Diploma Thesis in Electronic & Computer Engineering

Snort DPI on FPGA with GigE

Dimitrios - Stefanos Daskalakis

Chania, September 2012

Technical University of Crete
Electronic and Computer Engineering Department
Microprocessor & Hardware Lab

Advisor
Prof. Dr. Ioannis Papaefstathiou

Thesis Commitie
Prof. Dr. Apostolos Dollas
Prof. Dr. Dionisios Pnevmatikatos

Graduation Date
24.09.2012

Contents

Abstract

Massive growth in data processing power and new cyber threats have spurred the deployment of Deep Packet Inspection (DPI) technologies. These technologies are currently used by network intrusion detection and prevention systems, like (NIDS, IPS, IDPS), to efficiently filter inbound and outbound traffic and prevent sophisticated intrusions, such as Denial of Service (DOS) and buffer overflow attacks. Additionally, these systems require to operate at high speed to avoid a reduction of the Quality of Service (QoS), packet loss and additional latency.

In this Diploma thesis we present a customizable hardware DPI system, designed to support multi-layer packet decoding combined with multiple pattern matching against the transport-layer payload. Additionally, we incorporate a DPI configuration tool for automatic hardware code generation, to increase the adjustability to new rules and reduce maintenance. The tool creates the "header" and "pattern" matching files, from Snort compatible rules.

Finally, we evaluate our approach by instantiating a 750 rule configuration on a Virtex 5 FPGA with a 1Gbps GigE network interface. The results show that we met the time requirements, as there is no decrease at the throughput with only a store-and-forward delay.

1. Introduction

The information revolution in modern society and the growing number of personal computers, have increased the use of Internet. Most companies nowadays store all of their information on computers and database servers. Many computer applications connect to the web to exchange data such as chat clients, p2p programs, games, web browsers and others. All these applications may have vulnerabilities due to the lack of proper coding or bugs from the operating system. Firewalls, sometimes together with anti-virus, had a decent performance for many years at preventing hacker attacks, malicious programs and spam or phishing e-mails. But most of them would lack the ability to prevent buffer overflow attacks, DoS attacks, and exploits.

More recent protection systems, such as "Intrusion Detection Systems" IDS and "Intrusion Prevention Systems" use Deep Packet Inspection (DPI) to inspect packets in multiple network layers. A DPI system is a form of computer network packet filtering that examines the header and data part of a packet as it passes an inspection point.

Snort is a powerful open-source "Network Intrusion Detection System" NIDS that uses DPI to inspect the network traffic. It has by default a large rule database of suspicious signatures to block most of the known network attacks. This database is frequently updated in order to adjust to new attacks.

Motivation

Although many software DPI systems have been developed and released, they have a moderate network throughput even though large and energy intensive computers are used. In the worst case, when their throughput can not keep up with the network traffic, they drop or pass packets without inspecting them. Hackers exploit this disadvantage and create large virtual network traffic in order to attack servers.

Hardware based DPI can perform much better in terms of network throughput, but has a drawback in scalability and frequently updates of intruder signatures. Using FPGA's for DPI is an advantage, since the reconfiguration of the chip is an easy process.

In our work we will present a complete DPI system that operates on a GigE network interface. To configure the function of the DPI, we developed a tool that generates hardware description code from a Snort compatible rule file.

Focus

This work will analyze what Deep Packet Inspection is, why and where it is used. We present the different methods of DPI in software and hardware, the advantages and disadvantages of each method. We propose a datapath of a complete DPI system together with a customization tool. The results of the implementation on a Virtex5 and real data transfer over GigEthernet. Also ideas for future work to make this system even better.

Contributions

Within this work, we have created a stand-alone system for Deep Packet Inspection based on the Snort rules. We used a FPGA from the Virtex-5 family to implement 750 Snort rules identification system, a GigEthernet receiver and transmitter, a packet decoder and encoder. Further more we developed a tool for automatic VHDL code generation from a Snort rules files, this extends the functionality of our system by letting it easily to adjust to new rules and operations.

Recapitulating, this work has made the following basic contributions.

1. A stand-alone customizable DPI system with GigEth interface
2. Tools for customization of the system from Snort rule compatible files
3. Support of multiple patterns, IPs and ports on rules
4. Multiple rules match inside the same frame simultaneously

Table 1.1 shows the features of the implemented system.

Table 1.1.: List of Features

Feature	Description
Implemented Snort rules	750
Wire-speed operation	1Gb/s
Decoding	MAC, IP, TCP, UDP, ICMP
Multiple match on frame	Yes
Simultaneous match more than one rules	Yes
Customization tool	Yes, from Snort rules
Maximum number of rules supported	4095

2. Deep Packet Inspection Systems

The numerous attacks from the Internet like viruses, spam, software vulnerabilities as well as other malicious activities have increased the need for network security. For this reason a variety of methods have been used to protect data and services: firstly through cryptography, firewalls, IDS and finally with intrusion prevention systems (IPS).

Most intruder activity has some sort of signature. There are databases of known vulnerabilities that intruders want to exploit. These known attacks are also used as signatures to find out if someone is trying to exploit them. The signatures can be found in header parts of a packet or in the payload. A DPI system uses these signatures in order to prevent intruder attacks.

2.1. DPI Challenges

The DPI is commonly used by proxies, packet filters, sniffers, IDS, and IPS. DPI systems have many different challenges to overcome. Abuhmed et al[6] state the most important ones.

1. **Complexity of search algorithm** : the pattern matching algorithm complexity is the major affecting factor of system performance. Many DPI researches deal with optimization of these algorithms.
2. **Wire speed processing**: DPI systems have to perform in worst case at wire speed to prevent buffer overflow, DoS attacks and avoid dropping unchecked incoming packets.
3. **Growing number of signatures**: the growing number of signatures require continuously increasing performance of the DPI systems.
4. **Bad signatures**: bad written signatures may generate false alerts, drop packets that are harmless or use too many resources of the system.
5. **Encrypted data**: encrypted data packets cannot be inspected. A solution to this could be the placement of DPI after the decryption device.
6. **Scalability**: scalability is a big issue in hardware designs. ASIC devices usually are not scalable and most FPGA implementations need regeneration of the initialization bit-stream to overcome this problem.

2.2. Software DPI systems

Many packet scanning applications use DPI systems. The most popular open-source ones are Snort[16], Bro[14] and L7-filter[2] for Linux. Snort and Bro are intrusion detection systems (IDS), whereas L7-filter is an application layer data classifier. Next we will deal with Snort as it is one of the most popular IDS.

2.2.1. Snort IDS

Snort is an open source Network Intrusion Detection System (NIDS), it is extensively used, and is ranked among the best systems available today. Snort uses rules in order to describe the intruders signatures. These rules are stored in files which, along with a configuration file, define Snort's operation. Our work is based on these rule files, which are used to generate hardware description code that inspects receiving data for suspicious attacks. An example of a rule can be found bellow, while more information about Snort's rule syntax and the supported keywords of this work can be found in Appendix A.

```
alert tcp any any -> 192.168.1.0/24 111 (content:  "idc|3a3b|"; msg:
                            "mountd access";)
```

Snort can perform protocol and header packet analysis, but the heart of the program is a string matching algorithm that accounts for up to 70% of total execution time.

Many different algorithms have been used for this purpose. At the beginning brute-force pattern matching was used, with a great impact on the performance. Boyer Moore[8] algorithm increased significantly the performance (about 200-500%).

Boyer & Moore created an efficient string search algorithm that uses heuristics to reduce comparisons. The major drawback is the space required, which depends on the number and length of patterns that create multiple substrings of the incoming data.

Latest version of Snort 2.9 uses optimized versions of the Aho - Corasick[7] algorithm, which is supposed to be a powerful multiple pattern matching algorithm. It constructs a trie like finite state automaton that matches the given patterns simultaneously. Figure 2.1 shows an example of a simple automaton that searches for words “HE”, “HIS”, “SHE”, “HERS”.

If the network traffic is too high and the detection engine fails to operate at this speed, some packets may be dropped or forwarded without inspection. The load of the detection engine depends on several parameter, with notable the following ones:

- Network load
- CPU power of the machine running Snort

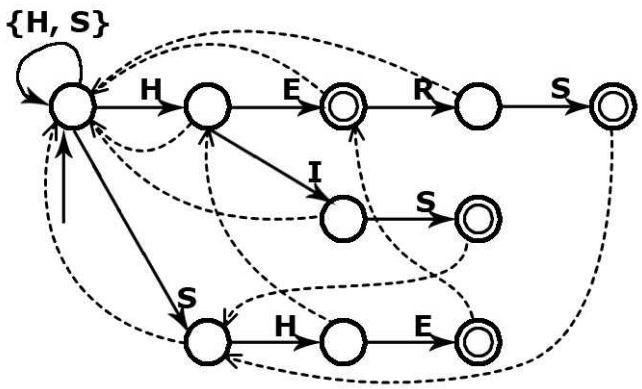


Figure 2.1.: Aho - Corasick FSM example

- RAM memory size
- Number of rules

2.3. Hardware DPI systems

The software DPI systems aren't fast enough to operate on demanding networks. Several companies have developed ASIC security programmable co-processors such as Cisco, NetScreen and PMC-Sierra. These programmable security co-processors have higher (but not impressive) performance compared to general purpose processors furthermore they are complicated and expensive.

Many different hardware DPI systems have been proposed by researchers. The Hardware implementations can be divided into three main categories.

1. Content Addressable Memory (CAM)
2. FPGA implementations
3. Multi-core network processors (NP)

2.3.1. Content Addressable Memory

Content addressable memory (CAM) is a computer memory used by many commercial high speed pattern matching applications. In a RAM a user supplies an address and the memory returns the stored data word. A CAM works the opposite way, the user supplies a data word and the memory returns the address of the data if it is stored anywhere in it. The main disadvantages of CAM are its cost and the power consumption. Each memory bit must have its own associated comparison circuit in order to detect a match and additionally the cells must be combined to find a

complete data word match. Every comparison circuit has to be active on every clock cycle which means very high power consumption.

A more complex CAM is the Ternary CAM (TCAM) which allows for a third matching state "X" or "don't care" that adds flexibility to the search. For example a TCAM may have stored the word "101010XX", which means that any of the four search words ("10101000", "10101001", "10101010", "10101011") would match.

CAMs and TCAMs are often used by network switches, routers, firewalls, network address translation and network intrusion detection systems (NIDS) based on DPI.

2.3.2. FPGA Implementations

FPGA implementations have the advantage of flexibility on search pattern alterations, they can reconfigure the design and keep the interface unchanged. A lot of pattern matching algorithms have already been implemented for FPGAs. These include discrete comparators, Bloom Filters, regular expression based pattern matching (NFAs and DFAs) and Knuth - Morris - Pratt string matching algorithm.

Discrete Comparators

A straightforward approach for pattern matching in FPGA is the use of discrete comparators. This leads to high frequency designs but with increased area cost. Each pattern uses at least one comparator (logic cells). Every character of a pattern is stored in LUTs. For example a virtex 5 FPGA has 6 bit LUTs which means 4 LUTs have to be used to store 3 characters. A reduced area cost discrete comparator solution has been proposed by I. Sourdis[17].

Bloom Filters

Bloom Filters store the patters as hash values, they have reduced area cost and good performance but allow false positives which is a major drawback. They also require multiple hash functions for different pattern lengths. Researchers have proposed parallel different bloom filters to reduce the probability of false positives[10].

Finite State Machines

Finite State Machines (FSM) have been used by DPI systems. The FSM implementations can be categorized in two groups, the Deterministic Finite Automata (DFA) and Nondeterministic Finite Automata (NFA).

The DFA is an automaton that has a set of input symbols called the alphabet, a finite set of states, a start state, a set of accept states and a transition function to move between the states. At each time the DFA has only one active state.

In the other hand NFA is a FSM where from each state and a given input symbol it can jump to several possible next states. A NFA may have a multi acceptance state.

Knuth - Morris - Pratt

Knuth - Morris - Pratt (KMP)[13] is an easy and well known string matching algorithm. The KMP, creates a graph for the matching pattern, which can be implemented in a FSM. An example of a graph that matches the pattern "ababaa" is shown in Figure 2.2. As we can see in the graph in case of a mismatch, the state machine is designed to jumps to next biggest substring matched (e.g. state 3 jumps to state 1 and state 4 jumps to state 2). In state 4 the fsm has already matched the substring "abab", when a mismatch occurs it will jump to state 2 having matched the substring "ab".

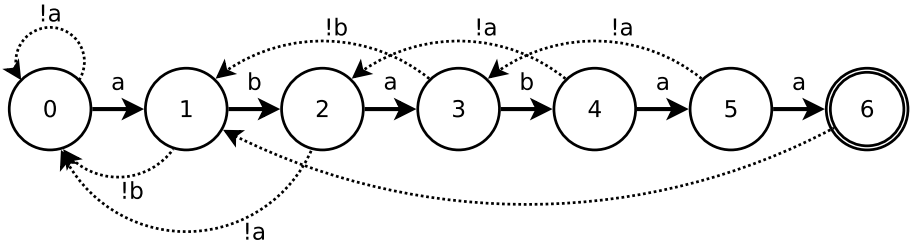


Figure 2.2.: KMP graph for "ababaa"

The main disadvantage of KMP is the difficulty to process one incoming character per cycle. In case of a mismatch two or more comparisons must perform for one character.

