N filters and D types of fields in each filter.

1. Prioritise the filter rules (if priority is irrelevant then priorities can be arbitrary)
2. For each axis
   (a) Divide the axis into regions bounded by the points where boundaries of regions in D-space that are orthogonal to the current axis may intersect the current axis.
   (b) For each region along the current axis
      i. Assign an N-bit bit vector equal to zero
      ii. If region x in D-space, described by rule x, lies on the current region
         A. Then set bit x of the current region's bit vector to 1

A graphical representation of a geometric view of the classification problem can be seen in figure 2.8 and a graphical representation of the result of the algorithm can be seen in figure 2.9.

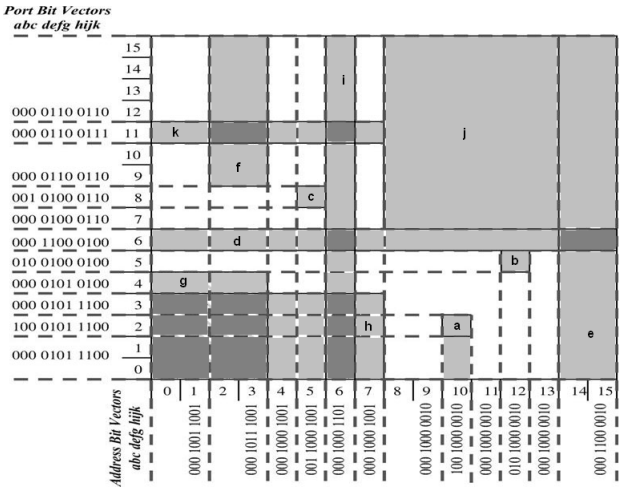

Figure 2.9: Regions in D-space showing filters and bit vectors for regions along each dimension (N = 11; D = 2) (Figure adapted from [17])

When an incoming packet is received its fields are decomposed and each one is processed independently of the others to locate which filter pertains to the packet with the given field value. Alternatively one can describe the process as locating which region along each of the D axes the packet might belong to. The D bit vectors (1 bit vector result from each field or dimension) are then AND'ed together to give a final bit vector result with each 1 bit representing a rule that pertains to the packet fields as a group e.g. a packet with port number 8 and address 5 will return bit vectors 001 0100 0110 and 001 1000 1001 as seen in figure 2.9. When these bit vectors are AND'ed the result will be 001 0000 0000 indicating that filter c classifies the packet. If the rules in each dimension are ordered according to some priority scheme then the significance of the 1 bits in the final bit vector will match the priority of the rules in the rule set. Li *et al* [12] describe a bit vector implementation that uses trie structures instead of taking the multidimensional geometric view described above.

Tuple Space Sirinivasan *et al* [23] introduced the tuple space classification algorithm and Taylor [26] provides a less formal description of the tuple space algorithm. The tuple space technique makes the observation that while there may be many filters in a filter set, the number of different lengths of prefixes in the fields of the filter is typically significantly less than the number of filters [26, 23]. For example, while a 32-bit IP address could conceivable provide 100's of different IP ranges and specific IP addresses to be used in a filter, there can only be 32 different lengths of IP address masks or prefixes in the filter set. Using this observation, a description of a filter in a tuple space technique indicates

the number of bits in the prefix describing each field in the filter and the concatenation of all these indicating numbers is called a tuple [26, 23]. An example from [23] supposes that if there are two 2-dimensional filters, F1 = (01*, 111*) and F2 = (11*, 010*) then a tuple which could describe the filters is [2, 3].

Port numbers in a filter set are frequently specified using ranges which make prefix representations inconvenient [23] hence Sirinivasan *et al* introduced an encoding scheme whereby port ranges are divided up into Nesting Levels with non-overlapping ranges and each range within a level is given a locally unique Range ID [26]. Port ranges are then specified within the tuple using the nesting level that the range is located in. An example of the complete tuple creating process is given in figure 2.10 taken from [26]. The description of the protocol field within the tuple was simply a '1' if the protocol was specified exactly or '0' if the protocol field filter contained a ternary character '*' [26].

(Address fields are 4-bits and port ranges cover 4-bit portnumbers.)

Filter	SA	DA	SP	DP	Prot	Tuple
a	0*	001*	2 : 2	0 : 15	TCP	[1, 3, 2, 0, 1]
b	01*	0*	0 : 15	0 : 4	UDP	[2, 1, 0, 1, 1]
c	0110	0011	0 : 4	5 : 15	TCP	[4, 4, 1, 1, 1]
d	1100	*	5 : 15	2 : 2	UDP	[4, 0, 1, 2, 1]
e	1*	110*	2 : 2	0 : 15	UDP	[1, 3, 2, 0, 1]
f	10*	1*	0 : 15	0 : 4	TCP	[2, 1, 0, 1, 1]
g	1001	1100	0 : 4	5 : 15	UDP	[4, 4, 1, 1, 1]
h	0011	*	5 : 15	2 : 2	TCP	[4, 0, 1, 2, 1]
i	0*	110*	2 : 2	0 : 15	UDP	[1, 3, 2, 0, 1]
j	10*	0*	2 : 2	2 : 2	TCP	[2, 1, 2, 2, 1]
k	0110	1100	0 : 15	0 : 15	ICMP	[4, 4, 0, 0, 1]
l	1110	*	2 : 2	0 : 15	*	[4, 0, 2, 0, 0]

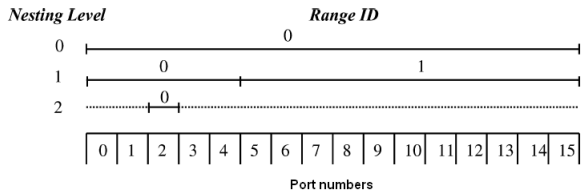

Figure 2.10: A table of tuples derived from a filter set and a diagram of the associated encoding of port ranges.[26]

A variation of the tuple space searching method, called the Pruned Tuple Space Search described in [26, 23] creates trees of the source and destination prefixes with nodes containing tuples that may match a packet that maps to that node in the tree. By mapping the source and destination address of a candidate packet to these trees two lists were obtained of tuples and the intersection of these lists provided the tuples that needed to be searched for filters that may match the candidate packet.

The BV-TCAM Architecture (A hybrid technique) The authors in [21] decided their system would be required to report all rules which were matched by a packet instead

of just the rule with the greatest priority that was matched by the packet. The authors also used a hybrid algorithm in their approach to solving the packet classification problem and their approach made use of Ternary TCAM and a grid-of-tries. Bit vector outputs from these two schemes were then combined thus they labelled their architecture the BV-TCAM architecture [21].

TCAM technology may be fast but it is however expensive in terms of the space required hence it is very beneficial to limit the number of entries required in TCAM memory to be able to perform effectively by careful structuring of the entries [21]. Song and Lockwood achieve this by observing that many rules in a rule set typically share common IP addresses and protocols but differ when specifying ports or port ranges. This led the authors to use TCAM technology to check matches in IP address and protocol fields. Port specifications were checked using a bit vector algorithm as TCAM is not very well suited to checking the arbitrary ranges that one finds for port fields in filters as mentioned earlier in the discussion focussing on TCAM technology.

The output of the IP addresses and protocol matching TCAM mechanism was encoded in a separate bit vector which had a bit length equal to the number of rules being checked. If a packet matched a certain rule then the corresponding bit in the bit vector was set to 1 otherwise it remained set to 0. To match port numbers the authors created a trie structure, The port numbers were mapped down the trie with branch selection governed by the presence of a 1 or 0 at the currently inspected bit of the port number binary value, this is the origin of the bit vector description in the algorithm. Branches in the trie structure extend until either they represent the exact port address to be matched or, in the case of port ranges, the prefix of the port number represented up to that point is sufficient to describe the range of port numbers. Each leaf node of the trie then contained a bit vector indicating which rules had been matched in an identical method to that reported from the TCAM segment of the classification unit. The combination of these two segments, the TCAM and the BV method, then produced a final result indicating which rules a given packet had matched.

The BV-TCAM architecture developed by Song and Lockwood achieved a throughput capable of operating at a rate of 2.488 Gbps [21]. It is therefore also a good demonstration of how separate technologies and algorithms can be combined to form a hybridized system.

2.3.2 Deep Packet Inspection

As mentioned earlier, deep packet inspection forms a key part of packet processing for the aim of defence of a network [14]. This often requires scanning the payload of packets, and the packet headers in some instances, to locate malicious data embedded within the packet by attempting to locate specific search strings within packets. [5, 6, 3, 14, 22] demonstrate different algorithms for such tasks which will be discussed below.

Cho *et al* [6], like a few of the other sources that discuss deep packet inspection [5, 22], make reference to Snort IDS rule sets. Briefly, "Snort is an open source network intrusion prevention and detection system" [11] that is able to detect packets on network interfaces and perform various actions based upon different properties of the packets detected [20]. This is done by applying rules contained in a rule file that form a rule set to each packet that Snort detects [20]. Relevant to deep packet inspection is the ability of Snort to locate strings specified in Snort rules within network packets [20].

Simple N Parallel Rule Checks A simple approach described in [5] is to have N comparators to match N strings for N rules. Each rule can then be treated independently and matching for all rules can occur in parallel. An illustration of the system is shown in figure 2.11 [5] where one can see that some initial processing on the packet headers is done first to check which rules for packet content matching need to be checked. The authors of [5] go on to address the problem of determining which rule has been matched once any match has been detected.

