

# DESIGN AND IMPLEMENTATION OF INTRUSION DETECTION SYSTEM WITH FPGA

## A DISSERTATION

*Submitted in partial fulfillment of the  
requirements for the award of the degree*

*of*

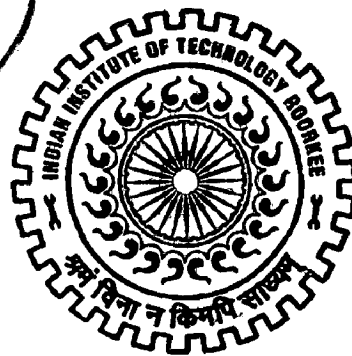
MASTER OF TECHNOLOGY

*in*

COMPUTER SCIENCE AND ENGINEERING

*By*

**ARUN KUMAR**



DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING  
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE  
ROORKEE - 247 667 (INDIA)

JUNE, 2008

## CANDIDATE'S DECLARATION

---

I hereby declare that the work, which is being presented in the dissertation entitled “**DESIGN AND IMPLEMENTATION OF INTRUSION DETECTION SYSTEM WITH FPGA**” towards the partial fulfillment of the requirement for the award of the degree of **Master of Technology in Computer Science and Engineering** submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee (India) is an authentic record of my own work carried out during the period from June 2007 to June 2008, under the guidance of **Dr. R. C. Joshi, Professor, Department of Electronics and Computer Engineering, IIT Roorkee.**

I have not submitted the matter embodied in this dissertation for the award of any other degree or diploma.

Date: 25 Jun 08

Place: Roorkee

  
(ARUN KUMAR)

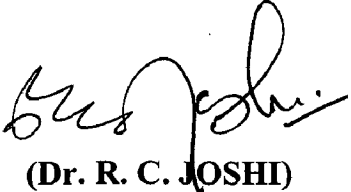
---

## CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

Date: 25.6.08

Place: Roorkee

  
(Dr. R. C. JOSHI)

Professor

Department of Electronics and Computer Engineering

IIT Roorkee – 247 667

## ACKNOWLEDGEMENTS

---

I would like to take this opportunity to extend my heartfelt gratitude to my guide and mentor **Dr. R. C. Joshi**, Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, for his trust in my work, his able guidance, regular source of encouragement and assistance throughout this dissertation work. I would state that the dissertation work would not have been in the present shape without his inspirational support and I consider myself fortunate to have done my dissertation under him.

I also extend my sincere thanks to **Mr Rajeev Goswami** for helping me and extending his full support while working in Information Security Lab.



**ARUN KUMAR**

## ABSTRACT

---

In this work design and implementation of Intrusion Detection System (IDS) with Field Programmable Gate Array (FPGA) is presented. Today's network security systems require high-performance as well as good functionality with the growing speed of the internet. But most of the software-based Network Intrusion Detection Systems (e.g. Snort) show inefficiency and even fail to perform for the faster internet. We have presented a fully hardware based system to overcome these shortcomings of software-based solutions. By implementing complete intrusion detection system on FPGA with embedded processor, we can solve the problem of performance and it has capability of intrusion detection in multigigabit network environment.

---

# CONTENTS

---

<b>Candidate's Declaration and Certificate</b>		<b>i</b>
<b>Acknowledgements</b>		<b>ii</b>
<b>Abstract</b>		<b>iii</b>
<b>Table of Contents</b>		<b>iv</b>
<b>CHAPTER 1</b>	<b>Introduction and Statement of the Problem</b>	<b>1</b>
	1.1 Introduction	1
	1.2 Motivation	4
	1.3 Statement of the Problem	4
	1.4 Organization of the Report	4
<b>CHAPTER 2</b>	<b>Background and Literature Review</b>	<b>5</b>
	2.1 Introduction	5
	2.2 Snort	5
	2.3 FPGAs and Reconfigurable Computing	8
	2.4 Networking Protocols	12
	2.5 Firewalls	15
	2.6 Policy Engines	15
	2.7 Research Gaps	15
<b>CHAPTER 3</b>	<b>Designing of IDS</b>	<b>17</b>
	3.1 Proposed Architecture	17
	3.2 FPGA Kit used	21

<b>CHAPTER 4</b>	<b>Implementation of IDS</b>	23
	4.1 ISE 8.2i	23
	4.2 EDK and XPS	27
	4.3 ModelSim	37
<b>CHAPTER 5</b>	<b>Results</b>	38
	5.1 Lab Setup	38
	5.2 Resource Utilization	40
	5.3 Traffic Analysis	40
<b>CHAPTER 6</b>	<b>Conclusions and Suggestions for Future Work</b>	42
<b>REFERENCES</b>		43
<b>PUBLICATIONS</b>		45
<b>APPENDIX</b>	<b>Configured Xilinx IP Cores Source Code</b>	I

### 1.1. Introduction

Intrusion detection systems (IDS) are software or hardware systems that automate the process of monitoring the events occurring in a computer system or network, analyzing them for signs of security problems. They, in the general sense, identify anomalous, inappropriate, or incorrect access to a system. As network attacks have increased in number and severity over the past few years, intrusion detection systems have become a necessary addition to the security infrastructure of most organizations [1, 2].

#### Types of Intrusions

Intrusions can take several forms. They can occur as abnormal, unauthorized, or unwanted system usage. Examples related to networking follow.