2.3.3. Multi-core Network Processors (NP)

Network processors are typically software programmable devices like general purpose CPUs with added optimized features or functions such as:

- Pattern matching
- Key lookup (e.g. address lookup)
- Data bitfield manipulation
- Packet queue management

Parallel processing with multiple cores is able to achieve higher throughput. The main advantage of NP is its flexibility to adapt on a new pattern searching set. On the other hand NPs have moderate throughput of of few hundred Mbps.

2.4. Hardware DPI Systems with Network Interface

Closer to our own work are DPI systems with network interface, capable to manage real network traffic. An implementation with four GigE network ports was proposed by C. Clark[9]. They used a Xilinx ML300 reference platform with a V2P7 FPGA to implemented a small fraction Snort's rule-set (71 rules, 505 chars) and achieved an impressive throughput of 4Gbps. Their header decoder supports IP, ARP, ICMP. TCP and UDP protocols. A payload processor is used to search the incoming packets against specified patterns, the processor uses a non-deterministic finite state automata approach, supports to compare up to 8 characters per cycle. The drawback of this implementation is the high area cost and small rule-set implementation.

3. DPI Architecture

3.1. Introduction

This chapter describes the architecture of a fully functional Deep Packet Inspection system that we designed in order to implement a compatible Snort sub-Ruleset in FPGA.

The implementation in FPGA's is an advantage compared to the ASIC because of the frequent updates of IDS rulesets. The modification or addition of even one rule would require the creation of a new ASIC. With FPGA's instead we need simply regenerate the changed code and download it on the FPGA chip.

The following sections describes in detail the overall system architecture as well as the architecture of the various subsystems. The last section describes in detail the code in C that was created to translate Snort rule files into VHDL code.

3.2. System Overview

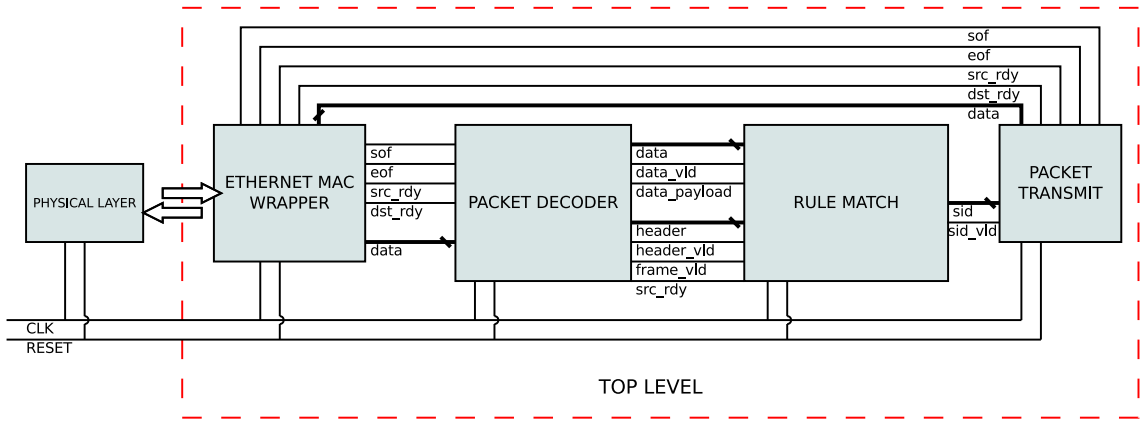


Figure 3.1.: Top Level of DPI System

3.2.1. Description

The block diagram in Figure 3.1 is an overview of the Deep Packet Inspection system architecture which uses a Ethernet Mac wrapper, a Packet Receiver, a Rule Matching

Module and a Packet Transmit Module.

The purpose of this system is to listen on incoming packets from the Physical Interface, inspect the header and the payload of the packet and find if it matches with a rule, send a raw Ethernet packet with the SID for each matched rule to the Physical Interface. The system operates in real time, with a clock at 125Mhz and data rate of 1 byte per cycle.

Specifically, when a new packet arrives the Embedded TEMAC (in Ethernet Mac wrapper) verifies the incoming FCS on every frame, strips off Preamble, SFD, Pad and FCS from the Ethernet frame (see Figure 3.2). The remaining frame is stored in a FIFO memory and frames marked as bad are dropped.

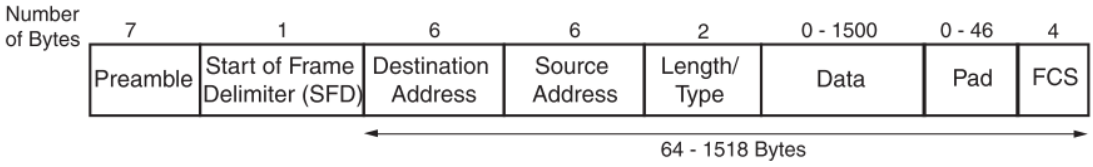


Figure 3.2.: Standard Ethernet Frame Format

After the frame has been stored in the receiver FIFO, the Receiver Module requests the new frame from the FIFO and extracts useful header information from MAC Layer, network layers (IPv4, ICMP) and transport layers (TCP, UDP). Any other layer is not being decoded.

Frame data and header information is passed to the Rule Match Module which in turn finds if the packet matches any of the given rules. The rules may contain:

- source & destination ip address check
- source & destination port number check
- string content match in the payload of the packet
- checks against the IP protocol header
- specific ICMP type value match.

In case a rule has been detected the SID of the rule is send to the Packet Transmit Module, which stores the SID value in a FIFO and transmits a 60bytes raw Ethernet packet with the SID value to the Ethernet Mac wrapper.

3.2.2. System Inputs

Table 3.1 describes the input signals of the DPI System.

Table 3.1.: DPI System Inputs

Signal	Direction	Description
CLK_100	Input	The clock input. 100Mhz clock is the input to a DCM which creates a 125Mhz clock used by the system.
GMII_RX_CLK_0	Input	GMII Receive Clock. GMII_RX_CLK_0 provides a 125 MHz clock reference for GMII_RX_DV_0, GMII_RX_ER_0, and GMII_RXD_0[7:0].
GMII_RXD_0[7:0]	Input	GMII Receive Data. Symbols received on the cable are decoded and presented on GMII_RXD_0[7:0].
GMII_RX_DV_0	Input	GMII Receive Data Valid. When GMII_RX_DV_0 is asserted, data received on the cable is decoded and presented on GMII_RXD_0[7:0] and GMII_RX_ER_0.
GMII_RX_ER_0	Input	GMII and MII Receive Error. When GMII_RX_ER_0 and GMII_RX_DV_0 are both asserted, the signals indicate an error symbol is detected on the cable.
RESET	Input	System Reset signal. Active high.

3.2.3. System Outputs

Table 3.2 describes the output signals of the DPI System.

Table 3.2.: DPI System Outputs

Signal	Direction	Description
GMII_TXD_0[7:0]	Output	GMII Transmit Data presents the data byte to be transmitted onto the cable. GMII_TXD_0[7:0] are synchronous to GTX_CLK and GMII_TX_CLK_0.
GMII_TX_EN_0	Output	GMII Transmit Enable. When GMII_TX_EN_0 is asserted, data on GMII_TXD_0[7:0] along with GMII_TX_ER_0 is encoded and transmitted onto the cable. GMII_TX_EN_0 is synchronous to GMII_TX_CLK_0.
GMII_TX_ER_0	Output	GMII Transmit Error. When GMII_TX_ER_0 and GMII_TX_EN_0 are both asserted, the transmit error symbol is transmitted onto the cable. GMII_TX_ER_0 is synchronous to GMII_TX_CLK_0.
GMII_TX_CLK_0	Output	GMII Transmit Clock. GMII_TX_CLK_0 provides a 125 MHz clock reference for GMII_TX_EN_0, GMII_TX_ER_0, and GMII_TXD_0[7:0].
RESET_PHY	Output	Reset of the physical layer device. Active low.

3.2.4. Subsystems

- **Ethernet Mac wrapper.** Created by Xilinx Core Generator. A wrapper that allows the system to communicate with the Physical layer. Described in more detail in section 3.3
- **Packet Receiver.** Decodes the incoming packet, extracts header data and passes the data to data_out with one clock cycle delay. Described in detail in section 3.4
- **Rule Match module.** Compares the incoming data and header with preset values and searches for any rule match. If a rule has been matched the module sends the SID value of the rule and asserts the sid_vld signal. Described in detail in section 3.6
- **Packet Transmit module.** Creates a raw Ethernet packet with every SID value being received. Described in detail in section 3.9

3.3. Ethernet Mac wrapper

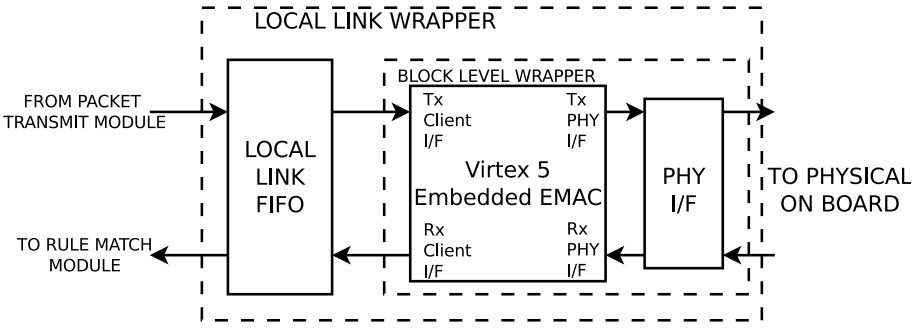


Figure 3.3.: Ethernet Mac wrapper Design

3.3.1. Description

The Ethernet MAC wrapper instantiates the full Ethernet MAC primitive and is generated by Xilinx Core-Generator. The device is utilized to support full-duplex 1Gbps-only operation with physical interface GMII that connects to the I/O of the FPGA.

3.3.2. LocalLink Interface

The Ethernet FIFO consists of independent receive and transmit FIFOs embedded in the Local Link wrapper. Each FIFO is built around 2 Dual Port block RAMs providing a memory capacity of 4096 bytes in each FIFO.

Data is transfered on Local Link interface with the flow governed by the four active low signals *sof, eof, src_rdy* and *dst_rdy*. Only when the signals *src_rdy* and *dst_rdy* are asserted simultaneously is data transferred, the individual packet boundaries are marked by *sof* and *eof* signals. Figure 3.4 shows the transfer of an 8-byte frame.

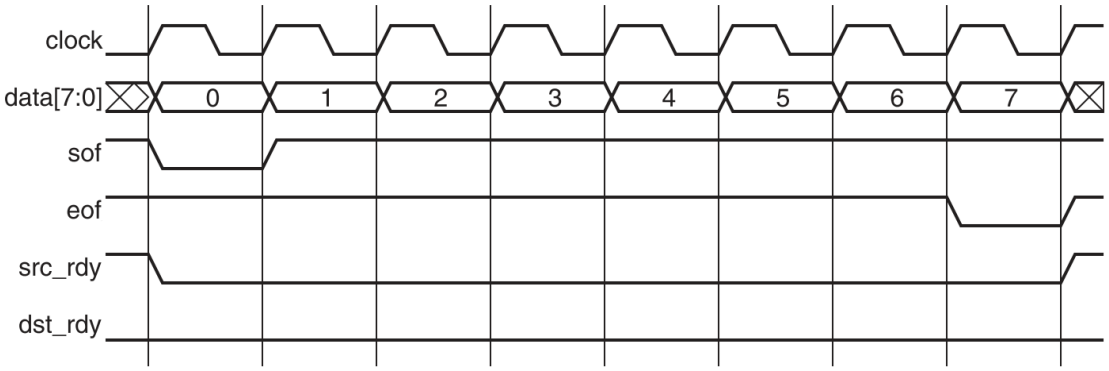


Figure 3.4.: Frame Transfer across LocalLink Interface

Table 3.3 describes the receive FIFO LocalLink interface.

Table 3.3.: Receive FIFO LocalLink Interface

Signal	Direction	Description
rx_ll_clock	Input	Read clock for LocalLink interface
rx_ll_reset	Input	Synchronous reset
rx_ll_data[7:0]	Output	Data read from FIFO
rx_ll_sof_out	Output	Start of frame indicator
rx_ll_eof_out	Output	End of frame indicator
rx_ll_src_rdy	Output	Source ready indicator
rx_ll_dst_rdy	Input	Destination ready indicator

Table 3.4 describes the transmit FIFO LocalLink interface.

Table 3.4.: Transmit FIFO LocalLink Interface

Signal	Direction	Description
tx_ll_clock	Input	Write clock for LocalLink interface
tx_ll_reset	Input	Synchronous reset
tx_ll_data[7:0]	Input	Write data to be sent to transmitter
tx_ll_sof	Input	Start of frame indicator
tx_ll_eof	Input	End of frame indicator
tx_ll_src_rdy	Input	Source ready indicator
tx_ll_dst_rdy	Output	Destination ready indicator

3.3.3. GMII Physical Interface

The Gigabit Media Independent Interface (GMII), defined in IEEE 802.3, clause 35, is an extension of the MII used to connect a 1-Gb/s capable MAC to the physical sublayers.

The physical signals of the Ethernet MAC are connected through IOBs to the external interface. The Figure 3.5 shows the GMII physical interface connections.