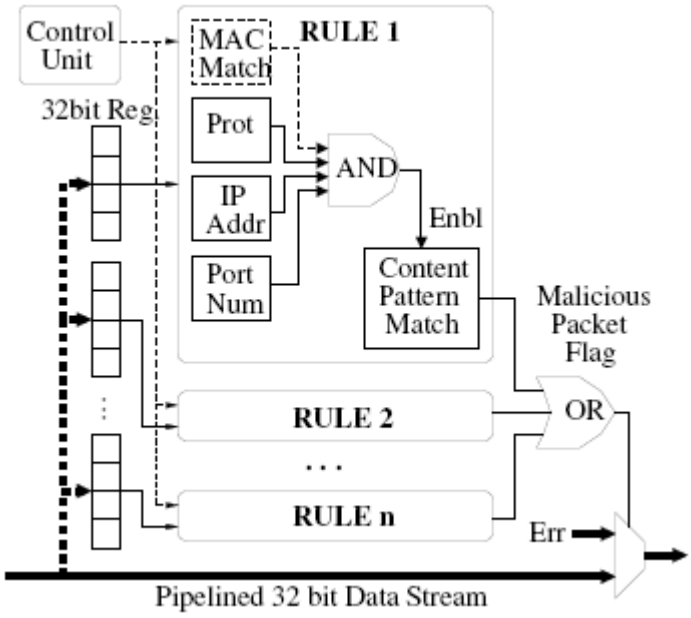

Figure 2.11: The structure of the system described by Cho and Magione-Smith [5]

Figure 2.12 shows how the content matching portion of the design operates by matching a four byte portion of the packet against a sample string with four different offsets.. If one assumes that a packet will only match one rule then the index of the rule matched is encoded in a binary number whose bits are used to navigate a path down a binary tree to the matched rule.

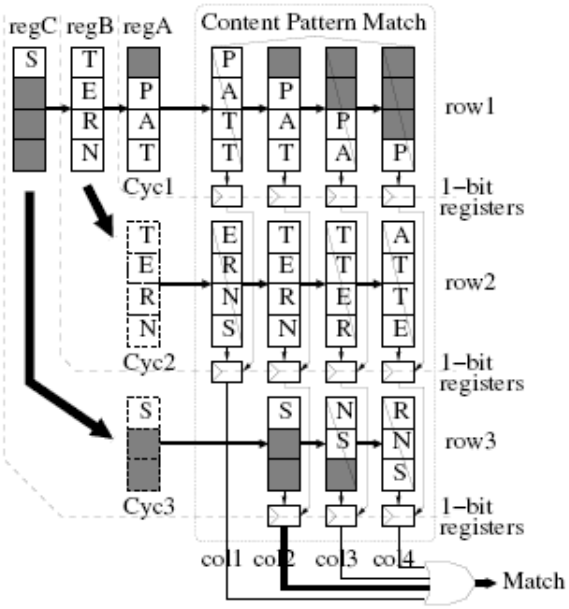

Figure 2.12: The structure of the content pattern matcher developed by Cho and Magione-Smith [5]

If more than one pattern is matched in a packet however then it becomes impossible to determine which patterns have been matched by looking at the index encoder output. Cho and Magione-Smith [5] then propose dividing the rules into multiple sets with no overlapping patterns, so that no more than one rule in a set can be fired by a packet at a time, and assigning to each set an index decoder as described above. This however has obvious increasing space requirements as the number of required index encoders increases with the number of rule sets.

Sourdis and Pnevmatikatos demonstrate in [22] a system similar to that of Cho and Magione-Smith described above but make use of a fan-out structure to essentially copy the packet to many comparator networks that check for the occurrence for patterns in the packet as explained earlier in [5]. Other noticeable differences between the approaches of Cho and Magione-Smith compared to that of Sourdis and Pnevmatikatos are:

- the assumption by Sourdis and Pnevmatikatos that only one pattern will be matched in their system at a time, provided that different pattern suffixes are used [22], enabling the use of a simple index encoder similar to that proposed in [5],
- and careful design of the pipelined comparator so as to conveniently make use of the one 4-input LUT and single flip-flop to identify half bytes of characters and then AND-ing the outputs of the LUT's to form a complete pattern match output.

The technique of matching the pattern to be identified at different one byte offsets within packet used in [5] is also repeated in [22].

Deterministic Finite State Automata in FPGA's An approach to pattern matching within a sample space is to use deterministic finite state automata (DFAs) and these have been successfully implemented on FPGA's as the work of Moscola *et al* demonstrates in [14].

In their work, Moscola *et al* first demonstrate briefly how a pattern composed of characters from a finite alphabet can be described using regular expressions, an example is given below: "Vi(R|r)u(S|s)" would represent the patterns Virus, ViRus, ViruS and ViRuS. Symbols such as '*' can be used to represent multiple occurrences of characters or character sets within a pattern [14].

Moscola *et al* then state the benefits of DFA's over Non-deterministic Finite Automata (NFA's), namely the less space required for a completed DFA than for a completed NFA without consideration for the space required during construction of either automaton [14]. Moscola *et al* parse the pattern for each rule through an application called JLex which constructs a DFA that can identify the regular expression for the desired pattern and the output from JLex is then used to generate VHDL source code which is then used to program the FPGA in the final implementation (VHDL is a hardware design language used to specify the functionality of FPGA's).

The action of the system presented in [14] is described as follows:

1. various protocol wrappers remove the different protocols that may surround the payload of a packet,
2. the packet is then routed through multiple content scanning modules which use the DFA's programmed onto the FPGA to search for the patterns to be identified,
3. once the packet payload has passed through all content scanning modules the appropriate actions are taken with regard to the patterns identified,
4. appropriate actions may include dropping the malicious packet, sending an alert message to a specified network address and outputting the packet from the content scanning module after wrapping the packet in the protocol headers stripped from the packet prior to searching.

Figure 2.13 shows a block diagram of the content scanning module presented by Moscola *et al* as it appears in [14].

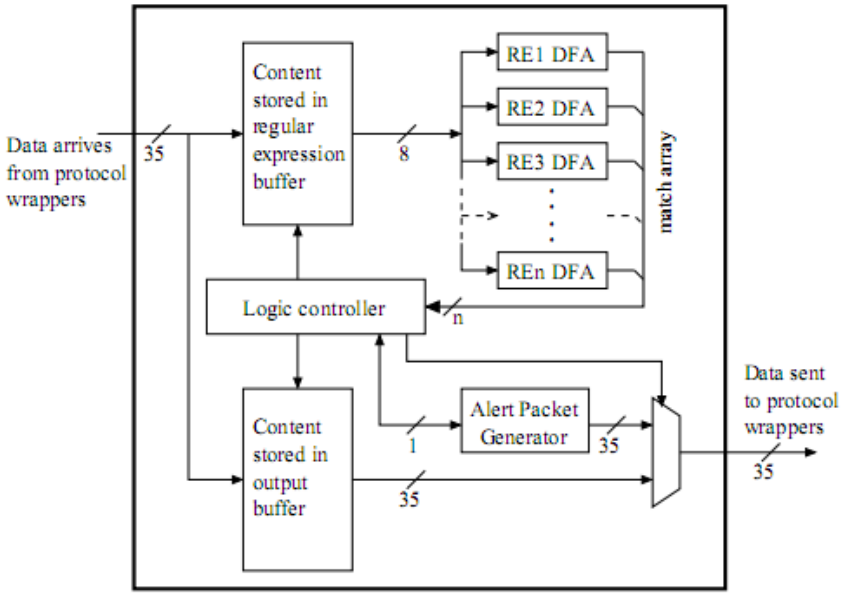

Figure 2.13: The DFA content scanning module presented in [14].

Moscola *et al* state that their system presents data to the content scanning module at 32 bits per clock cycle and note that their system passes data to their regular expression DFA's - (REn DFA) in the diagram above – at a rate of 8 bits per clock cycle. Since this would produce a back log in the system Moscola *et al* designed their system to present packets to one of four content scanning modules selected in a round-robin method thereby achieving a 32 bit per clock cycle processing rate to match the 32 bit per cycle incoming and outgoing rates of the system.

An alternate approach describe by Baker and Prasanna in [2] which 'uses a modified version of the KMP (Knuth, Morris, Pratt) algorithm' [2]. Briefly, the KMP algorithm is able to detect possible occurrences of a pattern within a string and upon a mismatch does not start the next search from very next offset in the string but rather from the next possible start of the pattern in the string [19]. Suppose the pattern is 123 and the string is 121231 the first search for the patter will start at offset 0 but when a mismatch occurs at the third character then the algorithm starts the next possible match at the third character rather than the next offset (which would be at the second character) because the next position in the string at which the patter may be matched is the third character. The KMP algorithm has worst case behaviour O(k + n) [2]. Baker and Prasanna state

that even though other algorithms may have better average case performances than the KMP algorithm no other algorithm has a more efficient worst case performance. Baker and Prassana then describe this property of the KMP algorithm as important because an attacker may try to flood an intrusion detection system in an attempt to force through some malicious packets past the IDS; obviously the best way for an attacker to do this would be to construct packages of the worst case for the IDS. The result of using the KMP algorithm is that this worst case then has O(k + n) behaviour [2].

The contribution of Baker and Prasanna is their demonstration that by adding a second comparator and an input buffer (the minimum size of which Baker and Prasanna derive) the KMP algorithm can be used and the resulting system can always accept at least one input character per cycle of operation [2]. This is an obvious improvement over the traditional approach of using a single comparator which results in the system being able to accept at most one input character per cycle of operation [2], a condition that can be seen to lead to the type of attack mentioned earlier where worst-case packets are injected into the input stream in an attempt to flood the IDS.

2.3.3 Related Work

The authors of [8] describe the implementation of a packet generator built on an FPGA. The NetFPGA platform is an open source development platform for creating and testing of network applications [15]. The packet generator described in [8] is able to achieve a transmission rate of 1000Mbps which is faster than the comparative test that the authors performed using the tcpreplay software. Noticeable results in [8] are the speed at which the NetFPGA platform packet generator was able to transmit packets and the small (<1us) variation in packet arrival time when sent using the NetFPGA packet generator indicating that the packet generator is suitable for time sensitive networking applications [8].

Another packet inspection and processing system implemented on an FPGA is described in [28]. The system created uses an FPGA as an accelerator to an IPS which analysis each packet passing through the FPGA and determines a course of action for each packet; forward the packet, drop the packet or pass the packet on to the IPS for inspection [28]. Similar to the system described in this paper the system described in [28] classifies packets based upon packet headers [28].