- **Unauthorized Access.** Unauthorized access occurs when an individual gains access to a system they have no right to use. For example, a user may view web pages containing proprietary information that they have not been authorized to view.
- **Authorized Access.** An intrusion can occur even if the credentials of the individual accessing the system are correct. For example, an intruder can fraudulently obtain account information such as login names and passwords. The system believes the intruder is authorized. This is the most difficult type of intrusion to detect since the detector must consider what is being accessed and what operations are being performed.
- **Spam.** Spam is an unwanted electronic message from individuals or companies who send the message to people that may not desire to receive the message. These messages generally try to sell items, such as medication, loan applications, or pornography. Phishing is a heinous form of spam where a message supposedly from an authoritative institution, such as a bank, e-commerce site, or government agency, directs the recipient to reply to the message or go to a web page and enter sensitive information. These messages can be quite persuasive, claiming accounts will be deactivated unless information is verified.
- **Virus.** A virus is a piece of malware hidden in files or emails. Once activated by the host, the virus replicates itself and spreads to additional hosts. Viruses

generally spread via email, requesting that the recipient view an attachment. A clever virus writer writes code to search an infected host's address book to find additional recipients. The virus assumes the host identity when sending new email messages, increasing the likelihood that the target becomes infected.

- **Worm.** A worm exploits vulnerability in a system to execute code without the user actively starting it. The most common form of worm exploits buffer overflows, whereby the processor stack is subverted. Malicious code extends beyond the allocated buffer and is executed. Worms take advantage of software flaws that may be difficult to find but are quick to exploit.
- **Denial of Service.** Denial of Service (DoS) prevents legitimate users from accessing a system. DoS is accomplished by flooding a system with data that takes time to process. This inundation of events grinds services provided by the system to a standstill as each request is processed sequentially. Web servers and email servers are frequent targets of such attacks, and the effects of a DoS attack can be very detrimental to those providing the service. Ecommerce is especially sensitive to such attacks, since any loss of service can mean the loss of a customer's business.

### **Methods of the Detection**

There are three basic ways to detect an intrusion: anomaly detection, signature detection, and learning. The detector should signal an alarm when a breach of security is attempted.

- **Anomaly Detection.** In anomaly detection, the detector recognizes deviations from standard behaviour. Abnormal behaviour is considered suspect. For example, a flood of traffic to a particular TCP port could signal the beginning of a worm/virus outbreak or a DoS attack. However, anomaly detection can result in false alarms. The event in question may be a previously unseen event that is perfectly legitimate, such as the distribution of a software update. The use of thresholds and statistical analysis is used heavily to prevent false alarms [27, 28].
- **Signature Detection.** The second method of detection is to search packets or flows for known signatures. This requires that the detector know what to search for in advance. This could involve searching packet headers for suspect port



numbers or IP addresses, or it could trigger searching payload for worm or virus signatures.

- **Learning.** Finally, an effective detector should be able to learn about and react to new intrusion attempts. Learning requires training time for the system to determine what constitutes normal behaviour and who are legitimate users. Once trained, the system reacts to abnormal behaviour and makes decisions on what actions to take. One technique to learn about new signatures is to use thresholds to determine if a signature appears to be occurring too frequently.

### **Responses**

Once an intrusion is detected, a response should be taken. The response could be to write to a log file or to email an administrator. The type of response depends on the way the system is configured. An intrusion detection system (IDS) can be configured as either passive or active. If the system is configured to be passive, as intrusion detection systems are, countermeasures cannot be performed to stop the intrusion because the system is only being monitored. Passive systems just inform an authority of security breaches. It is left to that authority to determine what to do about the problem. Intrusion prevention systems can take countermeasures. In-line, active systems stop the flood of worms, which otherwise infect all vulnerable hosts in a matter of minutes. Countermeasures may include dropping an offending packet, terminating a user's connection, or blacklisting an IP or email address.

### **Effects of False Alarms**

False alarms in intrusion detection systems are a serious problem. A false alarm occurs when an event or sequence of events causes an alarm to trigger even though the event was legitimate. For example, a false alarm may be raised at a web server if a certain web page becomes popular. A flash crowd, for example, occurs when there is a sudden surge of interest in a particular page. A term called "slashdotted" has been coined to describe the effect of large scale access to a web page when the URL is posted in an article on the [www.slashdot.org](http://www.slashdot.org) website. An intrusion detection system (IDS) may determine this is a distributed DoS attack when, in fact, the web server cannot meet the demand of all the legitimate users who want to access the content. When an alarm is raised, the reason for the alarm should be genuine. A system that generates many false alarms results in real alarms being overlooked. Additionally, when events occur too often, the logging and alarming mechanisms become overloaded.

## **1.2. Motivation**

Intrusions are unlikely to stop anytime soon. As more people, institutions, governments, and companies are networked together, the threat of intrusions increases. It is exceedingly difficult to build and maintain systems that are totally foolproof and have no security holes. Due to time-to-market constraints and because it takes too long to test all possible permutations of events that can produce an intrusion, it is unlikely that systems will ever be built that are totally secure. Network systems that can detect intrusions and prevent future intrusions are critical for security.

The main motivating factor for undertaking dissertation work was that current Intrusion Detection Systems are almost entirely implemented using software that runs on processors that cannot scale to process data on fast links. Hardware implementations allow for higher throughput, increase rule capacity, and take advantage of the parallelism that is inherent in rule processing.