Signals:

- **GMII_RX_CLK** (1 bit input) The clock signal input is sourced by the external PHY device and should be used by the FPGA logic to clock the receiver physical interface logic. Internally in the Ethernet MAC, this clock is used to derive all physical and client receiver clocks.
- **GMII_RXD** (8 bit input) The received data signal to the PHY.
- **GMII_RX_DV** (1 bit input) The data valid control signal from the PHY.
- **GMII_RX_ER** (1 bit input) The error control signal from the PHY.
- **GMII_TX_CLK** (1 bit output) This clock is generated by a DCM and provides a 125Mhz frequency to the PHY.
- **GMII_TXD** (8 bit output) The transmit data signal to the PHY.
- **GMII_TX_EN** (1 bit output) The data enable control signal to the PHY.
- **GMII_TX_ER** (1 bit output) The error control signal to the PHY.

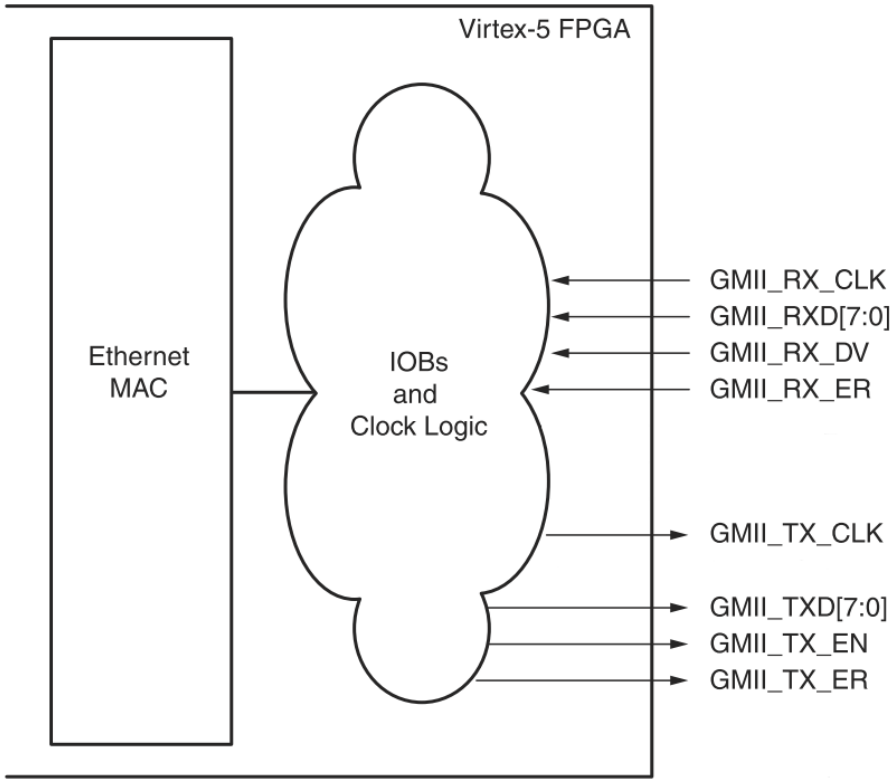


Figure 3.5.: GMII Physical Interface

3.4. Packet Receiver

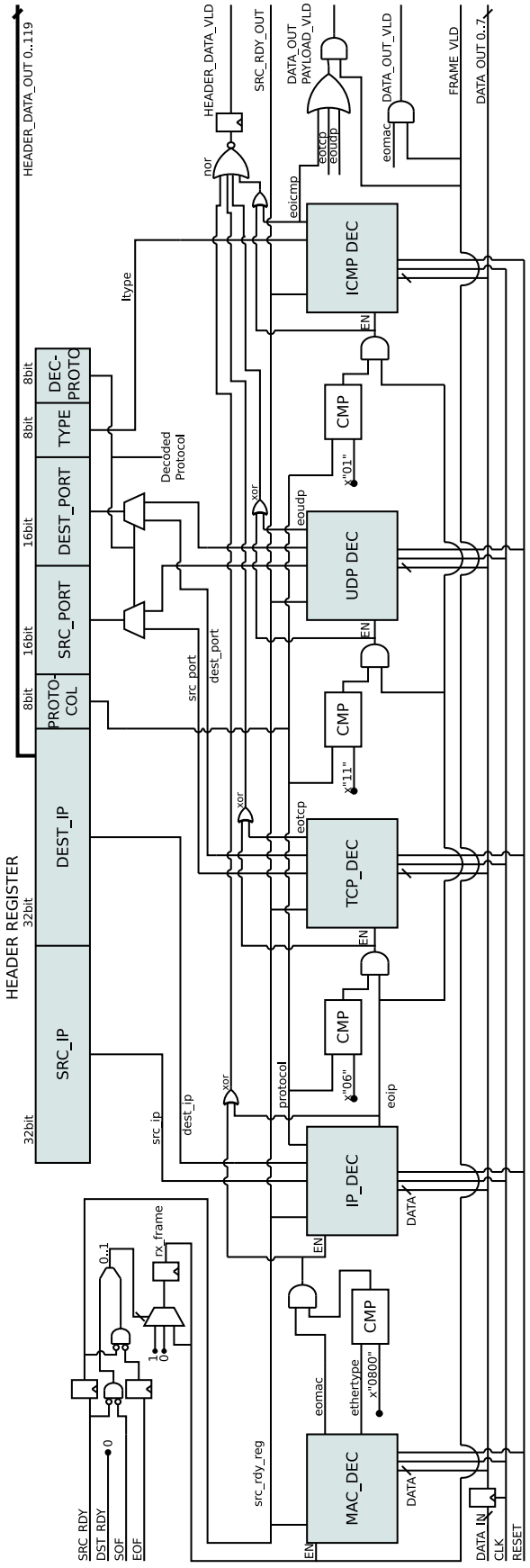


Figure 3.6.: Packet Receiver

3.4.1. Description

The purpose of the Packet Receiver is to unpack the received data, extract valuable header values, find where the packet and payload starts and ends. The block diagram In Figure 3.6 is an overview of the datapath of the Receiver.

First of all to receive a frame, *src_rdy* and *dst_rdy* has to be asserted low to 0 as shown in Figure 3.4. The signal *dst_rdy* is connected to ground and *src_rdy* is input from the FIFO's Interface. Also to receive a new frame, signal *sof* has to be 0 for the first byte of *data_in* to indicate the start of the frame. In this case *rx_frame* register turns high to 1 and holds this value until an *eof* signal is transmitted from the FIFO, *eof* is delayed by one clock cycle so that the last byte of data remains valid.

The *rx_frame* signal triggers the enable signal of Mac Decoder, which decodes the Mac layer and extracts the ethertype value which is used to indicate the protocol that is encapsulated in the payload of the Ethernet frame.[4]

IP Decoder is enabled when the *End of Mac (eomac)* is asserted high and ethertype is x"0800", this means that the following protocol is IPv4. The IP Decoder extracts from the header, the source & destination ip address and the protocol, that is used in the data portion of the IP datagram. The IP Decoder supports an IPv4 Header format of 20 bytes with an optional options field.

The protocol byte from the IP header is compared with x"06" (TCP), x"11" (UDP) and x"01" (ICMP) and *anded* with the *End of IP (eoip)* signal to trigger the corresponding enable signal of TCP, UDP or ICMP Decoder.

TCP and UDP Decoder when enabled, they extract the source and destination port number from the header. When the header of the packet ends, *eotcp* or *eoudp* is asserted signal. The Decoders support TCP and UDP packets as described in RFC793 and RFC768.

If Icmp Decoder is enabled, then the Decoder extracts the type of the icmp packet. This information is useful for Snort rules that search for suspicious Icmp packets. Signal *eoicmp* is asserted when the header of the packet ends. The Decoder supports the ICMP packet as described in RFC792

After extracting all the header data from the nested packets, a registered Header Data output (width 120 bit) is created. This register contains Source IP address (32 bit), Destination IP address (32 bit), the Protocol (8 bit) that is used in the data portion of the IP datagram, Source Port (16 bit) if a UDP or TCP packet was decoded, Destination Port (16 bit) if a UDP or TCP packet was decoded, icmp Type (8 bit) if an Icmp packet was decoded and the Decoded Protocol (8 bit) x"04" for IP, x"06" for TCP, x"011" for UDP and x"01" for ICMP.

A Header Data valid signal is asserted high when all the enabled decoders rise an end signal. This is established by *xor*ing the enable signal with the end signal of

all the Decoders and *nor* them. Which ensures that all activated Decoders have terminated.

The following signals are described briefly:

- Frame valid is true when the rx_frame register is 1.
- Data Out valid is true when rx_frame is 1 and Data Out is the payload of a Mac frame.
- Data Payload valid signal is true when the Data Out is a TCP, UDP or ICMP payload data.
- Src_Rdy_Out is same as src_rdy_in delayed by one clock cycle.
- Data Out is same as Data In delayed by one clock cycle.
- Frame valid is true when the rx_frame register is 1.
- Data Out valid is true when rx_frame is 1 and Data Out is the payload of a Mac frame.
- Data Payload valid signal is true when the Data Out is a TCP, UDP or ICMP payload data.
- Src_Rdy_Out is same as src_rdy_in delayed by one clock cycle.
- Data Out is same as Data In delayed by one clock cycle.

3.4.2. Inputs

Table 3.5 describes the input signals of Packet Receiver.

Table 3.5.: Packet Receiver Inputs

Signal	Direction	Description
CLK	Input	Clock signal
RESET	Input	Reset signal
DATA_IN[7:0]	Input	Data in from FIFO
SOF	Input	Start of frame indicator
EOF	Input	End of frame indicator
SRC_RDY	Input	Source ready indicator

3.4.3. Outputs

Table 3.6 describes the output signals of Packet Receiver.

Table 3.6.: Packet Receiver Outputs

Signal	Direction	Description
DST_RDY	Output	Destination ready indicator
DATA_OUT[7:0]	Output	Data out
FRAME_VLD	Output	Frame valid signal
DATA_OUT_VLD	Output	Data out valid signal
DATA_OUT_ PAYLOAD_VLD	Output	Indicator of (Tcp, Udp, Icmp) packet payload
SRC_RDY_OUT	Output	Source ready out
HEADER_DATA_ VLD	Output	Header data valid indicator
HEADER_DATA_ OUT[119:0]	Output	Header data out

3.4.4. Subsystems

- **MAC Decoder** that strips off mac layer header
- **IP Decoder** extracts source & destination IP address and protocol type of the next layer.
- **TCP Decoder** extracts source & destination port.
- **UDP Decoder** same as tcp decoder, extracts source & destination port.
- **ICMP Decoder** extracts the icmp type.

The Decoders are described in more detail in section 3.5

3.5. Decoders

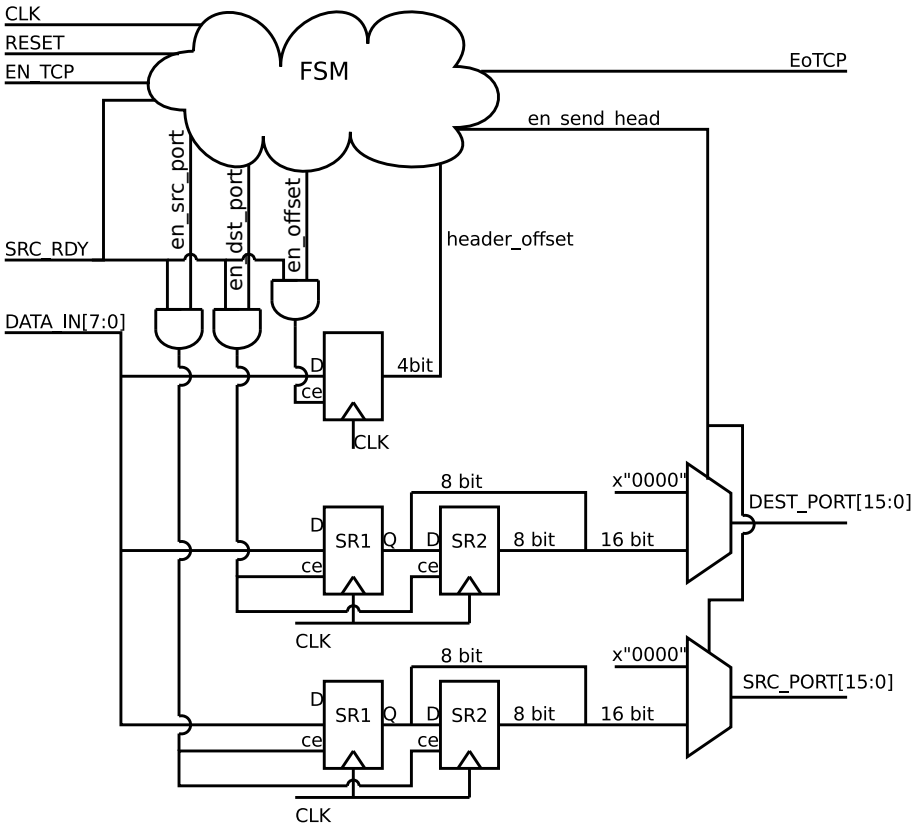


Figure 3.7.: Overview of TCP Decoder

3.5.1. Description

The Decoders of MAC, IP, TCP, UDP and ICMP have almost the same datapath and FSM, the most complex is the TCP decoder which has to decode a non standard size of header. To avoid duplication we will analyze only the case of TCP Decoder. Figure 3.7 shows the overview of the TCP Decoder.

The FSM controls two shift registers, a register that stores the offset of the header, two Mux that send the source and destination port and EoTCP signal that indicates the end of the TCP header. The shift registers are designed to store the source and destination ports when enabled. Figure 3.8 shows the FSM of the TCP Decoder.