2.4 Packet structure

When an application wants to send data across a network it is passed down through each protocol layer which then attaches headers to the data before passing the encapsulated data to the next layer down [25]. The header structure of a protocol is a feature of the protocol which is rigidly defined and well known; as mentioned before the fact that these structures are well known is leveraged to inspect the fields in the headers of the protocols. The header structures of the common protocols Enthernet, IP, UDP and TCP are shown in figures 2.14, 2.15, 2.16 and 2.17 respectively. The Ethernet frame is what is transmitted on the network medium and all other headers and the packet payload are encapsulated within the data portion of the Ethernet frame. IP packets are encapsulated within Ethernet frames when transmitted over an Ethernet network. UDP and TCP are two widely used protocols for transporting data. UDP and TCP datagrams are encapsulated within IP Packets when transmitted over networks that use the IP protocol such as the Internet. TCP offers a reliable, stateful and flow-controlled alternative to UDP.

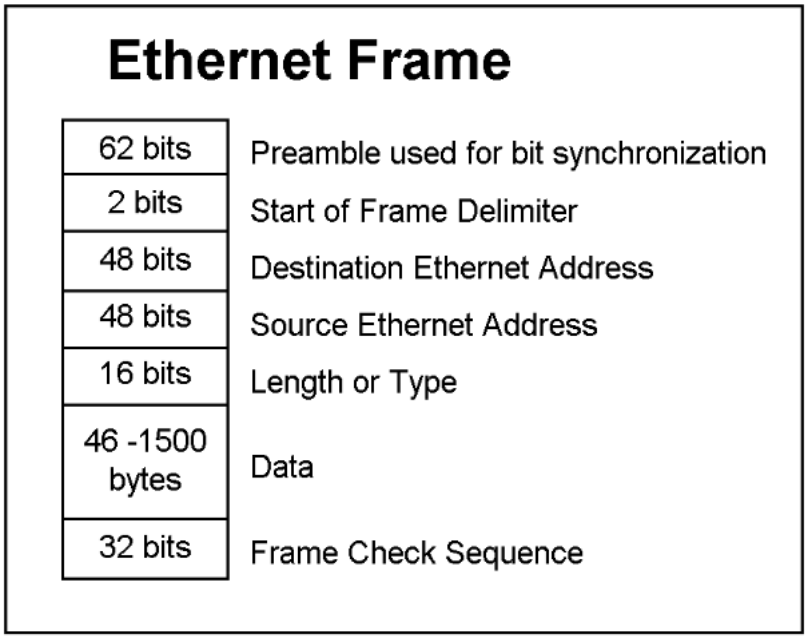

Figure 2.14: The structure of an Ethernet frame.

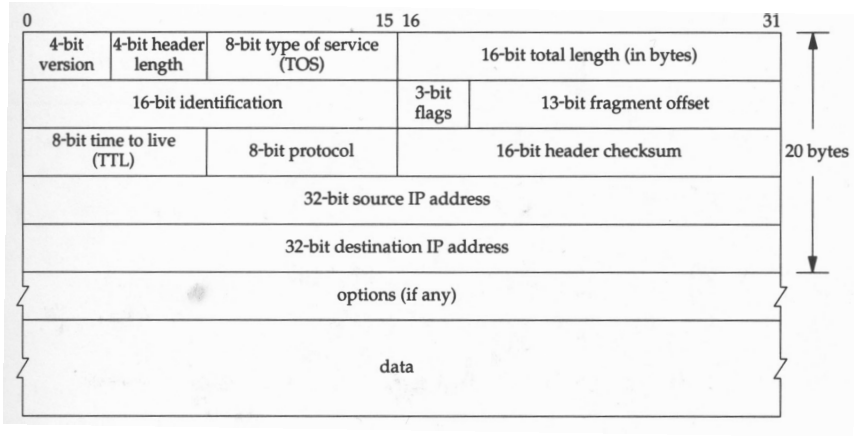

Figure 2.15: The structure of an IP packet.

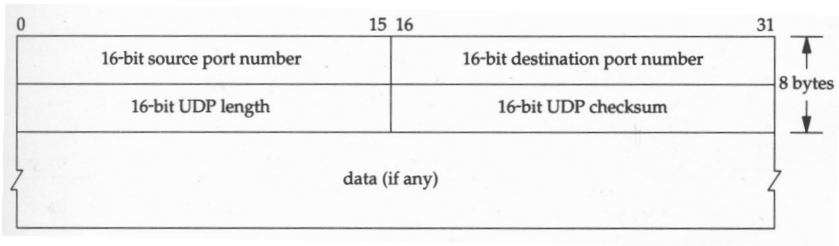

Figure 2.16: The structure of a UDP datagram.

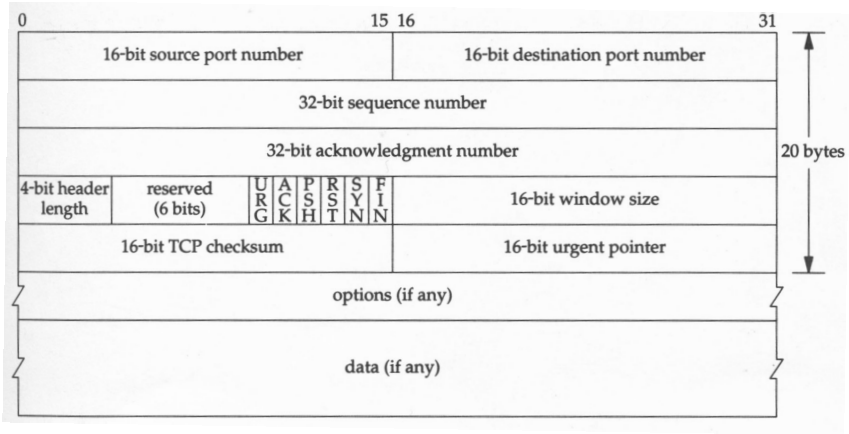

Figure 2.17: The structure of TCP datagrams.

Chapter 3

Design and Implementation

The packet filter described in this work was to implemented on a Spartan-3AN development kit. This development board was chosen because it had a variety of interfaces including an Ethernet PHY chip on the board and standard RJ45 Ethernet connector, obviously vital components for a project the required receiving data from an Ethernet connection. The board also came with 512MB of RAM memory which could be incorportated into the design and the multiple interfaces ensured that the board's application could remain versatile.

3.1 Design Overview

The overall design of the packet filter is modularised to allow easy expansion of the design to include more components for expansion of functionality. Modularisation of the design also allowed the developer to leverage the modular structure of packets with each module performing a specific task. The modules in the design are as follows:

• An Ethernet module that reads in nibbles of data from an Ethernet connection and outputs IP packets.

• An IPrx module that reads in bytes of IP packets and records the protocol and IP address fields.

• A TCPrx module that teads bytes of a TCP datagram and records the port number fields.

• A Trie module that matches protocol fields and IP addresses to rules stored in the design.

• A PortBitVec module that matches port numbers to rules stored in the design.

• An Aggregator module that accepts the outputs from the IPrx and TCPrx modules.

• A Count module that records the number of times that packets match rules stored.

• A Report module that reads the counts of rules and reports them over a serial RS-232 interface.

A diagram of the design is shown in figure 3.1 depicting the relative logical positions of the modules and the signals between them (single bit signals are shown with thin lines and multi-bit buses are depicted with thick lines). The filter described in this paper has provisions to keep a count of ten rules. It was decided that ten rules was an adequate number of rules to demonstrate the functionality of the filter design.