## **1.3. Statement of the problem**

To design and implement Intrusion Detection System with FPGA. This will involve:

- designing Gigabit Ethernet interface,
- an IP packet extractor,
- ruleset matching and updating framework and
- logging system for defaulting packets.

## **1.4. Organisation of the Report**

This report describes a solution to the problem presented in Section 1.2.

Chapter 2 provides a background on Snort the de facto standard for Intrusion Detection Systems, FPGAs and Reconfigurable Computing available on present day FPGA kits and Networking protocols. Existing work in the same field is reviewed.

In Chapter 3, design of Intrusion Detection System is presented. Proposed architecture is discussed and details components required for implementation are given.

Chapter 4 discusses the implementation details and details of the software tools used - Integrated Software Environment (ISE) 8.2i and Embedded Development Kit(EDK) 8.2i for the implementation of the system are given.

Chapter 5 presents the simulation results, discusses the Snort rules, measures the memory requirement, and analyzes the performance of the IDS.

Chapter 6 summarizes the findings and give suggestions for future work.

### 2.1. Introduction

High performance intrusion detection and prevention systems are needed by network administrators in order to protect Internet systems from attack [3]. Researchers have been working to implement components of intrusion detection and prevention systems for the highly popular Snort system in reconfigurable hardware. Several attempts have been made to improve the system performance by migrating functionality from software to hardware. Though software is relatively slow, it is well suited to perform lightweight processing on low volumes of network data. On the other hand, fast hardware is best suited for computationally intensive processing on network traffic and can sustain much higher network throughput [4]. Since Snort has become the de facto standard for NIDS, a number of groups have worked to measure the performance of the system. As the number of header rules and signatures to match increases, the number of packets dropped by the sensor also increases. It is unacceptable for an IDS to not examine some packets. Schaelicke et al found that Snort inadequately acts as a sensor on higher speed links [5]. Their study showed that Snort alone is not to blame, but the platform running the software is partially responsible. Architectural decisions and the memory subsystem are critical factors in the performance of the NIDS. They found that even on a dual Pentium-4 Xeon running at 2.4 GHz with Hyper threading technology; the system could only support 543 rules in the best case such that no packets were ever dropped. Furthermore, the authors found that only two of their test systems could support saturated 100 Mbps links. This is troublesome because Gigabit links are common today.

### 2.2. Snort

In Snort (a popular open source NIDS) -more than 2,600 rules are there and more than 80% of the rules contain signatures [3]. More than 80% of the CPU time for Snort is consumed by the string matching task alone. The pattern matching functions of the NIDS can be significantly accelerated using semi-custom hardware and, in particular, content addressable memory (CAM.) The CAM is used in a variety of applications and is arguably best known as the decision engine for the IP router. The CAM can provide an extremely fast pattern matching function on the order of 12 ns or less depending upon the size of the CAM [17].

In “Exploiting Reconfigurable Hardware for Network Security” [6] by Shaomeng Li, Jim Torresen, Oddvar Soraasen of Department of Informatics, University of Oslo, Norway- they have demonstrated that the Snort IDS performance can be improved using CAM for implementing the detection engine of Snort. Firstly, they have not done any optimisation in storing or comparing snort ruleset while using FPGA for the same. Secondly, only the computationally intensive portion of the system is offloaded on the FPGA whereas complete IDS can be designed as a separate hardware device.

### 2.2.1. Snort Rule Features

Table 2.1 is divided into header and payload options that are available to the intrusion rule writer. The header options are split into sections that represent where they are found in a packet, starting with the IP header, then the TCP header, and finally the ICMP header.

Table 2.1: The header and payload options that are available to Snort rule writers.

<u>Header Options</u>	<u>Payload options</u>
Protocol IP Addresses Same IP TTL ToS Identification IP Options Fragment Bits Fragment Offset Data Size	Content Perl compatible Regular expressions Case Sensitivity Offset Depth Within Raw Bytes Byte Jump Byte Tests
Ports Flags Sequence Number Acknowledgement Flags	
ICMP Type ICMP Code ICMP Identification ICMP Sequence Number	

### **2.2.1.1. Header Options**

The header options shown in Table 2.1 correspond to distinct fields in packet headers. The IP addresses and ports are unique in that they allow ranges and masks. The other fields are exact match.

### **2.2.1.2. Payload Options**

Payload options are concerned with the presence and location of strings to find expressed as either static signatures or regular expressions. The depth construct allows the search of a specified string up to a certain location in the payload. The offset construct specifies where to begin looking for a given string. As with the header options, payload options can be mixed together, and multiple signatures can be specified in a rule.

### **2.2.2. Portability Difficulties**

A brute-force translation of Snort from a software implementation to a hardware implementation is inefficient. The software implementation is inherently sequential, while hardware is efficient at implementing parallelism. The features available are difficult to port to hardware. For example, performing string matching within certain bounds of the payload is a complicated task for hardware to perform due to the very specific requirements that can be placed on different strings.

There are three challenges to performing rule processing in hardware:

- Scalability to process and store increasingly complex rules
- Correlation between header classification and payload content
- Adaptability to changing environment

One of the most challenging tasks in a rule processor is correlation of criteria. Every packet can contain matches for multiple header classifications and payload signatures. The system must correlate these matches to determine rule matches. While a single rule is trivial to process, consider that there are 2,464 rules found in Snort. In software, the correlation is performed using linked lists in memory. Implementing the same lists in hardware is detrimental to performance due to the numerous memory look-ups required.