The inputs of the FSM are clk, reset, en, src_rdy. When reset = 1 or en = 0 the FSM goes to initial state to read the first byte of source port. If src_rdy = 1 the FSM remains at its current state, this means that the current data byte is not valid because the source (FIFO receiver) was not ready. After reading the source and destination port, the Decoder reads the header offset from the packet header and compares it with a variable that stores the current length of header. This variable

has initial value 20, this is the length in bytes of the default TCP header without options. If the header offset is greater than the variable, the FSM waits for one cycle and rechecks. When the header offset is equal to the variable value the FSM jumps to end of header state and asserts the signals EoTCP and en_send_head high.

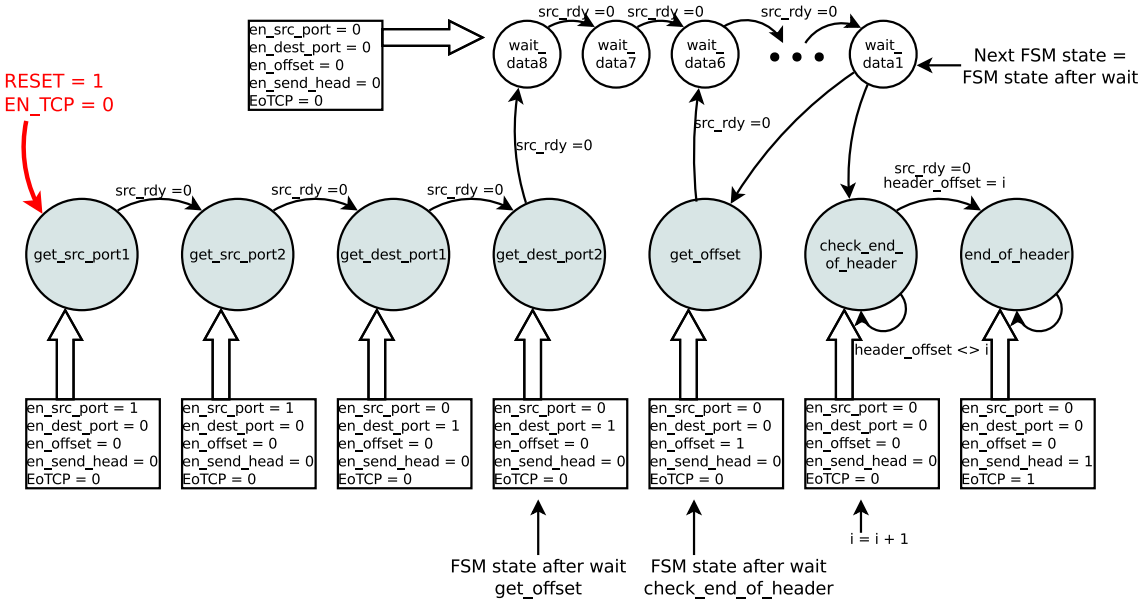


Figure 3.8.: FSM of TCP Decoder

3.5.2. Inputs

Table 3.7 describes the input signals of the Decoders.

Table 3.7.: Decoder Inputs

Signal	Direction	Description
CLK	Input	Clock signal
RESET	Input	Reset Signal
EN_MAC, EN_IP, EN_TCP, EN_UDP, EN_ICMP	Input	Decoders enable signals
SRC_RDY	Input	source ready signal
DATA_IN[7:0]	Input	8 bit data in

3.5.3. Outputs

Table 3.8 describes the output signals of the Decoders.

Table 3.8.: Decoder Outputs

Signal	Direction	Description
ETHERTYPE	Output	Packet protocol that follows the MAC layer, only for MAC Decoder
PROTOCOL	Output	Packet protocol that is encapsulated in the IP data, only for IP Decoder
SRC_IP[31:0]	Output	Source IP address, only for IP Decoder
DEST_IP[31:0]	Output	Destination IP address, only for IP Decoder
SRC_PORT[15:0]	Output	Source Port, only for TCP and UDP Decoders
DEST_PORT[15:0]	Output	Destination Port, only for TCP and UDP Decoders
ITYPE[7:0]	Output	ICMP type, only for ICMP Decoder
EoMAC, EoIP, EoTCP, EoUDP, EoICMP	Output	End of header signal of Decoders

3.6. Rule Match

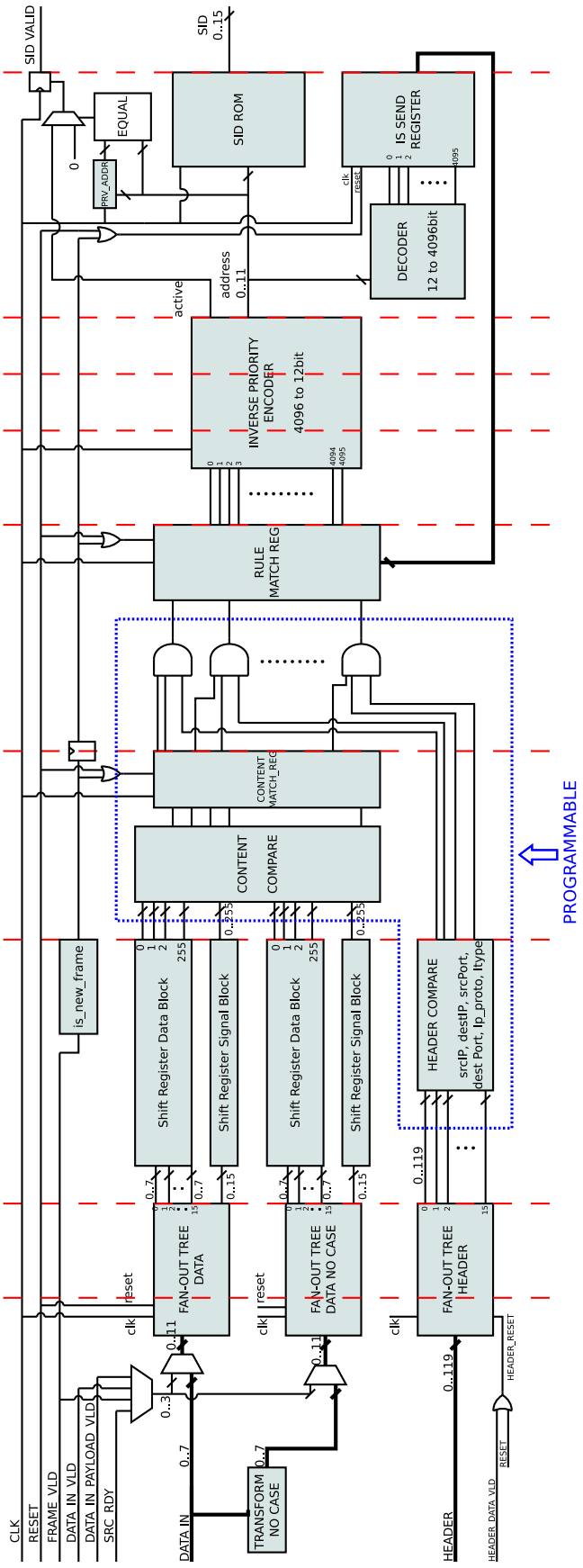


Figure 3.9.: Overview of Rule Match Datapath

3.6.1. Description

Figure 3.9 shows the datapath of the Rule Match module. The entire design is fully pipelined and meets the desired clock rate. Data is received in vectors *Data in* (8 bit) and *Header data* (120 bit). A process (Transform No Case) creates a new vector of *data in* that has no upper case characters to support non-case sensitive content rules. This process checks if *data in* is greater or equal to x"41" (character "A") and less or equal to x"5A" (character "Z"), in this case *data in nocase* vector is equal to *data in* + x"20" else *data in nocase* is equal to *data in*.

Data In, Data In Nocase, Header Data and signals *Frame vld, Data in vld, Data in payload vld* feed three Fan-Out Trees. These Fan-Out trees are designed to distribute the incoming data to 16 copies in the FPGA with 2 clock cycles delay. This is implemented with 4 registers in first level, which feed 16 registers in the second level. Each registered output of the tree is the input to 16 shift registers. For every fan-out tree we create up to 256 shift registers. We also create a shift register block of the *Data in payload vld* signal, this information lets us know when the compared data is in the payload of the packet.

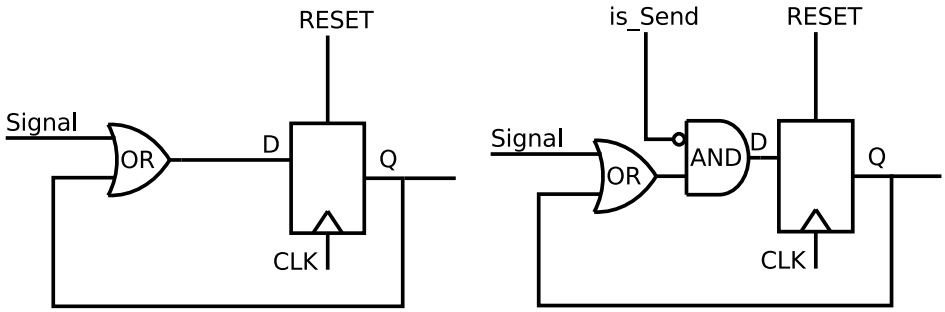
Header Compare module, uses comparators to match preset values of header options (ip address, port address ...) against the registered tree-outputs of *header data*. Each registered output of the header tree is the input to 16 comparators. This means that our design supports to check:

- 256 distinct source IP addresses
- 256 distinct destination IP addresses
- 256 individual source Ports or 128 range source Ports
- 256 individual destination Ports or 128 range source Ports
- all ICMP types
- all protocols

Content Compare module, uses comparators to match preset text strings against the shift registered data in. It also performs a check if the data is inside the borders of the packet payload data. Each shift register feeds 15 comparators, which means that this design can support up to:

- 3840 distinct case sensitive strings
- 3840 distinct non-case sensitive strings

When a content has been matched a register (with simple logic added) will be raised and hold it's value until reset (Figure 3.10a). With this registers we can support more than one separate content matches in the same rule, because the text matches may occur at different clock cycles and the information of the previous match will have been stored.



(a) Content Match Register **(b)** Rule Match Register

Figure 3.10.: Match Registers

The registered results of the comparators are combined together using an AND gate for every rule. A register for each matched rule will be asserted high until it is sent or reset (Figure 3.10b).

An Inverse Priority Encoder 4096 to 12bit is used to encode the matched rules. This encoder supports 4095 rules beginning from address x“001” to x”FFF” , address x“000” is reserved and not assigned to a Rule to avoid conflicts with the default output of the encoder. The encoder has 3 clock cycles delay and encodes the signals in three stages using small 16 to 4 priority encoders.

Rule Match module has an output vector with the matched rule’s sid and a signal that indicates when the vector is valid. A read only memory SID Rom is initialized with the Sid’s of the rules. The encoder’s output is the read address of the memory. The sid valid signal is asserted high for every new valid address, this is implemented by comparing the current address with the registered (1 clock) previous.

To support more than one rules to be matched and sent in the same data packet, we use a 12 to 4096 Decoder that decodes the output address of the encoder. The signal outputs of the Decoder are connected to registers that will remain high until reset. These registers are the input signals (is_sent) in rule match registers and when enabled they set the output of the register to zero (see Figure 3.10b). By zeroing the register of the sent rule, we let the encoder to send the next highest priority matched rule.

The registers of the whole module will reset when a new frame arrives. This is implemented by tracking the *Frame_vld* signal and sending a *is_new_frame* signal each time Frame_vld signal changes value from low to high.

The whole module is created automatically by an executable script with a text file of Snort Rules as input.

3.6.2. Inputs

Table 3.9 describes the input signals of the Rule Match module.

Table 3.9.: Rule Match module Inputs

Signal	Direction	Description
CLK	Input	Clock signal
RESET	Input	Reset signal
FRAME_VLD	Input	A valid frame is received when high
DATA_IN_VLD	Input	Valid data in signal
DATA_IN_ PAYLOAD_VLD	Input	Indicates that the data received is in the payload of the packet
SRC_RDY	Input	Source ready signal
DATA_IN[7:0]	Input	Data in
HEADER_ DATA[119:0]	Input	Header data vector
HEADER_DATA_ VLD	Input	Valid header data

3.6.3. Outputs

Table 3.10 describes the output signals of the Rule Match module.

Table 3.10.: Rule Match module Outputs

Signals	Direction	Description
SID[15:0]	Output	The Sid of the matched rule
SID_VLD	Output	Sid valid signal

3.6.4. Subsystems

- Transform No Case: Transforms the *Data In* uppercase characters to lowercase.
- Fan-Out Tree: A register tree that used to broadcast its input Data to 16 registered copies in order to achieve the desired timing clock.
- Header Compare: Compares the header vector with predefined values and outputs a registered signal for each matched value.

- Content Compare: Compares the Data from the shift registers with predefined content text. This module will be described in detail in section 3.7
- Inverse Priority Encoder: A inverse priority encoder that encodes in 3 stages 4096 signals to 12bit. The Encoder is described in section 3.8
- Decoder: Decodes the 12 bit address of the Sid Rom back to 4096 signals that indicate which rule register was sent.
- Sid Rom: A single port Rom has been used to store the sid values of the rules.
- Is New Frame: Is a register with added simple logic. The register's output is asserted high only when a new Frame Valid signal is raised. This is implemented by setting the input of the register as (*Frame_vld* **and not** *prv_Frame_vld*).