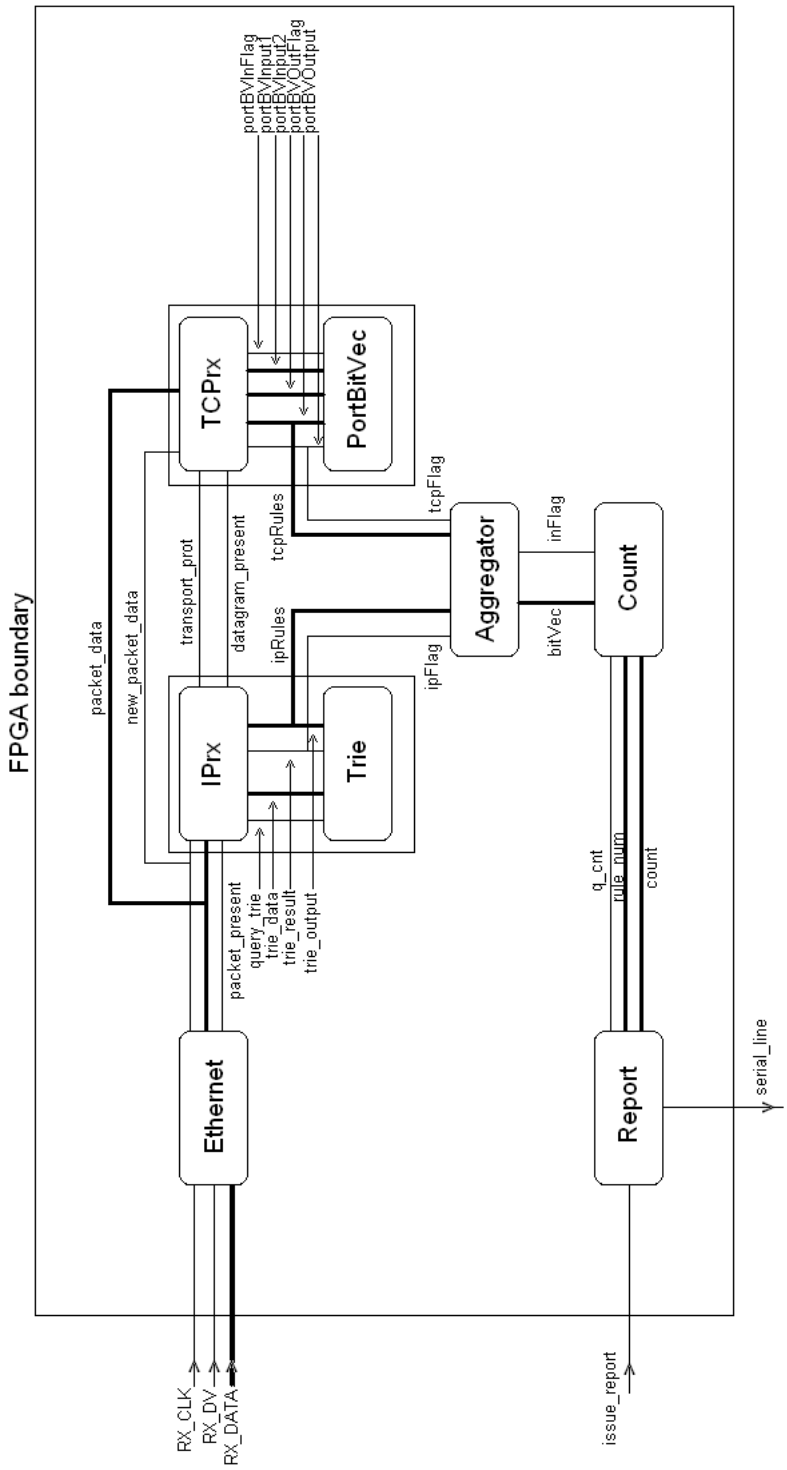

Figure 3.1: An overview of the filter design showing the modules and the signals between modules.

3.2 Module Descriptions

This sub-section provides a description of how each module in the design operates and how the modules interface with each other. It should be noted that finite state machines (FSMs) were used extensively within the design of certain modules and will be described in detail in the relevant modules.

3.2.1 Ethernet module

As indicated at the start of this chapter the development board has a 10/100 Ethernet physical layer interface (PHY) connected to a standard RL-45 Ethernet Connector [32]. These allow one to connect the FPGA device to an Ethernet network. The documentation for the development board lists the following signals available between the FPGA device and the Ethernet PHY:

E_TXD Transmit data bus to PHY (5 bits, MSB is the Media Independent Interface (MII) error bit)

E_TX_EN Transmit enable signal to PHY

E_TX_CLK Transmit clock to PHY (25MHz for 100Mbs operation

E_RXD Data received from PHY (5 bits, MSB is the MII error bit)

E_RX_DV Receive Data Valid signal from PHY

E_RX_CLK Data receive clock from PHY (25MHz for 100Mbs operation)

E_CRS Carrier sense signal

E_COL MII collision detect signal

E_MDS Management clock

E_MDIO Management data input/output

E_NRST PHY reset (active low)

For the purposes of implementing a packet filter the E_RXD, E_RX_CLK, and E_RXDV signals (termed RD_DATA, RX_CLK and RX_DV signals respectively in this report) were used to read data from the PHY. Since the application of this specific packet filter was to filter packets captured previously it was deemed unnecessary to detect errors in packet transmission as such packets would have been dropped when captured and need not require filtering. Furthermore since the packet filter was to operate as a purely passive listening device no operation was required to transmit packets to the PHY and then onto an Ethernet network, all transmission based signals and PHY management signals were also ignored.

The task of the Ethernet module was to read data coming from the PHY a nybble at a time, strip the Ethernet fields from the Ethernet frame, determine if the frame contained an IP packet and notify the IPrx module if an IP packet was present in the frame while passing full bytes of data to the IPrx module. Since data received from the PHY was delivered 'low-nybble first' the Ethernet module also had the task of ensuring that nybbles within a byte of data were ordered correctly. The required behaviour of the Ethernet module is described (and was implemented) in the FSM shown in figure 3.2 .

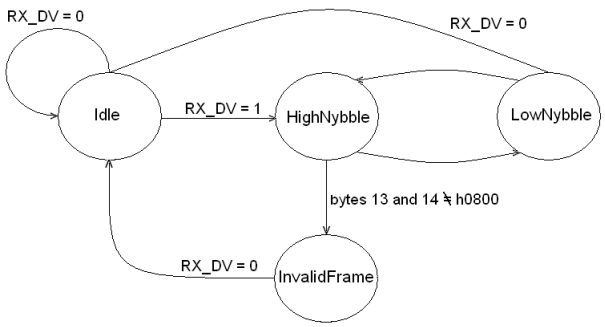

Figure 3.2: The FSM that captures the behaviour of the Ethernet module.

The Ethernet module remains in the Idle state until the RX_DV flag is asserted. When in the Idle state the byte counter is reset to zero. When the RX_DV flag is set the value of RX_DATA is copied to a four bit register (low_nybble) on the rising edge of the RX_CLK. The FSM then transitions to the HighNybble state on the falling edge of the RX_CLK.

When in the HighNybble state if less than 13 bytes of data have been received the value of RX_DATA is ignored as it is merely a portion of the Ethernet MAC addresses and of no interest else if 13 or 14 bytes of data have been received then the upper and lower nybbles are copied to a 16 bit register (frame_type) to store the value of the network layer

protocol field. After the 15th byte has been received the frame_type register checked to see if it matches the network protocol value for IP (0x0800) and the byte counter stops incrementing on each byte. If the network protocol is determined to be that of the IP protocol (0x0800) then a flag (new_packet_data) is raised to indicate to other modules that a full byte has been received and another flag (packet_present) is raised to alert the IPrx module that an IP packet has been detected and is being received. On the falling edge of the RX_CLK the FSM transitions to the LowNybble state.

When in the LowNybble state the new_packet_data flag is de-asserted and the value of RX_DATA is copied to the four bit register low_nybble. If the RX_DV flag is de-asserted then the FSM transitions back to the Idle state otherwise the FSM then transitions to the HighNybble state. The FSM constantly oscillates between the HighNybble and LowNybble states in this fashion until the RX_DV flag is de-asserted after the last full byte of a frame has been received.

The InvalidFrame state is a special state that occurs if the frame_type register does not hold the value after the 15th byte has been received. This check is performed as mentioned in the HighNybble state. When in the InvalidFrame state new_packet_data, packet_data and packet_present are all de-asserted. The FSM remains in this state until the packet has been fully received and the RX_DV flag is de-asserted whereupon the FSM transitions back to the Idle state. A screen shot depicting the signals and state transitions within the Ethernet module during normal operation and for an invalid packet is shown in figure 3.3.

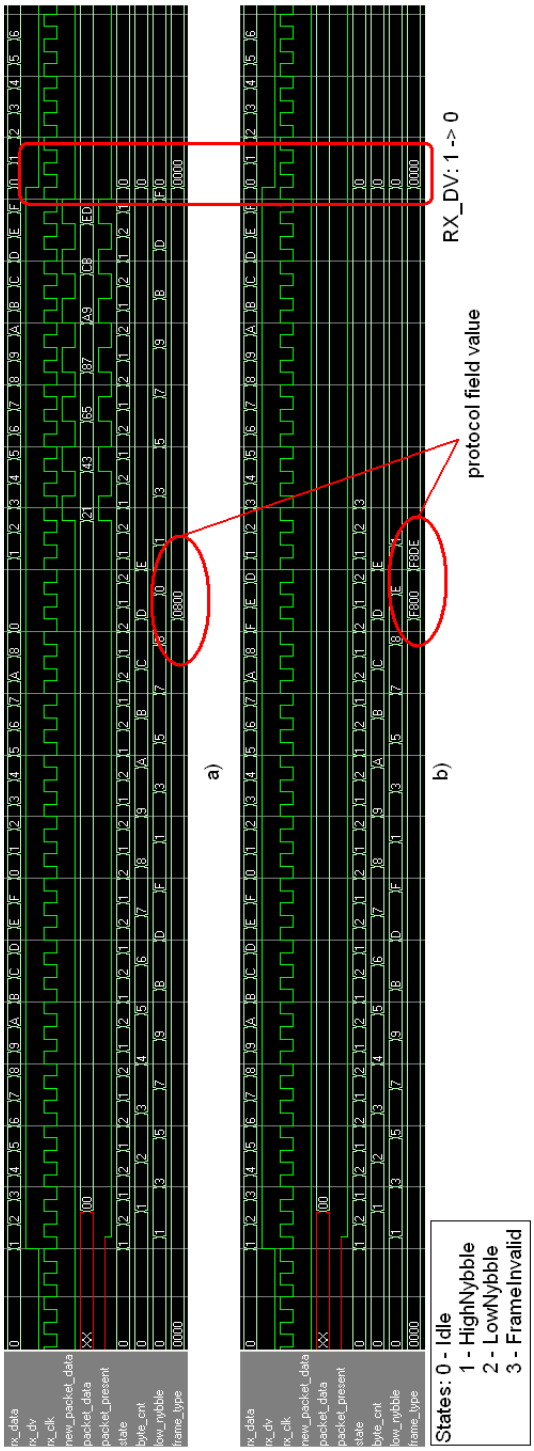

Figure 3.3: Traces showing the operation of the Ethernet module a) when an IP packet is encapsulated within and Ethernet frame and b) when the Ethernet frame does not contain an IP packet.

Figure 3.3 is a screenshot from ModelSim output depicting the operation of the Ethernet module. In trace (a) the protocol field value is set to 0x0800 and the reader can observe that once the network layer protocol has been identified as the IP protocol the packet_present flag is raised and the new_packet_data is raised each time a new byte is received.

In trace (b) the protocol field value is set one other than 0x0800 and the packet_present and new_packet_data flags are never raised.. In both diagrams is can be seen that even though nybbles may still be received on the RX_DATA bus once the RX_DV flag is de-asserted the packet_present and new_packet_data flags are also de-asserted. Note that nybbles from bytes are received lower nybble first and note too the different behaviour when the protocol field value (frame_type) is changed from 0x0800 (in figure 3.3 (a)) to another value (as in figure 3.3 (b)). As can be seen in the diagram the nybbles received are always paired to form bytes but that the new_packet_data and packet_present signals are only de-asserted if the frame_type register is 0x0800.