In order to protect against evolving threats, the system must be adaptable. Rules change over time as new threats emerge. The system must adapt to scan for new forms of malware [7]. Reconfigurable hardware enables the system to adapt to new threats quickly and at low expense.

## 2.3. FPGAs and Reconfigurable Computing

### 2.3.1. Introduction to FPGA's

A field programmable gate array (FPGA) is a general-purpose integrated circuit that is programmed by the designer rather than the device manufacturer. Unlike an application-specific integrated circuit (ASIC), which can perform a similar function in an electronic system, an FPGA can be reprogrammed by downloading a configuration program called a *bitstream*, even after it has been deployed into a system. Much like the object code for a microprocessor, a bitstream is the product of compilation tools that translate the high level abstractions produced by a designer into something equivalent but low level and executable. Over the last three decades, FPGA's have grown from simple logic components, through moderate prototyping platforms and more recently, as complete System on a chip (SoC) components. One of the greatest advantages with FPGA's is that they can be used as custom hardware avoiding the initial costs, fabrication costs and fabrication time.

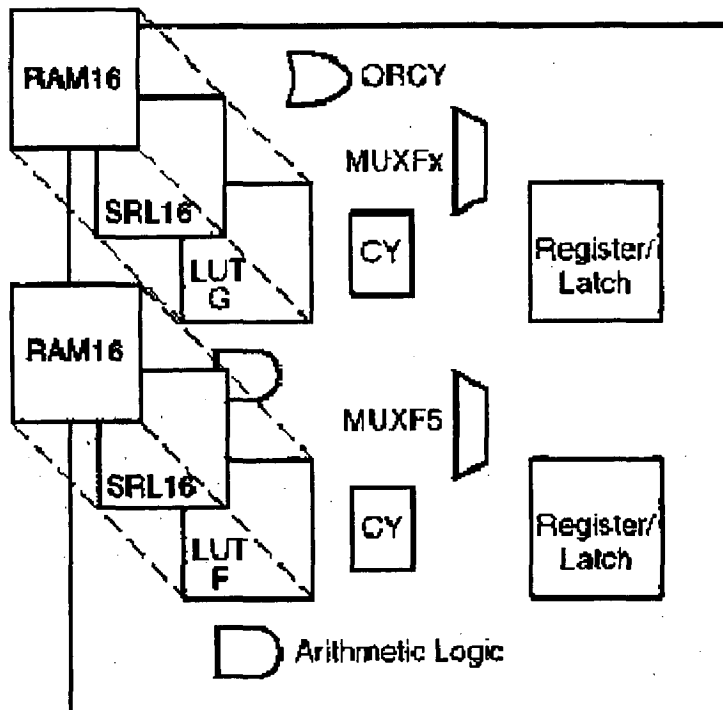


Fig 2.1. Virtex Family FPGA Logic slice

A simple FPGA fabric consists of an array of configurable logic blocks (CLBs) attached by a programmable interconnect. Digital circuits are mapped to the CLBs which consist of logic slices which consists of look-up tables (LUTs) and flip-flops (FFs). Each logic slice as shown in Fig 2.1 contains two 4-input lookup tables (LUTs), two configurable D-flip flops, multiplexers, dedicated carry logic, and gates used for creating slice based multipliers. Each LUT can implement an arbitrary 4-input Boolean function. Four inputs is a

good size for a look-up table as suggested by various studies, trading utility (complexity of a block) against utilization (what fraction ends up in use) [8, 9]. Coupled with dedicated logic for implementing fast carry circuits, the LUTs can also be used to build fast adder/subtractors and multipliers of essentially any word size. In addition to implementing Boolean functions, each LUT can also be configured as a 16x1 bit RAM or as a shift register. In addition to logic slices, current generation FPGAs include additional diffused hardware resources beneficial for embedded systems. For example the Xilinx XC4FX140 which is a product of the latest 90 nm CMOS technology features various dedicated digital signal processing 18-bits multipliers and accumulators which are called as DSP slices, dual port BLOCK RAM's which can be used for storing few kilobytes of data, Digital Clock Managers, 2 Power-PC RISC Processors, 10/100/1000 Ethernet MAC Blocks, and Rocket IO Serial Transceivers which can be used to provide high-speed connections for communication between FPGA's and inter-module communications. Moreover with the advance of Moore's Law, FPGA's are also increasing in total capacity and speed which gives the users more number of computational units.

### 2.3.2. Programming an FPGA

In current practice, hardware descriptive languages (HDL) and schematics are widely used to implement applications on the FPGAs. Fig 2.2 is a pictorial representation of the design flow that usually occurs with FPGA's.