3.7. Content Compare

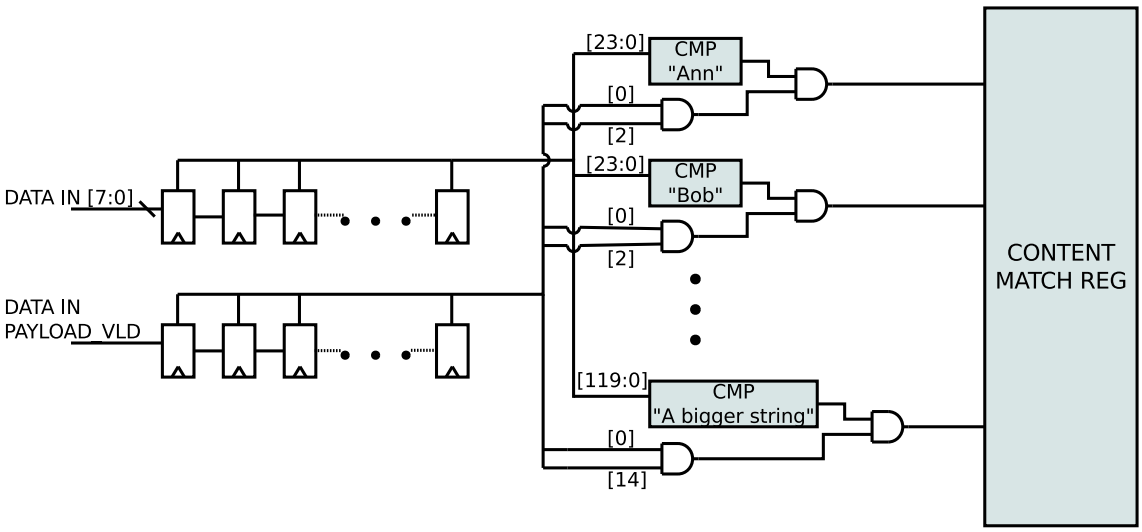


Figure 3.11.: Content Compare example datapath

3.7.1. Description

The block diagram of Figure 3.11 is an example for text match inside the payload of the received packet. Snort has also a modifier keyword "rawbytes" that allows rules to look at the raw packet data, ignoring any decoding that was done (see Snort Manual Rawbytes), this is not currently supported by our system.

To check for specific strings we use parallel comparators. This is **and**ed with a logic that controls if the shift registered data is in the payload. We want the first and the last byte of comparison to be inside the payload.

The shift register pairs of *Data in* and *Data in payload* feed 15 comparators and **and** gates respectively. Our system supports up to 256 shift register pairs for each fan-out tree, that means 3840 comparators.

3.8. Inverse Priority Encoder

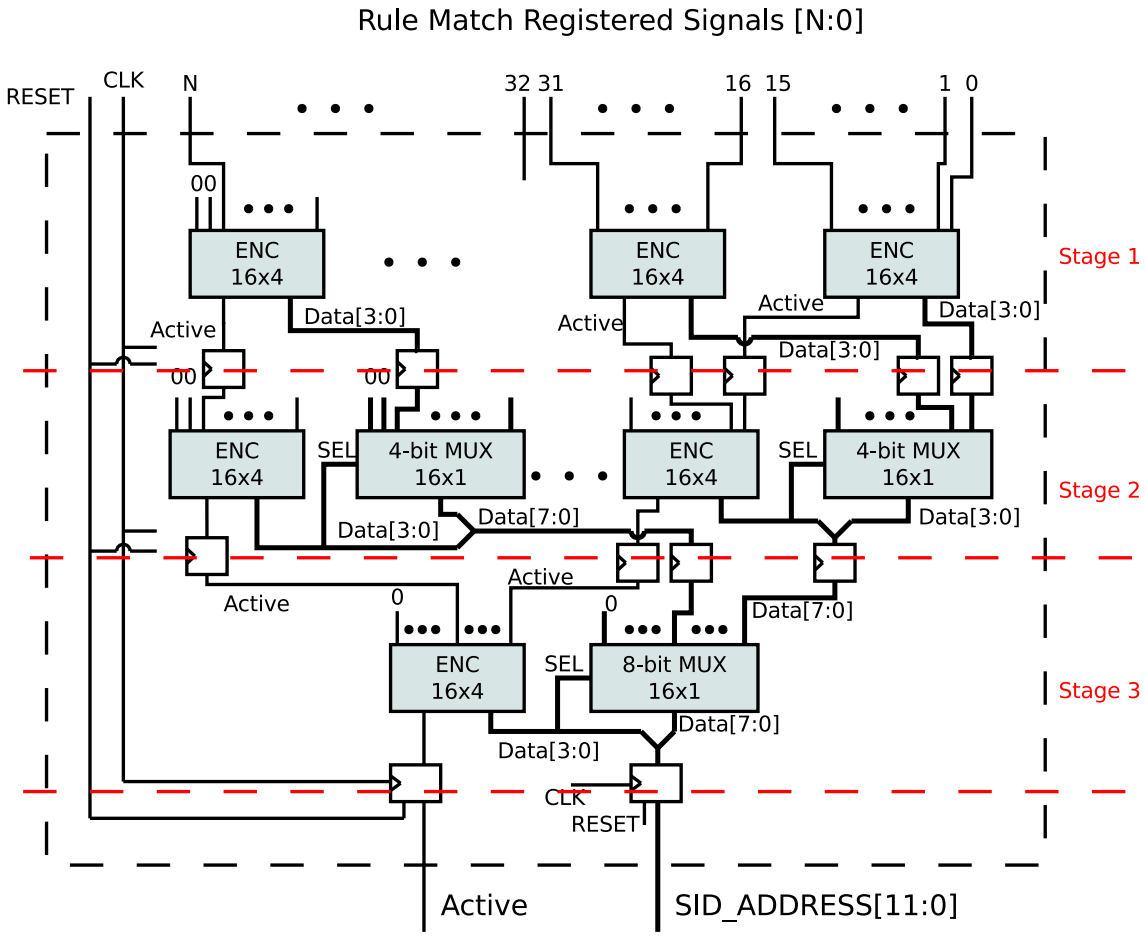


Figure 3.12.: Inverse Priority Encoder Datapath

3.8.1. Description

The Inverse Priority Encoder is responsible for taking the registered rule match signals and encode them in order to find the smallest active rule id. We used a 3-Stage pipeline encoder with 16-to-4 encoders and 16-to-1 variable width multiplexers instead of using a very large single cycle 4096-to-12 priority encoder (this would be too slow and probably not meet timing constraints). Figure 3.12 shows the datapath of our encoder.

To avoid making the datagram illegible we assume that clock signal is routed to the registers and reset to the registers and multiplexers. This encoder supports up to 4096 input signals and the output result (12 bit) is active after three cycles.

In the first level we have only encoders which encode 16 sequential input bits. When at least one input bit is high '1' the *Active* encoder output is asserted high, otherwise it is set to zero. In this level we can have at most 256 encoders.

In the second and third level a multiplexer is paired to every encoder. Each encoder in this level encodes (in groups of 16) the *Active* output signals of the previous level. The paired multiplexer uses the data output of the encoder as a *select* vector and selects among the corresponding 16 encoded data outputs of the previous level. In second level at most 16 encoders and 4-bit 16-to-1 multiplexers are used, while in the last third level only one encoder and one 8-bit 16-to-1 multiplexer is used.

In every stage when the input signal are not equal to multiple of 16, the remaining input signals of the last encoder and multiplexer will be padded with zeros.

3.8.2. Input Signals

Table 3.11 shows the input signals of the encoder.

Table 3.11.: Inverse Priority Encoder Inputs

Signal	Direction	Description
CLK	Input	Clock signal
RESET	Input	Reset signal
RULE_MATCH[N:0]	Input	Rule signals that indicate a match

3.8.3. Output Signals

Table 3.12 shows the output signals of the encoder.

Table 3.12.: Inverse Priority Encoder Outputs

Signal	Direction	Description
Active	Output	Active signal that indicates when the output address is valid
SID_ADDRESS[11:0]	Output	ID address of the active rule

3.9. Packet Transmitter

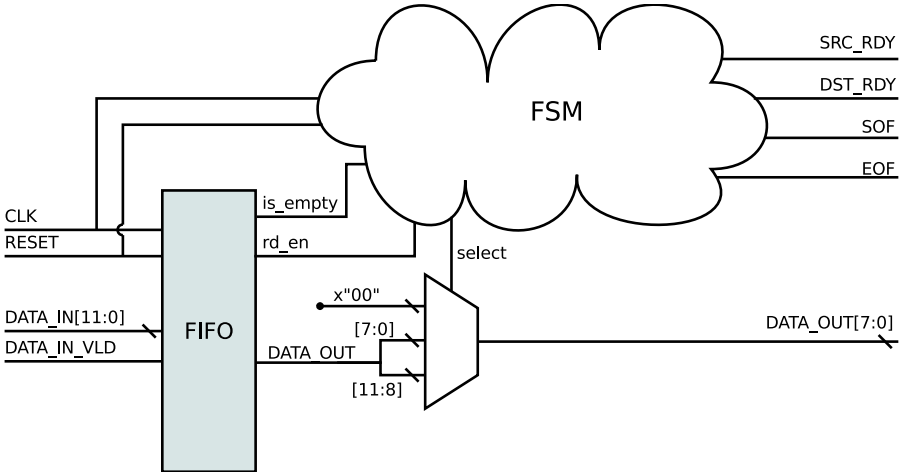


Figure 3.13.: Datapath of Packet Transmitter

3.9.1. Description

Packet Transmitter is used to transmit the Sid value of the matched rules. Figure 3.13 shows the data path of the transmitter. In order support more than one incoming Sid values while at the same time a packet is transmitted, we use a FIFO memory to store the data to be sent. This memory has 12bit width and 1024 write depth. The inputs of the transmitter are the outputs from Rule Match module (*SID, SID_VLD*) and are connected to *Data in* and *Data in vld*.

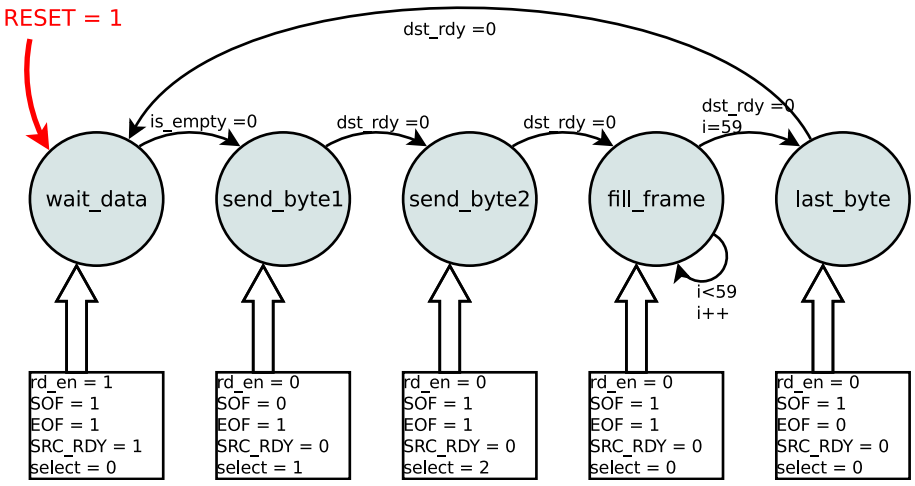


Figure 3.14.: FSM of Packet Transmitter

A FSM is used to control an 8bit 3-to-1 multiplexer, signals *SRC_RDY*, *SOF* and *EOF*. Figure 3.14 represents the function of the FSM. When reset or FIFO is empty

the FSM is in its initial state *wait_data*. Otherwise the *read_en* signal is asserted requesting data from the memory. The multiplexer selects the first byte of *data out* to be send and *SOF* & *SRC_RDY* signals are tied low so that the Ethernet MAC wrapper begins the transmission of a new valid frame. In the next cycle if *DST_RDY* is zero we continue the transmission with the second byte of data. When and the second byte has been sent we pad the remaining packet with zero bytes and in the last byte transmitted *EOF* is set zero, so that the wrapper indicates the end of packet.

3.9.2. Input Signals

Table 3.13 shows the input signals of packet transmitter.

Table 3.13.: Packet Transmitter Inputs

Signals	Direction	Description
CLK	Input	Clock signal
RESET	Input	Reset signal
DATA_IN[11:0]	Input	Data in from rule match module (SID data)
DATA_IN_VLD	Input	Data in valid signal from rule match module (SID_VLD)
DST_RDY	Input	Transmission Destination ready signal from Ethernet Mac wrapper

3.9.3. Output Signals

Table 3.14 shows the output signals of packet transmitter.