3.2.2 IPrx module

The IPrx module is activated by the packet_present signal that is driven by the Ethernet module. The packet_present signal is asserted when an IP packet is detected by the Ethernet module and all Ethernet fields have passed to the Ethernet module. When the packet_present signal is asserted the new_packet_data signal, also driven by the Ethernet module, is used as a clock for the IPrx module in much the same way as the RX_CLK signal is a clock to the Ethernet module. The functioning of the IPrx module is captured in a pseudo-state machine with the pseudo-state being the value of a byte counter that is used to count the number of bytes received by the IPrx module.

Then byte counter is incremented on the falling edge of the new_packet_data signal and increments until the 20th byte has been received. When the 10th byte is received by the IPrx module it is stored in a register (transportprot) as this is the protocol field of the IP packet and is used in the rule matching process. The protocol field value was also intended to be used to activate separate and distinct modules to match port numbers based upon the transport protocol; an example of this use is shown in the TCPrx module. When bytes 13 – 16 are received they are stored too as they are the source IP address of the IP packet and bytes 17 – 20 are also stored as they form the destination IP address of the packet.

The protocol field and IP addresses are stored on a 72 bit bus which is connected to the Trie module. Once the 16th byte has been received two events occur; first a flag is raised to the Trie module to begin checking if the protocol field, and IP addresses match any rules and secondly a datagram_present flag is raised to the TCPrx module to notify the TCPrx module that datagram bytes are currently being received on the packet_data bus from the Ethernet module.

3.2.3 Trie module

The Trie module is a simple module within the IPrx module with two inputs: a query_trie signal that activates the module and a 72 bit bus (triedata) structured as (71:64 – protocol field, 63:32 – source IP address, 31:0 – destination IP address). When activated the Trie module compares the values on the bus to pre-generated bit masks for each rule and if all the values on the bus match all the bit masks for a given rule the rule is said to match the packets. What is not shown in the overview diagram is a clocking signal passing into the IPrx module and passed to the Trie module. The Trie module begins checking rule bit masks on the rising edge of this clock and all bit masks are checked in parallel.

A flag (trie_result) is raised on the next falling edge of the clock to alert the IPrx module that a result for the bit masks has been determined and the results of the bit mask matches are placed on a n bit bus (trie_output, back to the IPrx module) where n is the number of rules to be checked by the Trie module. If rule x is matched in the Trie module then bit x in trie_output is asserted otherwise it is de-asserted. The clock to the Trie module can be of any frequency provided that its period is double the longest time required for a bit mask to be matched. A screenshot of a simulation of the function of the IPrx and Trie modules is given in figure 3.4.

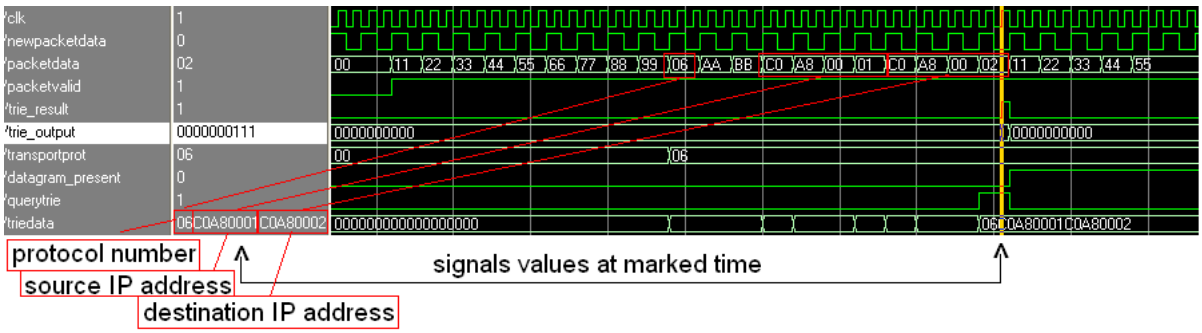

Figure 3.4: A trace output showing the behaviour of the IPrx and Trie modules.

In figure 3.4 one can see the incoming bytes to the IPrx module. As stated already these bytes are received from the Ethernet module on the rising edge of the new_packet_data signal. If one looks at the trie_data signal one can see how the protocol field and IP addresses are copied to the trie_data signal. The result of the Trie module's matching of the fields to rules is asserted at the time marked by the yellow bar but is obscured in the diagram as the result is de-asserted soon after and not enough space is present to display the bit vector; the result bit vector has therefore been highlighted on the left of the diagram and is valid from the yellow bar until the next transition of the trie_output signal which is the time that the trie_result signal is asserted for.

The trie_output bit vector has the least three significant bits asserted indicating that the first, second and third rules stored in the Trie module matched this particular packet; that the other bits are all de-asserted indicates that no other rules matched the packet. When the Trie module has matched the trie_data signal to rules the trie_result flag is raised. When the next byte is received on the packet_data signal the datagram_present signal is asserted as the next byte is part of the transport layer datagram (assuming no IP options are used).

The Trie module was not hardcoded but rather generated using a small Java application written by the author. This application reads in rules from a user specified rule file and parses it for values of fields pertaining to the IP packet. These values are then inserted within the Trie module to enable the Trie module to detect matches to the specified rules. This auto-generation of the Trie design file was chosen to allow one to specify customized rules without requiring knowledge of VHDL.

3.2.4 TCPrx module

The TCPrx module and the PortBitVec module within the TCPrx module are analogous to the IPrx and Trie modules respectively and are enabled by the IPrx module in a manner similar to how the IPrx module is enabled by the Ethernet module. The TCPrx module receives bytes of data from the packet_data signal and uses the new_packet_data signal as input clock. The datagram_present signal is used as an enable signal to the TCPrx module.

While the TCPrx module can still read the new_packet_data and packet_data signals no action will be taken unless two criteria are met: first the datagram_present signal must be asserted by the IPrx module which occurs after the IPrx module has parsed all

the IP fields and the second criteria is that the transportprot signal – driven by the IPrx module – is that of the TCP protocol (that is h6).

The design of the TCPrx module also makes use of a byte counter, as the IPrx module does, to locate specific fields in the transport datagram and the TCPrx module was designed in a manner that with the alteration of the target byte counter values different bytes within the datagram would be read. For the TCPrx module the module stores the first four bytes of the datagram as these are respectively the 16 bit source port and 16 bit destination port numbers and are the last two fields of a TCP/IP packet needed to match rules based on the basic 5-tuple.

That all transport level protocols contain the port numbers is the basis for the incorporation of the transportprot signal as an additional criterion to activate the TCPrx module; the TCPrx module will only be activated when the IP fields have been parsed (the datagram_present signal is asserted) and the datagram is a TCP datagram (when the transportprot signal carries the value h6).With the alteration of the byte counter target values the TCPrx module could capture the port numbers for another protocol located at different locations within the datagram; this idea is expanded on later in this paper (see section 5.1.1).

3.2.5 PortBitVec module

The PortBitVec module is placed within the TCPrx module, similar to how the Trie module is placed within the IPrx module. Once the TCPrx module has identified the source and destination port numbers a flag (portBVInFlag) is raised to the PortBitVec module and the source and destination port numbers are passed to the PortBitVec module (on busses portBVInput1 and portBVInput2 respectively). The port numbers are checked against values specified in the rules and matches are reported in a bit vector in an identical fashion to that returned from the Trie module. A screenshot of a simulation of this process is shown in figure 3.5.

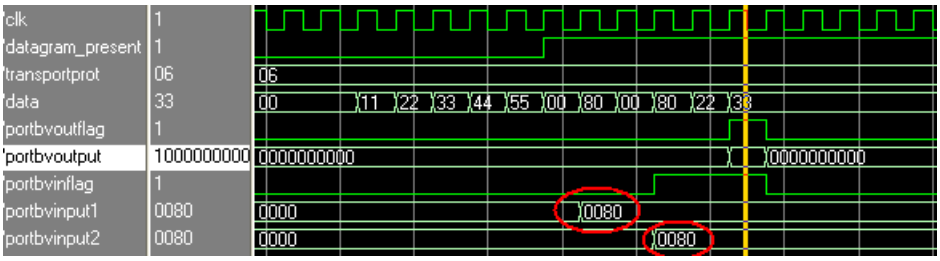

Figure 3.5: A trace output showing the behaviour of the TCPrx and PortBitVec modules

The process whereby port numbers are matched is not as simple as the use of bit masks to match IP addresses. As seen in the literature the complication arises because while IP address ranges can typically be described by bit masks port number ranges are typically within arbitrary numbers that cannot be defined by a single bit mask as with an IP address. The approach taken in this paper to identify the rule pertaining to port numbers provided very high speed matching but at the expense of resources on the FPGA..

The port numbers specified in the rule set were converted into binary numbers and each bit was combined with another using AND and OR gates to evaluate to TRUE or FLASE if the sample number is either greater or less than the port number in the rule set. As an example consider the number 6 in a domain of 4 bit numbers. Expressing 6 as a 4 bit binary number one gets 0110. To detect if a test number, t, is greater than 6 one can evaluate the following expression (where t(3) is the most significant bit of t and t(0) is the least significant bit of t): t(3) or (not t(3) and t(2) and t(1) and t(0)) where t(x) is considered TRUE if t(x) is 1 otherwise FALSE if t(x) is 0. This method is scaled up to accept 16 bit numbers.

The estimated delay from using this method is the sum of the switching time for each logic gate used in the evaluation of the test number and the complete propagation delay between each gate. However in practice this method as implemented in this packet filter is encoded in the outputs of 4 input LUTs.