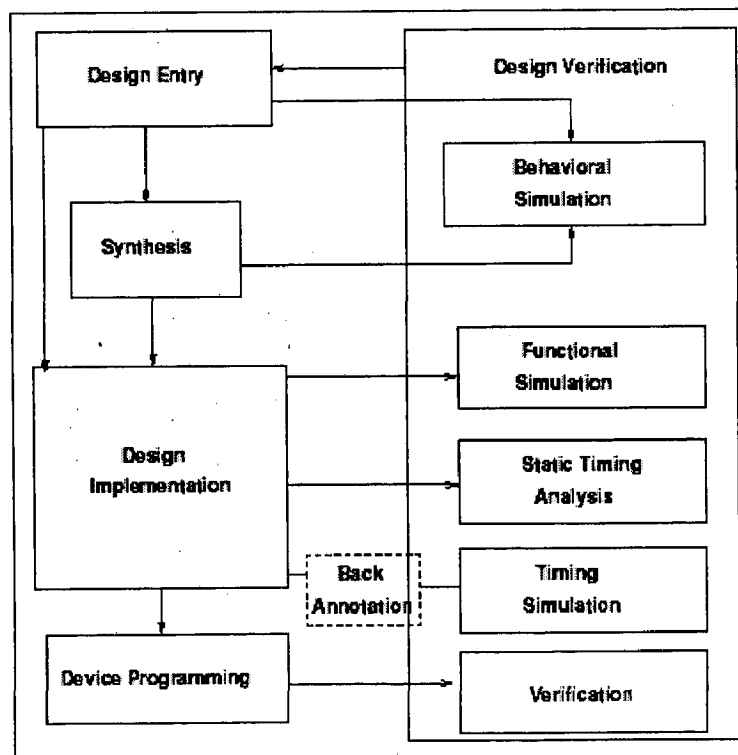


Fig 2.2. FPGA Design Flow

Several HDL languages like VHDL (Very High Speed Integrated Circuit Hardware Description Language), Verilog, JHDL, SystemC, Streams - C, HandelC etc exist where in the application can be specified and this stage is usually called the Design Entry stage. After this stage, the design is verified for its functionality through a Simulation process. After the Simulation process the design is converted to a form of representation called the netlist which is the complete representation of the logic in terms of basic gates (AND,OR,XOR,NOT). After this process the design is mapped which is mapping the above obtained netlist to the actual Configurable Logic Blocks (CLB) and Input/Output Blocks (IOB) available in the device that has been targeted. After the design has been mapped the next stage in the process is called Place and Route where in the design that has been mapped is physically mapped to the device's logic cells based on the timing and layout requirements. After these steps, a timing simulation is performed and the design is modified so that the best possible timing is obtained. After the re-design, the design is again sent through the process of converting the design into a netlist, MAP and then Place and Route. After the final Place and Route the design is converted to a configuration file called a BIT file which defines the behaviour of the FPGA that has been targeted. The BIT file obtained can be downloaded into the FPGA and verified for functionality.

### **2.3.3. Hybrid CPU/FPGA Architecture's**

Hybrid CPU/FPGA architecture's are the first of its kind from Xilinx, Inc which are also called as Platform FPGA's which are the latest FPGA's with processors embedded (Hard Cores) in the FPGA fabric apart from the vast number of freely available logic gates. The processors inside the Platform FPGA's are IBM PowerPC 405's which implement the standard RISC style architecture and are based on the Core-Connect Architecture [10] from IBM and are implemented as Hard Cores inside the FPGA. This level of integration allows various Intellectual Property (IP) cores to be attached to the processor and the cores are also easily accessible through the Core Connect Architecture that is provided as a Intellectual property core (Soft Core). The Core Connect provides three bus standards as a means of communication between the PowerPC and other cores. The three bus standards are Processor Local Bus (PLB) , On-Chip Peripheral Bus (OPB) and the Device Control Register (DCR) bus. The processor local bus (PLB) is used to connect processor cores to the system main memory and other high speed devices. The OPB bus is dedicated for connecting slower on-chip peripheral devices indirectly to the CPU. The OPB bus supports variable size data transfers and as well as flexible arbitration protocols. Both the PLB and OPB buses have their



own bus arbiters, and the two buses are interconnected by at least one bridge (PLB2OPB Bridge or OPB2PLB Bridge). Various intellectual Property (IP) Cores (Soft Cores) are also available in order to interact with various standard peripherals in the FPGA such as the DDR SDRAM (Double Data Rate - Synchronous Dynamic Random Access Memory) , EEPROM ( Electrically Erasable Programmable Read-Only Memory), PCI (Peripheral Component Interconnect), RS232 UART (Universal Asynchronous Receiver/ Transmitter). In addition to the peripheral and utility Intellectual Property cores, an interface called the Intellectual Property Interface (IPIF) is available in the form of a soft core which allows any Intellectual Property (IP) Core to connect to either of the buses. The IPIF is decomposed into two layers to allow easy migration of peripherals or IP cores to each of the different system buses in the Core Connect Architecture. The first layer provides an interface facility to be used between the IP core and the IPIF. The second layer is a bus specific portion, and interfaces the IPIF to one of the buses. These interface modules allow to greatly accelerate the process of connecting pre-existent IP, or creating a new IP in a system. The IPIF provides two different types of attachment to an IP core: a slave and a master attachment. With the master attachment, user cores have the ability to initiate bus transactions. Moreover, bus arbitration logic is also included within the master attachment. However it is the user core's responsibility to re-arbitrate or abort the bus and switch the data bus between the slave and master modes.