Table 3.14.: Packet Transmitter Outputs

Signals	Direction	Description
SRC_RDY	Output	Source ready signal to transmitter wrapper
SOF	Output	Start of frame indicator
EOF	Output	End of frame indicator
DATA_OUT[7:0]	Output	Data out to Ethernet Mac wrapper transmitter

3.10. Automated Code Generation

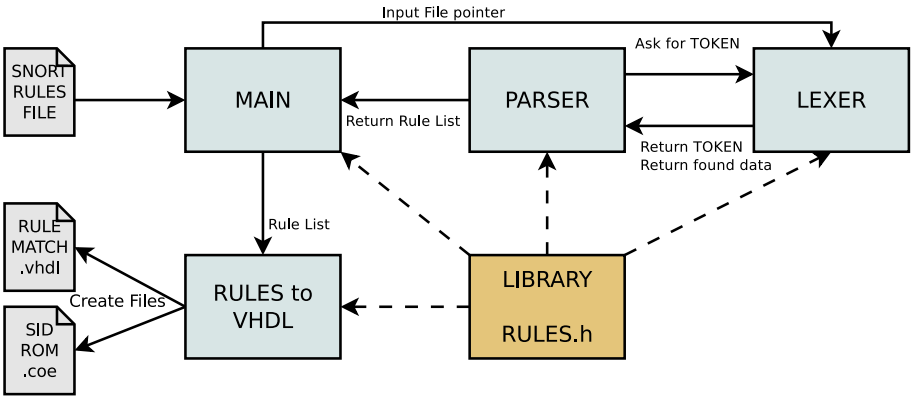


Figure 3.15.: Automated Code Generation system overview

3.10.1. Description

The Automated Code Generator is a program written in C, Flex and Bison that creates the *vhdl* file for Rule Match module (see section 3.6) and a "*SID_ROM.coe*" file for Sid ROM initialization. The input ruleset passes through a Lexical analyzer and grammar parser, to create a Rule List structure. Figure 3.15 is a general overview of the code generator while Figure 3.16 represents the schematic structure of the Rules List.

When the code generator is called with a ruleset file as parameter it sets the lexer to read from the file in other case from the standard input. Thereafter we call the parser which purpose is to validate syntax of the input using the tokens that the lexer returns. The parser also combines the data returned from the lexer to create a valid rules list. The parser and lexer support the whole rules syntax as described in the Snort Manual. Options that are not supported by our hardware system are parsed but their data is never used.

RULES Library is a very important library used in all the code files. This library contains the rule structure and basic functions, such as list manipulation and extraction functions. The lexer creates ip, port or option nodes with the data matched and returns the token found. Then the parser assumes to join this nodes into lists and create rules. For example the following rule would create the tokens and return nodes as described below.

```
# A simple comment row
alert tcp any any -> 192.168.1.1 [80, 443] (sid:123; content:"Hello World"; nocase;)
```

#	Token	Returned Value
1	COMMENT	-
2	ACTION	ALERT
3	TCP	-
4	ANY	-
5	ANY	-
6	"->"	-
7	IP_ADDR	headerNode with data = ipNode, address = x"C0A80101" and cidr = 32
8	"["	-
9	PORT	headerNode with data = portNode, port1 & port2 = 80
10	PORT	headerNode with data = portNode, port1 & port2 = 443
11	"]"	-
12	"("	-
13	OPTION	optionNode with data = int, sid = 123
14	OPTION	optionNode with data = optContent, string = "Hello World", len = 11
15	NOCASE	-
16	")"	-

The parser with the above example would check if the tokens are placed in the right order and create a rule from the extracted data. Append the two ports to create a list, set the nocase value of the second option "true" and append the two options to a list.

The Main function calls the "rules2vhdl" function and passes the rule list. This function will at first create unique lists of ip addresses, port addresses, content options[1], itype and ip_proto from the given rules list and then call functions for each list created to write in the output file the generated vhdl code.

Back again to the main this time it calls "rules2vhdl_sid" function to create a coe file with the sid values of the rules.

[1]The unique content list is also sorted so that the created but unused shift registers will be trimmed by the vhdl synthesis tool efficiently.

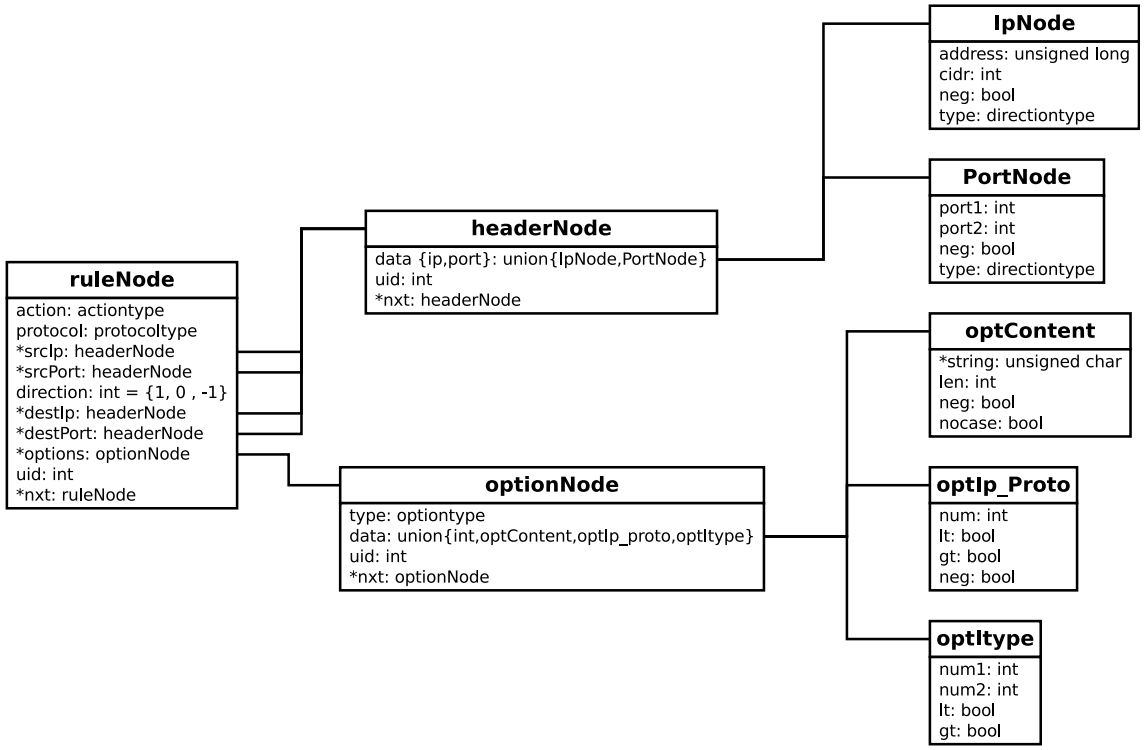


Figure 3.16.: Rule structures

In order to change the default variables of Snort for:

- $HOME_NET
- $EXTERNAL_NET
- $DNS_SERVERS
- $SMTP_SERVERS
- $HTTP_SERVERS
- ...

we have to edit the rules.c file, set the desired values and recompile the code.

3.10.2. Inputs - Outputs

The input file to the code generator is a Snort compatible ".rules" file or if no file is given the STD IN.

The output files are the rule match module and a initialization ".coe" file for the Sid Rom.

4. Results

4.1. Introduction

This chapter presents the implementation results for a specific rule configuration, the performance of the system, how this was calculated and the evaluation method of measuring the results accuracy.

4.2. Rule Configuration

The configuration of the symbolic variables used in the snort rules (e.g $HOME_NET, $EXTERNAL_NET) were:

- $HOME_NET: set to 147.27.3.0/24 (cidr is 24, that means we are interested only in the first 24 bits of the IP address)
- $EXTERNAL_NET: set to !HOME_NET
- $HTTP_SERVERS: set to 147.27.4.0
- $TELNET_SERVERS: set to 147.27.5.0
- $SMTP_SERVERS: set to 147.27.6.0
- $DNS_SERVERS: set to 147.27.7.0
- $SQL_SERVERS: set to 147.27.8.0
- $SSH_SERVERS: set to 147.27.9.0
- $FTP_SERVERS: set to 147.27.10.0
- $SIP_SERVERS: set to 147.27.11.0
- $AIM_SERVERS: set to the same values used by Snort. These are IP addresses 64.12.24.00/23, 64.12.28.00/23, 64.12.161.0/24, 64.12.163.0/24, 64.12.200.00/24, 205.188.3.0/24, 205.188.5.0/24, 205.188.7.0/24, 205.188.9.0/24, 205.188.153.0/24, 205.188.179.0/24 and 205.188.248.0/24
- $HTTP_PORTS: set to 10 ports. These are ports 80, 81, 8000, 8008, 8014, 8028, 8080, 8088, 8800 and 8888
- $SHELLCODE_PORTS: set to !80

- $ORACLE_PORTS: set to 1024
- $SSH_PORTS: set to 22
- $FTP_PORTS: set to ports 21, 2100 and 3535
- $SIP_PORTS: set to ports 5060, 5061 and 5600

A sub-set of 750 rules of the whole Snort-2.9[16] ruleset was used in this work.

To represent the statistical data of the rules with the above configuration we use histograms for the number of ports used by rules and the number of searching contents in rules (Figure 4.1). As we can see, most of the rules search for only one port or for ten, this occurs because of the variable "$HTTP_PORTS" which has ten ports. Rules have from zero up to six searching contents. Almost all the rules search on two ips ($EXTERNAL_NET and $HOME_NET).

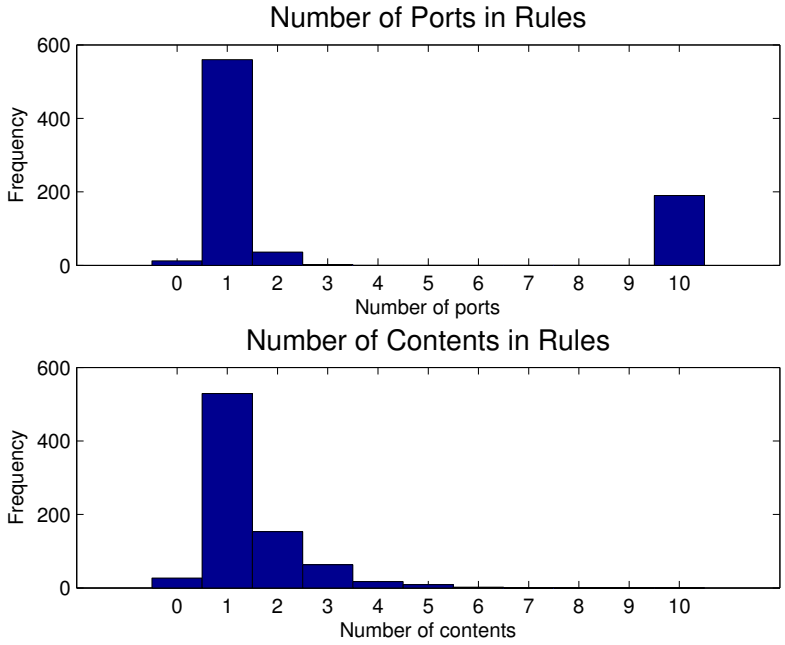


Figure 4.1.: Statistical data of rules

Ports have fixed 16bit length, while the ip addresses length is variable which depends on the cidr variable. On the other hand the searched content has big variance in its length (146.7). In our implemented ruleset, contents have a size between 1 and 109 bytes. Figure 4.2 shows a histogram of the 1062 content's size with a summary of 12803 characters, used by the implemented rules. Table 4.1 presents the statistical infotrmation of the contents.

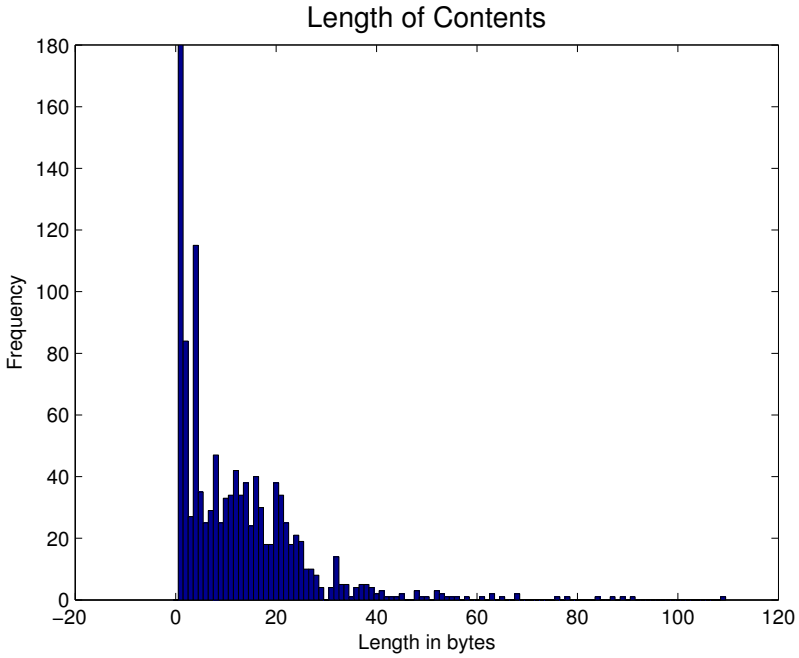


Figure 4.2.: Size of contents in bytes

Table 4.1.: Content Statistics

Num. of Contents	1062
Min Length	1
Max Length	109
Mean Length	12.1
Median Length	9
Variance	146.7
Sum of Chars	12803

4.3. FPGA Implementation

The design was implemented in a Genesys™ Virtex-5 FPGA Development Board[1]. This board has aVirtex-5 LX50T FPGA with 28800 6-input LUTs/FFs. Due to the limited capacity of our FPGA, only a fraction of the default Snort rule set could be implemented in the development board. We implemented a rule set of 750 rules and utilized 28018 LUT Flip Flop pairs out of the available 28800 (97,3%) and 99% of the slices. Table 4.2 shows the LUT usage of individual components.