Like the Trie module the PortBitVec module is also a generated file rather than a hard-coded file. The Java application that generates the PortBitVec module also reads rules from a user specified rule file and parse the rules to locate the values of the port numbers and the relational operators that applies to the port numbers (equality, less than, greater than or port range). The java application then inserts the correct expressions within the PortBitVec design file. Again, the benefit of this approach is that someone wishing to specify a customized rule set need not have knowledge of VHDL programming.

3.2.6 Aggregator module

Since the IPrx and TCPrx modules both report the matches of different components of the rules in the rule sets at different times and the results of these matches are not implied to both be valid at the same time an Aggregator module is implemented to aggregate the results of the IPrx and the TCPrx modules before passing the final rule matching results to the Count module.

The Aggregator module has a single bit input flag from each of the IPrx and TCPrx modules (ipFlag and tcpFlag respectively) and an n bit bus (where n is the number of rules in the rule set) from each of the same modules (ipRules and tcpRules respectively) that carries the results from the two modules. The Aggregator makes use of a FSM to control its operation and collating of the rules. This FSM is shown in figure 3.6.

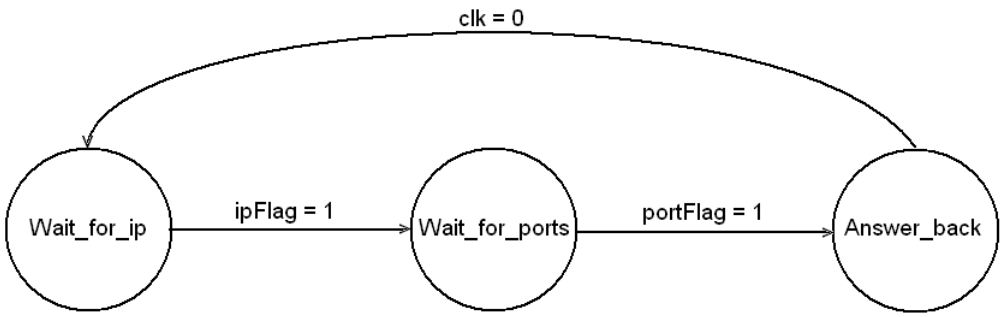

Figure 3.6: The FSM describing the behaviour of the Aggregator component

The Aggregator module begins in the Wait_for_ip state. When the ipFlag is raised the Aggregator module stores the value carried on the ipRules bus in an n bit register (ipRules_reg) and the Aggregator moves to the Wait_for_ports state. When the portFlag is raised the contents of the ipRules_reg is AND'ed bit for bit with the value on the portRules bus. The result is placed on the Aggregator output bus (cntRules) and a flag is raised by the Aggregator (cntFlag) to alert other modules that a valid set of match results is present. The Aggregator then moves to the Answer_back state. On the next fall of the clock signal into the Aggregator module the Aggregator module moves back to the Wait_for_ip state in preparation for the next IPrx result. This process is shown in figure 3.7.

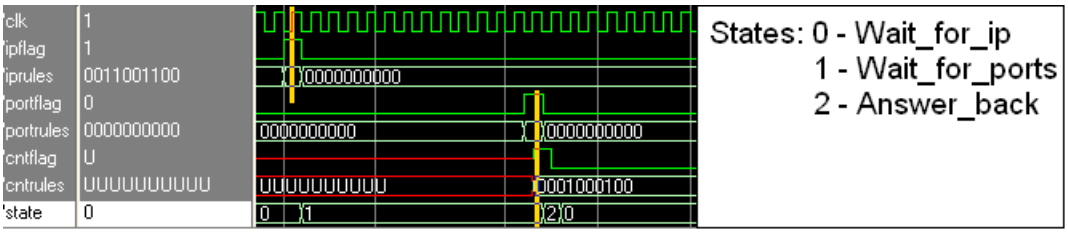

Figure 3.7: A trace output showing the bahviour of the Aggregator component's output signals relative to the module's input signals.

3.2.7 Count module

The Count module reads the output from the Aggregator module and increments counters for all rules matched to a particular packet. When the cntFlag from the Aggregator module is asserted the Counter module cycles through each bit on the cntRules bus and if the bit read has the value 1 then a register that stores the number of times a rule has been match is incremented by one. The Count module reads through each bit on the cntRules bus instead of just until the first bit read is 1 to ensure that all registers are incremented in the case of multiple rule matches. This process is shown in figure 3.8.

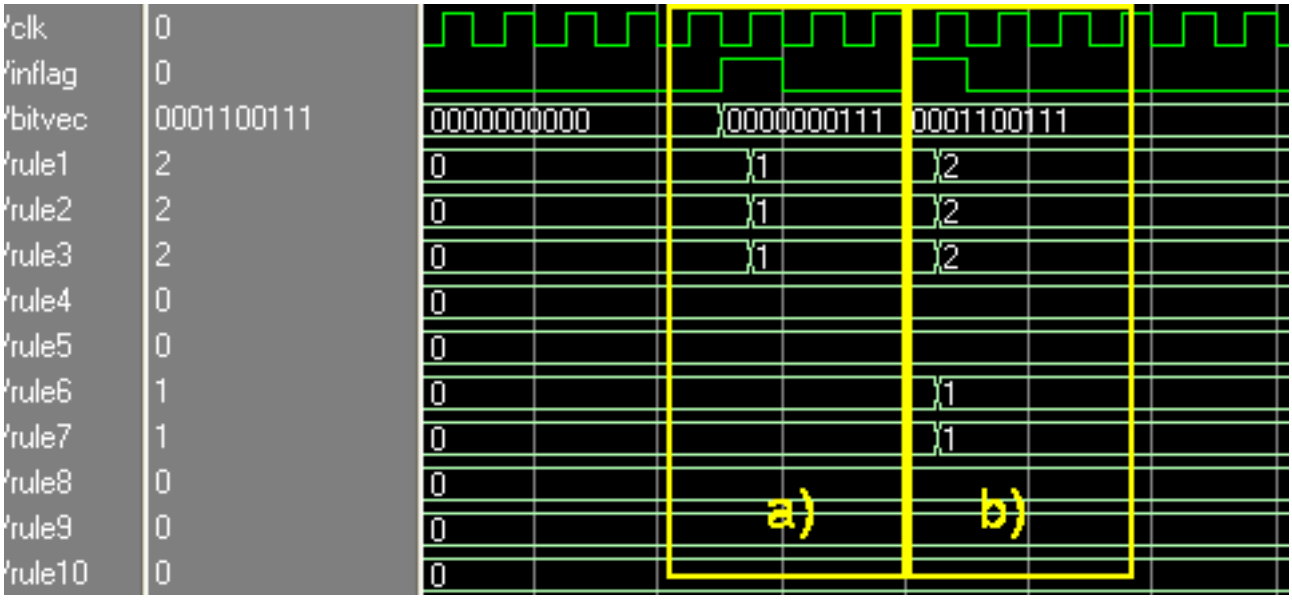

Figure 3.8: A trace output showing the behaviour of the Count module when updating the rule counters after a packet has been through the matching process.

In the area marked (a) the packet is seen to match the first three rules by the assertion of the three least significant bits of the bitvec signal. The counters for the first three rules are therefore incremented by 1 each. In the area marked (b) a similar analysis indicates that rules 1, 2, 3, 6, and 7 were matched by the packet hence the relevant counters are incremented.

The Count module also reads a q_cnt flag from the Report module; when the q_cnt flag is asserted the Count module also reads a bus (rule_num) which specifies the particular rule for which the associated count should be reported back. The Count module then places the value of the relevant register on a bus (count) to be read by the Report module.

3.2.7.1 Report module

The Report module is responsible for providing the values of the rule counters over a serial interface on request. This is a process and is captured in the FSM shown in figure 3.9. The Report module remains in the Init state and rule_num is set to h0 until the issue_report signal is asserted. The issue_report signal is controlled by a slide switch on the development board and therefore requires user interaction to be driven high or low.

When the issue_report signal is asserted the query signal to the Count module is asserted, the value of rule_num is h0 and the Report module transitions to the ReadRule state. The Report module waits in the ReadRule state for one clock cycle to allow the rule count for the first rule to appear on the rule_cnt bus and then the Report module transitions to the TransmitFirstByte state, a flag is raised to the SerialComs component and the most significant byte is placed on the transmit_byte bus to the SerialComs module. The Report module then transitions to the FirstByteFinished state to await the completion of the transmission of the most significant byte via the serial interface.

The completion of transmission is indicated when the SerialComs module asserts the transmit_complete signal. The Report module then transitions to the TransmitSecondByte state and the start_transmit flag is raised again. The Report module then transitions to the SecondByteFinished state to await the completion of the second most significant byte of the rule count. A similar process is followed for the third and forth bytes of the count bus and the Report module then transitions to the ReadRule state again to read the count of the next rule after the rule just transmitted. When this rule's counts are received from the Count module the Report module transmits it over the serial interface again as before. This process continues until all the rules have been read from the Count module and transmitted over the serial interface.

Since this module is controlled through user interaction no signal is asserted upon completion of this module's task. When all of the rule counts have been transmitted via the serial interface the Report module transitions from the ForthByteFinished state to the Finished state until the Report module is asked to issue another report.

With regard to the reporting of rule counts by the filter three options were available: use of the onboard 16 character LCD display, encapsulate the counts in an Ethernet frame and transmit the data over a network or transmit the counts over a serial RS-232 interface to a connected device (most likely a computer). The author chose to make use of the serial interface for the sake of simplicity however the other options will be discussed in slightly more detail in chapter seven.