#### **2.3.4. Reconfigurable Computing**

Reconfigurable Computing (RC) [11] started of during the late 1960's but was still a research field until the late 1980's because of lack of availability of suitable hardware. But with the advent of Field Programmable Gate Array Technology (FPGA), the field of reconfigurable computing got a boost since FPGA's provided a reconfigurable platform and gave a broader meaning to the field. The main feature of Reconfigurable Computing is the ability of the hardware to reconfigure based on various functions. Although FPGA's provided a full reconfiguration of the chip since its ingression until recently, due to increase in technology various FPGA's now even support partial reconfiguration which means that a portion of the device can be altered even though when the FPGA is actually running. When Reconfigurable Computing was in its initial development stages, the cost of FPGA hardware and Reconfigurable cards were very costly, but as years passed by and with the advancement of Moore's Law which gave more transistors per die, FPGA's and Reconfigurable Computing boards have become a lot cheaper. Moreover the introduction of FPGA's with processors embedded in it became a stepping stone to the field of Reconfigurable Computing. For

example, today a Reconfigurable Computing Mother board with a Xilinx Virtex II Pro FPGA which houses around 100,000 free logic gates, two PowerPC processors and the ability to house a DDR SDRAM, a Compact Flash Card and various other peripherals costs around \$250 as compared to \$6000 in the year 1998 which housed a Xilinx XC4085 FPGA with only 10,000 logic gates and with minimal peripheral support.

### **Dynamic Reconfiguration of Functional Blocks (DRP)**

In the Virtex family of FPGAs, the Configuration Memory is used primarily to implement user logic, connectivity and I/Os, but it is also used for other purposes. For example, it is used to specify a variety of static conditions in functional blocks, such as Digital Clock Managers (DCMs) and Multi-Gigabit Transceivers (MGTs). Sometimes an application requires a change in these conditions in the functional blocks while the block is operational. This can be accomplished through the global Internal Configuration Access Port (ICAP), or through partial dynamic reconfiguration using JTAG or SelectMAP in the Persist mode. However, the reconfiguration port that is an integral part of each functional block simplifies this process greatly [17].

This addressable, parallel write/read configuration memory that is implemented in each functional block that might require reconfiguration and it has the following attributes:

- It is directly accessible from the FPGA fabric. Configuration bits can be written to and/or read from depending on their function.
- Each bit of memory is initialized with the value of the corresponding configuration memory bit in the bitstream. Memory bits can also be changed later through the ICAP.
- The output of each memory bit drives the functional block logic, so the content of this memory determines the configuration of the functional block.

### **2.4. Networking Protocols**

The Internet can be described using a seven-layer model [12]. Starting at the lowest level, the layers are physical, data link, network, transport, session, presentation, and application. The physical layer is concerned with how bits of information are transferred from one location to another. The data link layer defines frame formats to specify where information begins and ends. The network layer defines how data is forwarded between hosts. The transport layer defines how data is transferred reliably. The session layer determines how communication sessions are created and authenticated. The presentation layer defines how data is internally

represented for transmission. Finally, the application layer generates and/or interprets the data that has been transferred [12,13]. Rule processing resides in layers three and above.

### 2.4.1. Internet Protocol

The Internet Protocol (IP) is used extensively today in the global network, providing a best-effort delivery of IP packets [12]. A typical IP packet header consists of 20 bytes, as shown in Fig 2.2. The main fields of the header are:

- ToS - type of service, used for applications requiring certain quality of service (QoS) guarantees

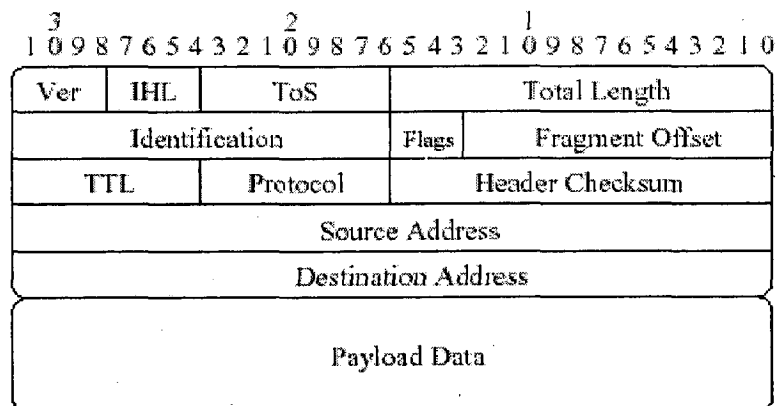


Fig 2.2: A typical IP packet consists of 20 bytes of header and up to 1480 bytes of payload data on an Ethernet network.

- Total length - the length, in bytes, of the entire IP packet
- TTL - time to live, the maximum number of hops the IP packet can make in the network before being discarded
- Protocol - the encapsulated protocol used in the IP packet
- Source Address - the source network and local address of the sender
- Destination Address - the destination network and local address of the receiver

IP packets are the fundamental unit of processing for the rule processing architectures. Higher level protocols, such as the User Datagram Protocol and the Transmission Control Protocol are encapsulated within the IP payload data.

### 2.4.2. User Datagram Protocol

The User Datagram Protocol (UDP) is a best-effort delivery protocol [12]. UDP is most commonly used for multimedia applications such as streaming video and audio. The main addition of UDP over IP is port numbers, which allows the operating system to deliver data to the appropriate application.