To optimize the design we used the following parameters, *optimization goal* to “area”, *register duplication* “true” and *remove equivalent register* to “false”. Flag “remove

Table 4.2.: LUT Utilization

Component	#LUT
Wrapper	409
Decoder	291
Match Mod.	27209
Packet Trans.	109

equivalent registers" is important to be set "false" in order the "place and route" process to implement the design.

With a 2.1GHz processor the compilation time of the design was:

- ≃15min for Synthesize with XST
- ≃13min for mapping
- ≃25min for Place and Route

4.4. Performance

The implemented design is a real time system which operates on Gig-Ethernet speed with a 125Mhz clock. It does not delay the receiving frame at no point, this means that the throughput is depended on the ability of the transmitter to send data to our system.

Taking into account the minimum interframe gap (12 bytes) -as specified by IEEE Std 802.3- for full duplex systems , preamble (7 bytes), start of frame delimiter (1 byte) and mac layer (18 bytes), we calculated the data transfer speed of an ideal transmitter as follows. The IP packet size varies from 46 to 1500 bytes, with 38 Ethernet overhead bytes as previously mentioned.

$$tr\_speed = \frac{IP\_packet\_size}{38+IP\_packet\_size} 1000Mbit/sec$$

Figure 4.3 shows the maximum Ethernet Payload data rate in relation to the IP packet size sent by an ideal Gigabit Ethernet transmitter.

All rules are matched at wire speed, generating an Ethernet frame of 46 bytes containing the sid of the matched rule. In the case of multiple match the sid's of the rules matched are transferred to the transmitter fifo-memory at a rate of one sid every four cycles. This means that in the worst case of a multiple match at the end of an IP frame with another frame following by a 38 byte gap, it will manage to store 9 sid values to the transmitter fifo. Because of the inverse priority encoder only the 9 sid's with the smallest values[1] will be transmitted.

[1]We have to take into account that when writing rules, the most important ones should have lower values

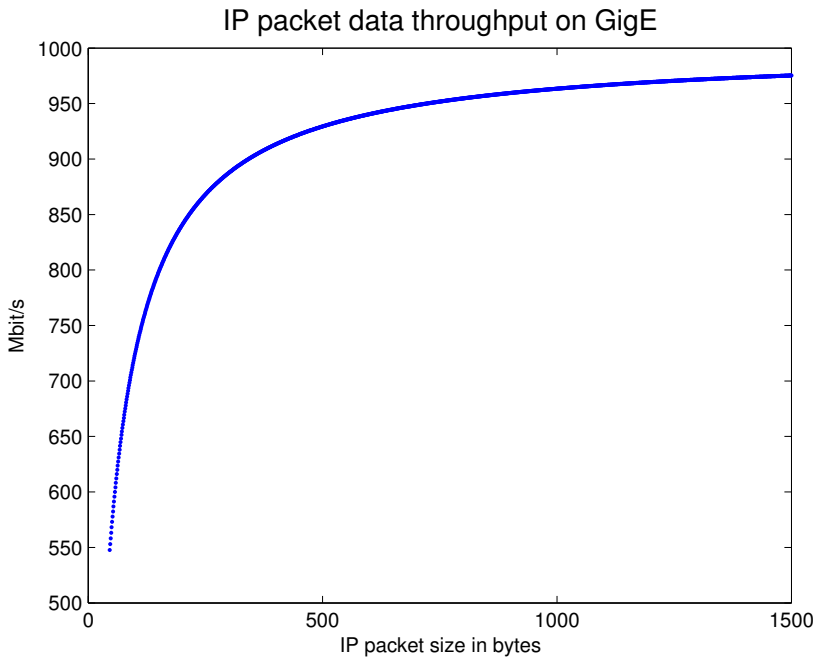


Figure 4.3.: Data Rate of an IP packet on Gigabit Ethernet

4.5. Evaluation

The evaluation of the results was done by hand, creating packets that met the given rules using an open source program Packeth-1.6 and a network analyzer Wireshark to receive the sent packet from the FPGA board. The (16 bit) received "sid" was compared to the actual "sid" of the rule which the created packet met, with 100% accurate results.

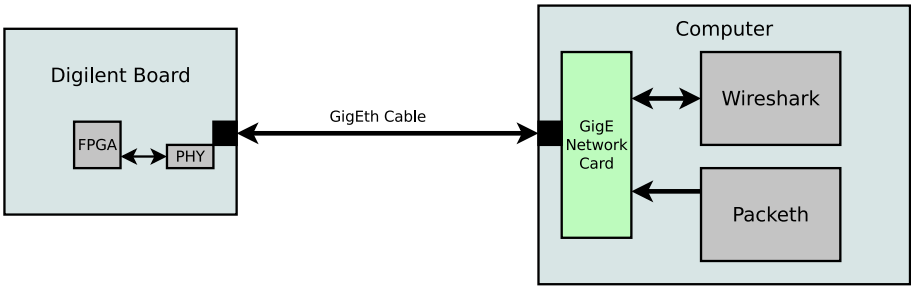


Figure 4.4.: Setup used to test the design

5. Conclusion

In this diploma thesis, we have presented a complete DPI system that works on GigE wire speed compatible with Snort rules. A physical interface and a MAC wrapper are responsible for the data exchange on network. Packet receiver decodes "network" and "transport" layers. A pattern matching module that searches strings in the payload of a packet and specific header values has been presented.

The background of Deep packet inspection systems, why they are used and the advantages - disadvantages of software and hardware implementations was described in chapter 2.

We presented on chapter 3 a DPI-on-a-chip architecture with network interface working on wire speed 1Gbps.

Results have been presented in chapter 4, for a rule-set of 750 rules implemented in a V5-LX50T utilizing the 97,3% of the available LUTs.

Future Work

There is a variety of opportunities for future work. Our architecture uses parallel comparators to match patterns on the incoming data, this has a good performance in speed, but on the other hand a large area cost. Better algorithms could be used, such as the optimized discrete comparators I. Sourdis proposed in his work[17].

Furthermore, another physical medium could be used such as the XGMII and extend the input data vector to 32 bits. If the design achieves a clock rate of 152,25MHz (currently 125MHz), it would have a throughput of 5Gbps.

The compatible Snort ruleset could be extended to support more options such as:

- **rawbytes**: The rawbytes keyword allows rules to look at the raw packet data, ignoring any decoding that was done. This acts as a modifier to the *content* keyword.
- **depth**: The depth keyword allows to specify how far into a packet should be searched for a specified pattern.
- **pcre**: The pcre keyword allows rules to be written using perl compatible regular expressions.

Last but not least, FPGA - Power PC co-design could be used to manage session layer options (e.g. `flow:to_server,established;`) or to join fragmented packets.

A. Snort Rules

A.1. Introduction

Like viruses, most intruder activity has some sort of signature. Snort's detection system is based on rules which in turn are based on intruder signatures. A rule may be used to generate an alert message, log a message or drop the data packet. Snort rules are written in an easy to understand syntax.

This chapter provides information about the structure of a rule, a short description of the basic functions and what our system supports.

A.2. Structure of a Rule

All the rules have the following structure:

Action Protocol Address Port Direction Address Port (Options)

A Snort rule has two logical parts the rule *header* and rule *options*.

The rule header contains information about what *action* should be taken and on what *protocol* the rule applies. It contains also the source and destination *address*. In case of TCP or UDP protocol, the *port* determines the source and destination ports of a packet on which the rule is applied.

The part of the rule enclosed by the parentheses is the options part. This part usually contains an alert message, a *sid* number that uniquely identifies Snort rules and more specific information about the signature of suspicious packets.

An example of a (very bad) rule, that alerts on every ip packet sent or received:

alert ip any any -> any any (msg: "IP Packet detected";)

More information about Snort rules can be found in the Snort Manual[3].

A.3. Supported Rules

The rule description language (as defined by Snort) can be parsed by our code generator tool, but only a subset is functionally supported by our system.

More specifically, the rule *options* that functionally work in our design are:

- **sid**: The sid keyword is used to add a "Snort ID" to rules. The only argument to this keyword is a number. The sid value is used by our system as the output value of the matched rule. Syntax: `sid:<snort rules id>;`
- **ip_proto**: The ip_proto keyword is used to determine the protocol number that follows an IP header. The keyword requires a protocol number as argument. Protocol numbers are defined in RFC 1700 at http://www.rfc-editor.org/rfc/rfc1700.txt. Syntax: `ip_proto:[!|>|<] <number>;`
- **itype**: The ICMP header comes after the IP header and contains a type field. The itype keyword is used to detect attacks that use the type field in the ICMP packet header. Syntax: `itype:min<>max;` *or* `itype:[<|>]<number>;`
- **content**: One of the most important features of Snort is the ability to find patterns inside a packet. The content keyword is used to find intruder signatures in the packet. The pattern may be an ASCII string or a binary data in the form of hexadecimal data. Syntax: `content:[!]"<content string>";`
- **nocase**: The nocase keyword is used in combination with the content keyword. It has no arguments. Its only purpose is to make a case insensitive search of a pattern within the data part of a packet. Syntax: `nocase;`

One or more *options* may be used simultaneously in a rule. The multiple *options* form a logical AND. Rules can contain more than one searching *contents* and the *nocase* modifier is applied to each content separately.

All the *header* rule parts (as defined by Snort) are supported:

- **Action**: This part shows what action will be taken when rule conditions are met. The keywords used are: **pass**, **log**, **alert**, **activate** and **dynamic**.
- **Protocol**: The protocol part shows on which type of packet the rule will be applied. Supported protocols are **IP**, **ICMP**, **TCP** and **UDP**.
- **Address**: The two address parts are used to check the source and destination address of the packet. It can be a single IP, a network address or the keyword *any* if the rule applies to all the addresses. (e.g. 192.168.1.0/24 or 147.27.3.1 or a list [147.27.3.1 192.168.1.0/24]).
- **Port**: Port number is meaningful only for TCP and UDP protocols. The two ports are the source and destination port of the packet. It can be a single port (e.g. 80) a port range (e.g. 1024:1050) a list (e.g [80 1024:1050]) or the keyword *any*.
- **Direction**: Symbol that indicates the direction of source and destination addresses and ports (e.g. *src* -> *dest*, *dest* <- *src* and *src_dest* <> *src_dest*).

B. Packet Header Formats

B.1. Overview

Snort rules use the protocol type field to distinguish among different protocols. Various header parts in packets are used to identify the protocol type used in a packet. Furthermore, rule options such as (ip_proto and itype) can test some of the header fields. The purpose of this appendix is to explain the headers of different protocols used in Snort rules. The packet header formats are very important to understand the function of the packet receiver and decoder of our design and also useful for writing effective Snort rules.

B.2. IP Packet Header

The default IPv4 header consists of 20 bytes. An option part may be presented after the first 20 bytes of the header and can be up to 40 bytes long. Figure B.1 presents the IP header structure.

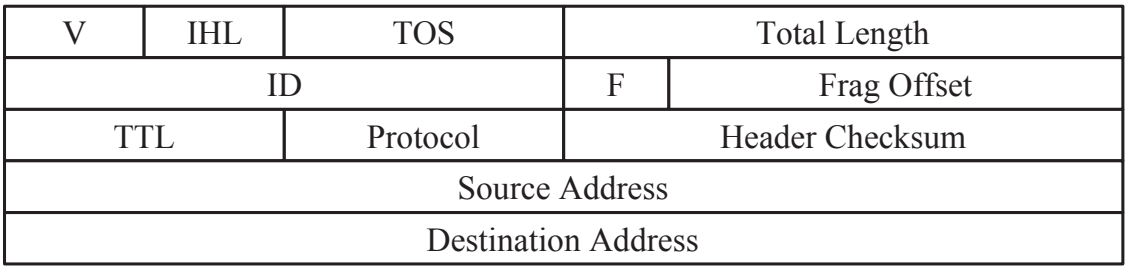


Figure B.1.: IP header

More information can be found in RFC 791 at http://www.ietf.org/rfc/rfc791.txt. Table B.1 is a brief explanation of the IP packet header fields.

Table B.1.: IP Packet Header Fields

Field	Description
V	Version number. The value is 4 for IPv4. Four bits are used for this part
IHL	This field shows length of IP packet header. This is used to find out if the options part is present after the basic header. Four bits are used for IHL and it shows length in 32-bit word length. The value of this field for a basic 20-bytes header is 5.
TOS	This field shows type of service used for this packet. It is 8 bits in length.
Total Length	This field shows the length of the IP packet, including the data part. It is 16 bits long.
ID	This field packet identification number. This part is 16 bits long.
F	This part is three bits long and it shows different flags used in the IP header.
Frag Offset	This part is thirteen bits long and it shows fragment offset in case an IP packet is fragmented.
TTL	This is time to live value. It is eight bits long.
Protocol	This part shows transport layer protocol number. It is eight bits long.
Header Checksum	This part shows header checksum, which is used to detect any error in the IP header. This part is sixteen bits long.
Source Address	This is the 32 bit long source IP address.
Destination Address	This is the 32 bit long destination IP address.

B.3. ICMP Packet Header

Internet Control Message Protocol is a network layer protocol that uses the basic support of IP as if it were a higher level protocol. Figure B.2 shows the basic structure of ICMP header. Note that depending upon type of ICMP packet, this basic header is followed by different parts.