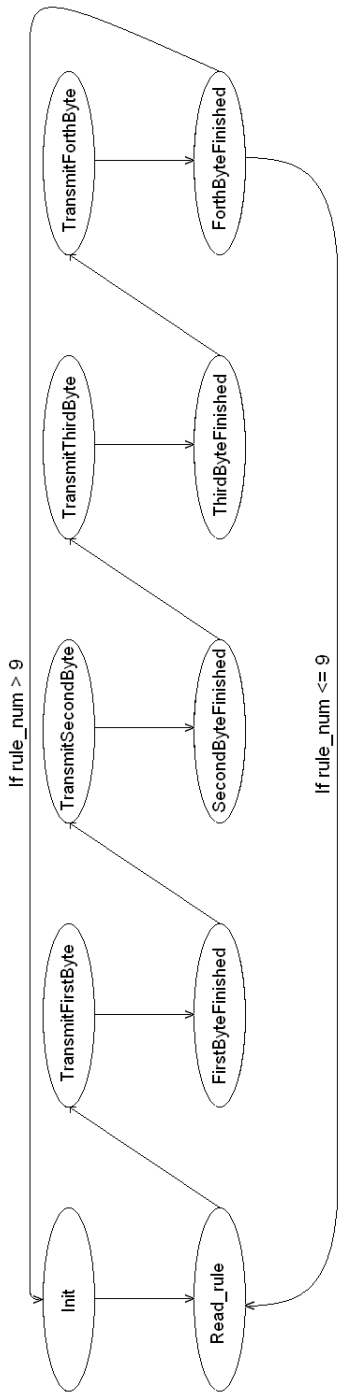

Figure 3.9: The FSM that captures the behaviour of the Report module when asked to transmit the rule counts for the filter rules in the rule set.

Chapter 4

Results

At the time of writing this paper the filter described in this project had not yet been downloaded to the FPGA device although the logic in the design had been verified through the use of ModelSim Starter Edition. This software provided trace outputs similar to the ones shown in chapter 3 which were used in the design process to debug faults in the designs as they were recognised. The lack of results that could give a measure of the performance of a functioning filter as described in this paper is unfortunate; hence this section is therefore a discussion of the result of the implementation process to date and the reasons why the implementation of the filter did not progress beyond design and functional verification.

Unfortunately the verification of the logic of a design in ModelSim does not imply that the design can successfully be downloaded to the FPGA device or that the design will operate as intended. This was realised while implementing the Filter design described above in two primary ways.

Firstly to simulate the functional behaviour of a design the simulation tool must be able to compile the source design files of the design which requires that the usual compiler requirements be met (correct syntax etc.); however, implementing a design on an FPGA device once the design has been described (by using VHDL design files in this project) is a process during which many sub-processes occur.

One of these sub-processes is synthesis of the design. According to the FPGA enthusiast website www.FPGA4Fun.com [16] synthesis of a design creates a netlist of the design which is a description of how basic discrete logic components are connected together. This process implicitly requires that any signals described in the design can be physically

realized using discrete logic circuitry such as logic gates and flip-flops. This is a requirement that need not be met during simulation and in the case of this project required the re-design of large portions of the design to accommodate the synthesis requirements – particularly the description of synchronous components such as the states of finite state machine's used throughout the design. This re-design extended the time required to fully design the filter and was a contributing factor in the inability to fully implement the filter.

The second factor that had an adverse effect on the performance of the design was the effect of physical limitations on signal delays within the device particularly the problem of clock skew. Clock skew is the difference in time between the arrival of a clock signal at one component and the arrival of the same clock signal at another component further along the path of the clock [33]. The clock skew of a clock signal causes a de-synchronisation of signals that should be synchronised and this has obvious implications in the validity of data passed between modules and the filter design cannot hence be expected to operate correctly if clock skew is unintentionally present in the final design.

The reason for the occurrence of clock skew within the final design downloaded to the FPGA device is due to the result of the translate, map and place and route sub-processes within the implementation process. During these processes the netlist created during the synthesise sub-process is used to create a file to be downloaded to the FPGA device that specifies the configuration of the device to enable it to perform as described in the design files. It is during these sub-processes that constraints upon the performance of the design (such as timing specifications of clock signals and the physical pins on the FPGA device to which signals should be connected) are evaluated and the results of the evaluations are reported back to the user.

It was during these sub-processes that the design software reported that excessive skew may be present in some signals. Again this was due to a lack in understanding of the processes by which VHDL design files are used to generate the final configuration bit file; during this process signals used as clocks were mapped to general routing paths within the device and not the dedicated clock routes within the design. The debugging process that ultimately led this conclusion and its solution was rather lengthy and was therefore a second contributing factor to the time required to fully design and implement the module.

While the above two issues describe factors responsible for the filter not being fully realised it is pertinent to also discuss the method of retaining the number of times rules have been matched particularly since a 512MB DDR2 SDRAM chip is available on the Spartan-3AN development board but did not feature in the filter design. The documentation for the

development board [32] suggests that instead of the application developer creating the interface to the RAM chip the user should instead use the Memory Interface Generator (MIG) tool developed by Xilinx to generate an interface to the RAM chip.

The MIG user guide [31] states that the MIG tool outputs Verilog and VHDL design files as well as all other files required to successfully implement an interface to the specified RAM chip with all timing requirements met to ensure proper behaviour of the RAM memory. A top-level module is also created by MIG which is the simplified interface to be implemented by the application developer. It was intended that the filter in this design make use of the DDR2 SDRAM available on the development board however the author of this paper was unable to implement the MIG generated interface.

While the MIG tool is intended to output an interface in both Verilog and VHDL the version of MIG used in this project (MIG v3.0) outputs only Verilog files for the specific DDR2 SDRAM chip used on the Spartan-3AN board (MT47H32M16 developed by Micron Technology). To output VHDL files a more up-to-date version of MIG would be required which in turn requires the use of a new version of the Xilinx ISE Design Suite which would require a new software license. Due to the short amount of time left to implement the filter this option was abandoned.

However it is possible to declare and instantiate Verilog design files within VHDL design files and vice versa. This however led to problems when trying to synthesize the design; the Xilinx ISE seemed to insist on locating the VHDL based design files instead of the Verilog ones and this issue could not be resolved. It was for this reason that the rule count storage method described above (the use of registers) was implemented.

It is for reasons described above that the design resulted in that described in chapter 3.

Despite not being able to fully implement the filter design on the FPGA device the synthesis process does provide a report on various aspects of the FPGA device utilization and one of these aspects is the number of 4 input LUTs used to implement the logic of the design. This feedback from the synthesis process enables one to estimate the cost of a rule in terms of the number of 4 input LUTs required to implement the matching logic. The synthesis process was run twice; first with all rules set their worst case forms (full 32 bit bit masks for both IP addresses and ranges specified for port numbers) and the number of LUTs required for synthesis was recorded. Then the synthesis process was run again but this time the rules were set to their best case forms (0 bit bit masks for the IP addresses and no port number matching performed) and the number of LUTs needed were again recorded. The results are reported in Table 4.1.

To conclude, various aspects prevented the design of the packet filter being implemented on the development board. These were largely due to the inexperience of the author in a new language in a new field. However if the above mention problems could be overcome the design itself does not seem to present any forseeable flaws.

Table 4.1: LUT resource usage for best and worst case rule forms (ten rules)

	Number of LUTs required	Percentage of total LUTs available
Best case	765	6%
Worst case	1108	9%

Chapter 5

Discussion

The final design described in chapter 3 is severely limited in terms of the number of rules that are stored because an interface to the on board DDR2 SDRAM chip could not be implemented and this limited the number of rule counts that could be stored in a practical manner. However, to provide evidence that FPGAs are a viable platform on which to base packet filters an analysis of the rate at which packets can be processed is a more important measure of a packet filter's capabilities than the ability to record the number rule matches as any actions to take place following a rule match can be offloaded to another system; it is the specific task of matching packets to rules that needs to be optimised to avoid the pitfalls of an overwhelmed filter described in the literature. Hence the use of only 10 rules as a rule set for this packet filter need not be a concern.

What may be of some concern to the reader is that one may be inclined to assume that the time taken to match rules increases as the number of rules increases. Observing the design files of the protocol and IP address matching component and the port numbers' matching component one can see that within each of the Trie and PortBitVec modules the fields are matched against the values specified in each of the different rules in parallel i.e. the IP fields are checked against rule 1 at the same time as they are checked against rule 2 and at the same time as against all of the other rules and a similar parallelism exists in the PortBitVec module. Hence an increase in the number of rules should not have an effect on the time taken for the packet to be compared to all the rules except to introduce longer path delays between the IPrx module and its internal Trie module and similarly for the TCPrx module and its internal PortBitVec module.

As is often the case this very appealing performance with regard to time incurs a hefty penalty with regard to the number of FPGA resources required however the results ob-

tained from the study of best case vs worse case LUT requirements are promising. In the worst case rule specifications the rules were crafted specifically to require as many logic components as possible and hence as many LUTs as possible to implement these components but in the best case rule specifications all rule outputs were tied to 1 and hence the number of LUTs required was minimal. As can be observed from table 4-1 roughly 350 LUTs, or around 4% of available LUTs, are required to implement ten rules with worst case forms. Using 6% of available LUTs for other logic in the design leaves 94% of LUTs available for rule specification but as the number of rules increases the number of other support LUTs, besides those directly related to matching rules, can also be expected to increase. So if one were to allow a conservative estimate of 70% of LUTs available to direct rule matching logic with the remaining 30% dedicated to other logic in the application and supporting logic to implement the design a user could conceivably specify 175 rules of the worst case form in the filter rule set before running out of FPGA resources.

Chapter 6

Conclusion

In conclusion, this paper alone demonstrates that while the packet filter described within this paper was never fully realised it is at least possible to describe the logic required to implement a packet filter operating on the basic 5-tple of protocol number, sour and destination IP addresses and source and destination port numbers and this paper, through the use of the VHDL hardware description language and the Xilinx proprietary synthesis software, also demonstrates that the logic for a packet filter can be realised in a circuit of discrete logic components. Beyond this paper there is an extensive array of literature supporting the use of FPGAs in networking tasks and technologies with the performance of the described systems advancing continuously as FPGA and FPGA related technologies continue to improve.

The gaol of this project was to implement a packet filter that was capable of storing counters that kept track of how many times packets matched certain rules and this process was successfuly simulated. The values of the counters then had to be transmitted via a serial interface upon request by the user and this process was successfully implemented on the FPGA device.