### 2.4.3. Transmission Control Protocol

The Transmission Control Protocol (TCP) is the predominate protocol used today [12]. The authors of [102] showed 85% of network traffic is TCP. TCP provides

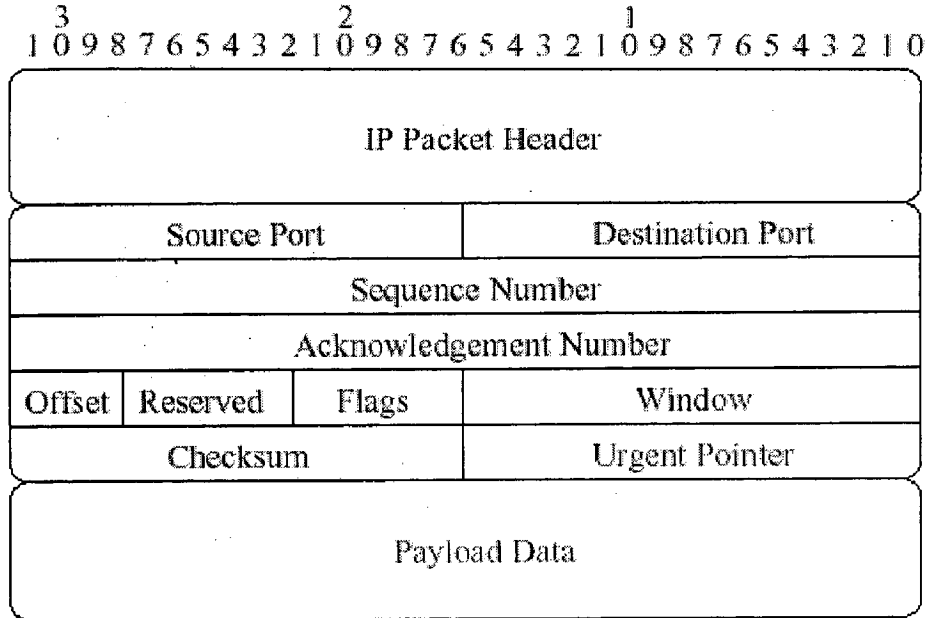


Fig 2.3: A TCP packet consists of 40 bytes of header information (20 from the IP packet header and 20 from the TCP header) and up to 1460 bytes of payload data on an Ethernet.

a reliable, in-order transmission of data. While protocols such as IP and UDP are stateless, TCP is a state-based protocol, requiring a connection to be established. By maintaining state, large transfers of data are possible. The principle application of TCP is reliable data transfer, such as for web page viewing, email, and file transfer [14].

A TCP packet, as shown in Fig 2.3, appends 20 additional bytes of header onto the IP packet header. Fields of note are:

- Source Port & Destination Port - numbers to aid the operating system in determining where to send the payload
- Sequence Number - the number given to the first byte of data found in the payload to properly order data for delivery to the application
- Acknowledgement Number - the number given to the next byte expected at the receiver, which informs the sender as to what bytes have been received
- Window - the number of bytes that can be in-flight between the sender and receiver

TCP data is transferred in flows. A flow is characterized by four fields: the source IP address, the destination IP address, the source port and the destination port. These four fields uniquely identify a TCP communication channel between the sender and receiver. Since the

maximum transfer unit of most networks is 1500 bytes (Ethernet frames) and most files are larger than 1460 bytes, a flow needs to be established in order to reliably transfer all data bytes in the file.

## **2.5. Firewalls**

A firewall is a device that inspects packets before they arrive at their destination. If the packets are found to contain questionable data, they are flagged. Firewalls can be used to drop traffic entering a network, or they can be used to prevent traffic from leaving a network. The term "firewall" is used to suggest the prevention of spreading harmful materials from one area to another.

The earliest firewalls were based solely on examining the header of IP packets [15]. These implementations relied on allowing known port numbers and IP addresses to pass through. For example, TCP traffic destined to port 25 or port 80 is generally safe since these are the ports for email and web traffic. However, an attacker can easily hide intrusions in these well-known port numbers. Header-based software solutions are still common, and can be effective at removing a significant portion of unwanted traffic. Zone Alarm is a common example of a firewall solution.

## **2.6. Policy Engines**

A policy engine examines more areas of a packet than just the header before deciding whether a packet is safe or not. The problem with current policy engines is their complexity. As a result, they passively monitor the network. Policy engines perform signature detection, correlate events, and compute complex logical operations.

Paxson et. al developed an IDS called Bro [16]. Using a proprietary security language, this software-based system used libpcap to read network packets on a PC. Event engines used libpcap to validate packets, correlate the received packet with similar packets from the same flow, and process payload data. If alerts were generated, a policy script was run to determine what action to take.

Snort is another type of policy engine that uses rules to determine whether intrusions have occurred [3]. Snort has been adopted as the tool for intrusion detection.

## **2.7. Research Gaps**

The literature review shows that complete Intrusion Detection Systems does not have a Gigabit Ethernet interface on a Hardware Kit with known implementation details. Also there is a need to develop a Web based GUI to manage it, and a facility for user/administrator to update the ruleset over the network while the IDS is running.

We have tried to develop an Intrusion Detection System as a separate device with a Gigabit Ethernet interface on a Hardware Kit, and with a facility for user/administrator to update the ruleset while the IDS is running.