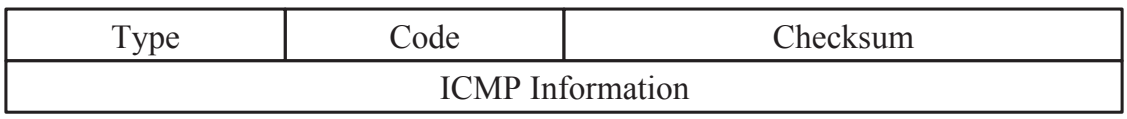


Figure B.2.: ICMP header

Table B.2 provides an explanation of the basic ICMP header fields.

Table B.2.: ICMP Packet Header Fields

Field	Description
Type	This part is 8 bits long and shows the type of ICMP packet.
Code	This part is also 8 bits long and shows the sub-type or code number used for the packet.
Checksum	This part is 16 bits long and is used to detect any errors in the ICMP packet.

The ICMP information field is variable depending on the ICMP type field. Figure B.3 shows an ICMP ECHO REQUEST used for the ping command.

Type	Code	Checksum
Identifier		Sequence Number

Figure B.3.: ICMP Echo Request Packet

ICMP protocol details are explained in RFC 792 at http://www.ietf.org/rfc/rfc792.txt.

B.4. TCP Packet Header

Transmission Control Protocol is a transport layer protocol, encapsulated in a IP packet. TCP packet header is described in detail in RFC 793, available on the web at http://www.ietf.org/rfc/rfc793.txt. Figure B.4 shows the TCP header structure.

Source Port			Destination Port
Sequence Number			
Acknowledgement Number			
Offset	Reserved	Flags	Window
Checksum			Urgent Pointer
Options and Padding			

Figure B.4.: TCP header

The basic TCP header length is 20 bytes. A variable options field may be presented. Header parts are explained in Table B.3.

Table B.3.: TCP Packet Header Fields

Field	Description
Source Port	This part is 16 bits long and shows source port number.
Destination Port	This is a 16-bit long field and shows the destination port number.
Sequence Number	This is the sequence number for the TCP packet. It is 32 bits long. It shows the sequence number of the first data octet in the packet. How- ever if SYN bit is set, this number shows the initial sequence number.
Acknowledgement Number	This number is used for acknowledging packets. It is 32 bits long. This number shows the sequence number of the octet that the sender is expecting.
Offset	This is a 4- bit field and shows the length of the TCP header. Length is measured in 32-bit numbers.
Reserved	Six bits are reserved.
Flags or Control bits	The flags are six bits in length and are used for control purposes. These bits are URG, ACK, PSH, RST, SYN and FIN. A value of 1 in any bit place indicates the flag is set.
Window	This is 16 bits long and is used to tell the other side about the length of TCP window size.
Checksum	This is a checksum for TCP header and data. Its 16 bits long.
Urgent Pointer	This field is used only when the URG flag is set. It is 16 bits long.
Options	This part has variable length.

B.5. UDP Packet header

User Datagram Protocol is transmission layer protocol encapsulated in an IP protocol. The UDP header is very simple, it has four field as shown in Figure B.5.

Figure B.5.: UDP packet header

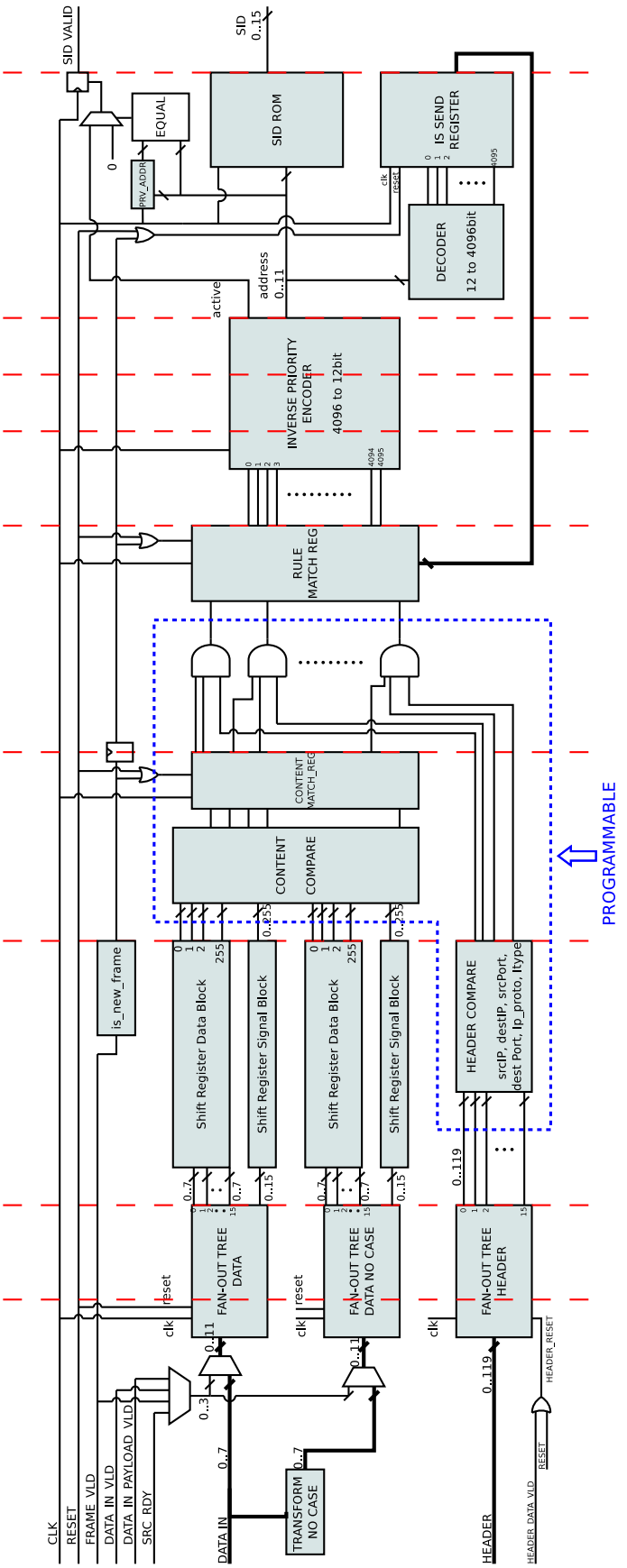
Table B.4 describes the header fields of a UDP packet.

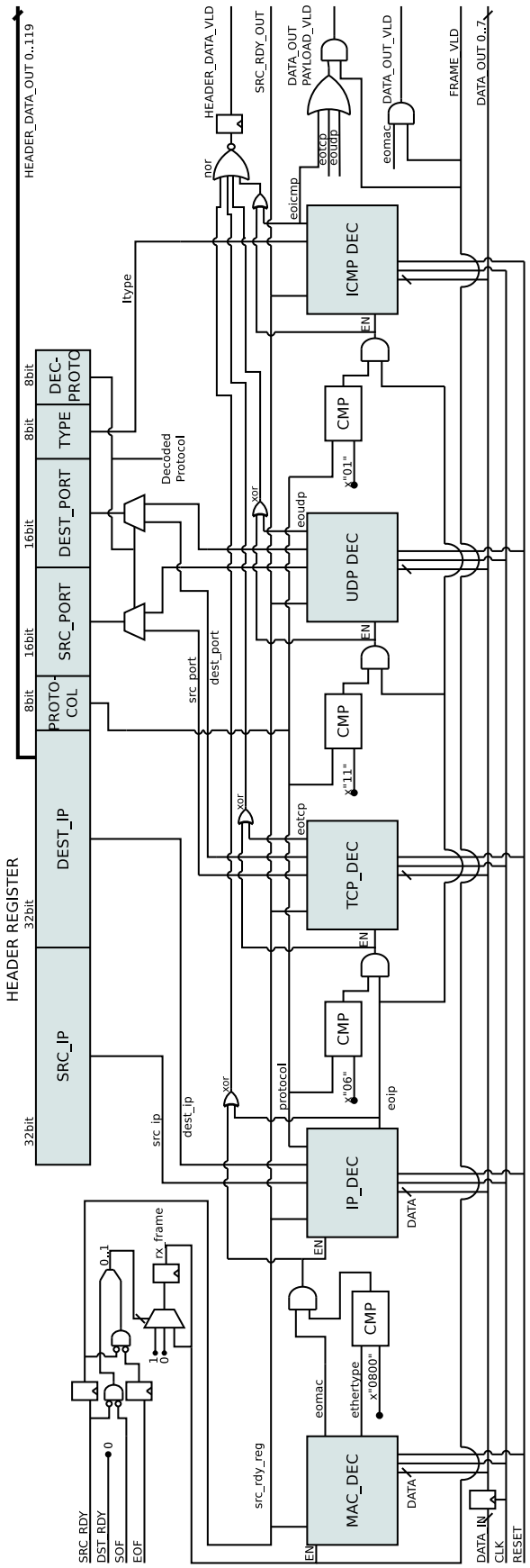
Table B.4.: UDP Header Packet Fields

Field	Description
Source Port	This part is 16 bits long and shows source port number.
Destination Port	This is a 16-bit long field and shows the destination port number.
Length	A 16-bit long field that specifies the length in bytes of the entire datagram: header and data.
Checksum	This part is 16 bits long and is an optional checksum, if not used the field must be all zeros.

More information about UDP packets can be found in RCF 768, available on the web at http://www.ietf.org/rfc/rfc768.txt.

C. Detailed Datapaths


SID VALID
SID 0..15
EQUAL
SID ROM
IS SEND REGISTER
PRV_ADDR
DECODER 12 to 4096bit
address 0..11
active
INVERSE PRIORITY ENCODER 4096 to 12bit
RULE MATCH REG
CONTENT MATCH_REG
CONTENT COMPARE
PROGRAMMABLE
is_new_frame
Shift Register Data Block
Shift Register Signal Block
HEADER COMPARE srcIP, destIP, srcPort, dest Port, Ip_proto, Itype
FAN-OUT TREE DATA
FAN-OUT TREE DATA NO CASE
FAN-OUT TREE HEADER
TRANSFORM NO CASE
CLK
RESET
FRAME VLD
DATA IN VLD
DATA IN PAYLOAD VLD
SRC RDY
DATA IN
HEADER
HEADER_RESET
HEADER_DATA VLD


HEADER_DATA_OUT 0..119
HEADER_DATA_VLD
SRC_RDY_OUT
DATA_OUT PAYLOAD_VLD
DATA_OUT_VLD
FRAME_VLD
DATA_OUT 0..7
HEADER REGISTER
DEST_IP
32bit
SRC_IP
32bit
PROTO COL
8bit
SRC_PORT
16bit
DEST_PORT
16bit
TYPE
8bit
DEC-PROTO
8bit
ltype
Decoded Protocol
src_port
dest_port
src_ip
dest_ip
protocol
eoip
eotcp
eoudp
eoicmp
eomac
ethertype
x"0800"
x"06"
x"11"
x"01"
nor
xor
CMP
EN
ICMP DEC
UDP DEC
TCP_DEC
IP_DEC
MAC_DEC
DATA
rx_frame
src_rdy_reg
SRC_RDY
DST_RDY
SOF
EOF
DATA_IN
CLK
RESET

Bibliography

[1] Digilent Genesys Development Board. http://www.digilentinc.com.

[2] L7-filter. Application Layer Packet Classifier. http://l7- filter.sourceforge.net.

[3] Smort online manual. http://manual.snort.org/.

[4] IANA Considerations and IETF Protocol Usage for IEEE 802 Parameters. 2008.

[5] User Guide - Virtex-5 FPGA Embedded Tri-Mode Ethernet MAC. *ReVision*, 194, 2011.

[6] T. Abuhmed, A. Mohaisen, and D. Nyang. A Survey on Deep Packet Inspection for Intrusion Detection Systems.

[7] A. V Aho and M. J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[8] R. S. Boyer and J. S. Moore. A Fast String Searching Algorithm.

[9] C. R. Clark, C. D. Ulmer, and D. E. Schimmel. An FPGA-based Network Intrusion Detection System with On-chip Network Interfaces. *International Journal of Electronics*, 93(6):403–420, 2006.

[10] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. *IEEE Micro*, 24(1):52–61, 2004.

[11] S. Dharmapurikar, J. W. Lockwood, et al. Fast and scalable pattern matching for network intrusion detection systems. *IEEE Journal on Selected Areas in Communications*, 24(10):1781, 2006.

[12] V. Dimopoulos, I. Papaefstathiou, and D. Pnevmatikatos. A memory-efficient reconfigurable aho-corasick fsm implementation for intrusion detection systems. In *Embedded Computer Systems: Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007. International Conference on*, pages 186 –193, july 2007.

[13] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[14] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.

[15] R. U. Rehman. *Intrusion detection systems with Snort: advanced IDS techniques using Snort, Apache, MySQL, PHP, and ACID*. Prentice Hall, 2003.

[16] M. Roesch et al. Snort-lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration*, pages 229–238. Seattle, Washington, 1999.

[17] I. Sourdis and D. Pnevmatikatos. Fast, Large-Scale String Match for a 10Gbps Fpga-Based Network Intrusion. *FPL*, 2003:880–889, 2003.

[18] I. Sourdis and D. Pnevmatikatos. Pre-decoded cams for efficient and high-speed nids pattern matching. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 258–267. IEEE, 2004.

[19] F. Yu and R. H. Katz. Efficient multi-match packet classification with tcam. In *High Performance Interconnects, 2004. Proceedings. 12th Annual IEEE Symposium on*, pages 28–34. IEEE, 2004.