The aim of these goals was to demonstrate the vaibility of using FPGAs as packet filters with the successful simulation of the design and the extensive literature supporting this claim it is concluded that FPGAs are indeed a viable platform on which to implement packet filters provided that the necessary experience is present within the application designer and developer.

6.1 Future Extensions

Due to the design described within this paper being very simplistic in its approach there is much scope for extension of the capabilities of the filter and a few of these possibilities are discussed in the following section 6.1

The new programming environment presented by FPGAs when compared to more conventional programming environments requires much time spent learning the new techniques relevant to coding in VHDL for FPGAs, more than the expected time frame of this project. As a result many attractive aspects of this project that formed part of the concept of the final packet filtering system were not realised. These aspects will be discussed in this chapter to provide the reader with an idea of the best performing filter that can be implemented on the development board used in this project and of what other technologies are available for use in FPGA based packet filters in general.

6.1.1 Expanding filter capabilities

Using the 512MB DDR2 SDRAM As stated the development kit has 512MB DDR2 SDRAM chip and the implementation of an interface to this memory has been successfully designed and verified by Xilinx (sp3anddr2}. The incorporation of this memory into the filter design would allow the size of rule sets to be greatly expanded and their associated rule counters to be stored too providing the user with the option of specifying many more rules for the filter. Since the interface is capable of operating at DDR2-400 specifications it is envisaged that the impact on maximum filter speed will be acceptable to still allow the filter to operate at line speeds of 100Mbps or less. However the absolute maximum line speed at which the FPGA will be able to operate effectively can be expected to decrease as the period of RX_CLK begins to approximate that of the DDR2-400 instruction cycle. Ultimately the balance between speed and size of the filter will have to be adjusted to suit one's needs but the implementation of the DDR2-400 interface would most certainly be a great benefit when operating at the speeds the development kit is capable of.

Adapting the Filter to Other Protocols Naturally the variety of transport level protocols that the filter can process is important in order for the filter to still maintain effectiveness. As stated in the introductory chapter of this paper the goal of the filter was to count how many times packets collected using a network telescope matched certain

rules and it makes sense for the filter to be able to match datagrams that are not only TCP datagrams.

While different transport level protocols posses different structures as shown in chapter 3 the relevant fields of these protocols in this filter are the source and destination port numbers which are always 16 bits each and may occur at different offsets within a datagram. Leveraging the knowledge of transport protocol standards it was intended for another Java application to be written that, when parsing the rule file for the filter, notes each different transport protocol and generates a separate *rx module for each transport protocol (where * is the transport protocol e.g. TCPrx in the described design but not IPrx since IP is a network level protocol). These modules would be identical except for the byte counter values at which the datagram bytes are stored; these values would be such that the port numbers are retained within each *rx module. Since the protocols are mutually exclusive the datagram_present and transport_prot signals visible in figure 13 would be routed to each *rx module. The datagram_present module would enable all *rx modules but a module would only operate if the transport_prot signal carried the transport protocol number for that specific module. Each module would instantiate the same PortBitVec module to obtain the results of the port number component of the filter rules.

Such an extension to the filter design described above is vital to ensure that the filter is useful beyond filtering any datagrams besides TCP.

6.1.2 Means of reporting rule counts

Two other methods of reporting the rule counts were looked at but discarded in favour of using the serial interface on the development board. These options are discussed below.

Ethernet Frames The current design of the filter requires that the filter be connected to some device (such as a computer) capable of processing the data output over the serial interface in the manner required by the user. This has the obvious drawback of localising the operation of the filter to a single location where the user must be present at the other end of the serial cable to accept the data from the filter. A more versatile option is to report back the rule counts stored within the filter over the network on which the filter resides as yet another Ethernet encapsulated IP packet. UDP would be an adequate transport protocol to use in this process and so the implementation of a UDP/IP/Ethernet

stack within the filter design as a part of the report back process of the filter is attractive. However the most significant drawback of implementing this approach on the specific development board used in this project is with only one Ethernet connection available to the board the rule counts would either have to be issued after all Ethernet frames have been received (which may not be known) or the reception of Ethernet frames would have to be synchronised to the process of transmitting the rule counts stored on the device, this would hamper the overall performance time of the filter has it would have to stop receiving frames at some point to allow transmission of the rule counts.

LCD Screen The development board also has an on board a 16 character LCD screen to which data can be written or read from. For the purpose of merely counting the number of various rule matches outputting of data to the LCD screen would be sufficient to achieve this task with the selection of which rules are being displayed at any one time controlled through the use of any one of the boards various switches, buttons or the rotary dial. However, due to the specific operation of the LCD screen this approach would require that the 32 bit rule count be converted to a sequence of ASCII characters representing the value of the 32 bit binary number, a rather tedious process.

Bibliography

[1] Altera, 2010. Altera Corporation webpage.

[2] Zachary K. Baker and Viktor K. Prasanna. Time and area efficient pattern matching on fpgas. In *International Symposium on Field Programmable Gate Arrays*, pages 223 – 232, 2004.

[3] Peter Bellows, Jaroslav Flidr, Tom Lehman, Brian Schott, and Keith D. Underwood. Grip: A reconfigurable architecture for host-based gigabit-rate packet processing. In *In: Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 121–130. IEEE Computer Society Press, 2002.

[4] Stephen Brown and Jonathan Rose. Architecture of fpgas and cplds: A tutorial. *IEEE Design and Test of Computers*, 1996.

[5] Young H. Cho and William H Mangione-Smith. Deep network packet filter design for reconfigurable devices. In *ACM Transactions on Embedded Computing Systems*, 2008.

[6] Young H. Cho, Shiva Navab, and William H Mangione-Smith. Specialised hardware for deep network packet filtering. In *International Conference on Field Programmable Logic and Applications*. The University of California, 2002.

[7] Pawel Chodowiec and Kris Gaj. Very compact fpga implementation of the aes algorithm. In *5th International Workshop on Cryptographic Hardware and Embedded Systems*, volume 2779, pages 319–333, 2003.

[8] Adam Covington, John Lockwood Glen Gibb, and Nick McKeown. A packet generator on the netfpga platform.

[9] Gregory Ray Goslin. A guide to using field programmable gate arrays (fpgas) for application-specific digital signal processing performance, 1995. Xilinx Incorporated.

[10] Scott Hauck. The roles of fpgas in reprogrammable systems. In *Proceedings of the IEEE*, volume 86, pages 615–639, April 1998.

[11] Sourcefire Inc. webpage.

[12] Ji Li, Haiyang Liu, and Karen Sollins. Scalable packet classification using bit vector aggregating and folding. Technical report, MIT-LCS, 2003.

[13] David Moore, Colleen Shannon, Geoffery M. Voelker, and Stefan Savage. Network telescopes: Technical report. Technical report, San Diego Supercomputer Center, University of California and Computer Science and Engineering Department, University of California, 2004.

[14] James Moscola, John Lockwood, Ronald P. Loui, and Michael Pachos. Implementation of a content-scanning module for an internet firewall. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 2003.

[15] NetFPGA, 2010.

[16] Jean P. Nicolle, 2007.

[17] Alastair Nottingham and Barry Irwin. Gpu packet classification using opencl: A consideration of viable classification methods. In *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 160 – 169, 2009.

[18] Kaushik Ravindran, Nadathur Satish, Yujia Jin, and Kurt Keutzer. An fpga-based soft multiprocessor system for ipv4 packet forwarding. In *In Proc. 15th International Conference on Field Programmable Logic and Applications (FPL-05*, page 487492, 2005.

[19] Mireille Regnier. *Lecture Notes in Computer Science*, chapter Knuth-Morris-Pratt algorithm: An analysis, pages 431 – 444. Springer Berlin / Heidelberg, 1989.

[20] Martin Roesch and Chris Green. *SNORT Users Manual 2.8.6.* The Snort Project.

[21] Haoyu Song and John Lockwood. Efficient packet classification for network intrusion detection using fpga. In *International Symposium on Field-Programmable Gate Arrays*, 2005.

[22] Ioannis Sourdis and Dionisios Pnevmatikatos. *Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System.* Springer Berlin / Heidelberg, 2003.

[23] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, 1999.

[24] V. Srinivasan, G. Vargheset, S. Suri, and M. Waldvogelg. Fast and scalable layer 4 switching. In *ACM SIGCOMM Computer Communication Review*, 1998.

[25] Rishard Stevens. *TCP/IP Illustrated*, volume One. Addison Wesley Longman, Inc., thirteen edition, 1999.

[26] David E. Taylor. Survey and taxonomy of packet classification techniques. ACM Computing Surveys, 2005.

[27] Jean-Pierre van Riel and Barry Irwin. Identifying and investigating intrusive scanning patterns by visualizing network telescope traffic in a 3-d scatter plot. In *Proceedings of 6th Annual Information Security Conference*, 2006.

[28] Nicholas Weaver. *The Shunt: An FPGA-based accelerator for network intrusion prevention*, pages 199–206. ACM Press, 2007.

[29] Xilinx. *The Programmable Logic Databook 2000.*

[30] Xilinx. *Xilinx Content-Addressable Memory v6.1 Product Specification*, 2008.

[31] Xilinx Corporation. *Memory Interface Solutions User Guide.*

[32] Xilinx Corporation. *Spartan-3A/3AN FPGA Starter Kit Board User Guide*, v1.1 edition, 2008.

[33] Xilinx.com. clock skew (defn). website, 2005.

[34] Cho H. Young, Shiva Navab, and William H. Mangione-Smith. Specialized hardware for deep network packet filtering. In *Proceedings of 12th International Conference on Field Programmable Logic and Applications*, 2002.

[35] Jihan Zhu and Peter Sutton. Fpga implementations of neural networks - a survey of a decade of progress. In *Proceedings of the 13th Annual Conference on Field Programmable Logic and Applications*, pages 1062–1066, 2003.