3.1. Proposed Architecture

The overall design principle for this architecture is the use of dedicated IP cores by

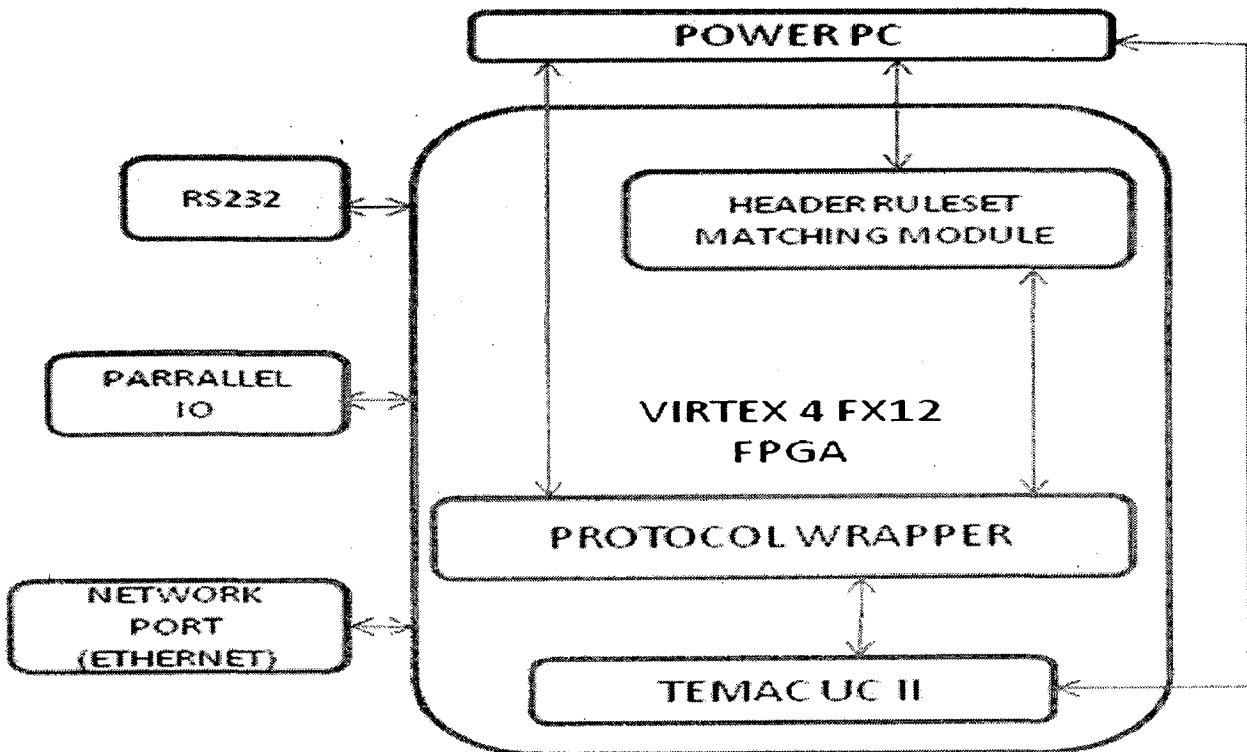


Fig 3.1. Architecture of IDS

Xilinx [17] and Treck Embedded TCP/IP Stack [18] with parallel ruleset matching header processor module. Block diagram of the architecture is as given in Fig 3.1. Internally with protocol wrapper and Treck Embedded TCP/IP Stack the packet header is extracted and matched with the ruleset using Content Addressable Memories. The header part to match is placed in a CAM entry and is compared in parallel with all rulesets currently in the database. The packet is either logged or left depending on the matching result.

The designed Intrusion Detection System is intended to be placed at the entry level switch of the LAN where in it will receive the complete incoming and outgoing network traffic from the port of the switch configured for promiscuous mode as shown in Fig 3.2.

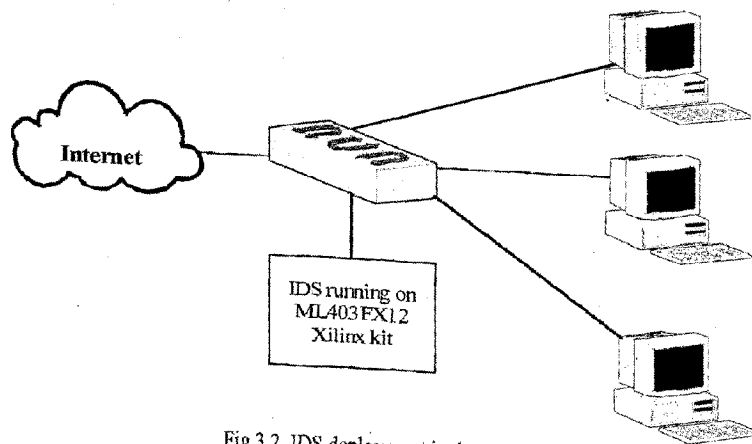


Fig 3.2. IDS deployment in the network

### Hardware Infrastructure

#### Layered Protocol Wrappers

The layered protocol wrappers provide an interface that identifies the fields within an IP packet [19]. This eases the design of an IP-based networking application by allowing it to operate at OSI layer three. We have used this module to extract the IP packet header and have used TCP packets only.

The input for this module is 32-bits on every clock input and on every clock the state machine changes its state according to the state diagram. The 32-bits are used for this design as the header protocols are 32-bit spanned and the design utilizes this concept efficiently. The state diagram is as follows (Fig 3.2):

1. Check for IP version 4. If the version is incorrect discard the 32-bits and start over. If the version is correct, Extract the IP header length and total packet length, go to next state (i.e. State 2).
2. Extract the ID (Used to identify the fragments of one datagram from those of another). Go to next state (i.e. State 3).
3. Check for TCP protocol. If not found then skip the 32-bits and go to next state which is State 1 else go to next state (i.e. State 4).



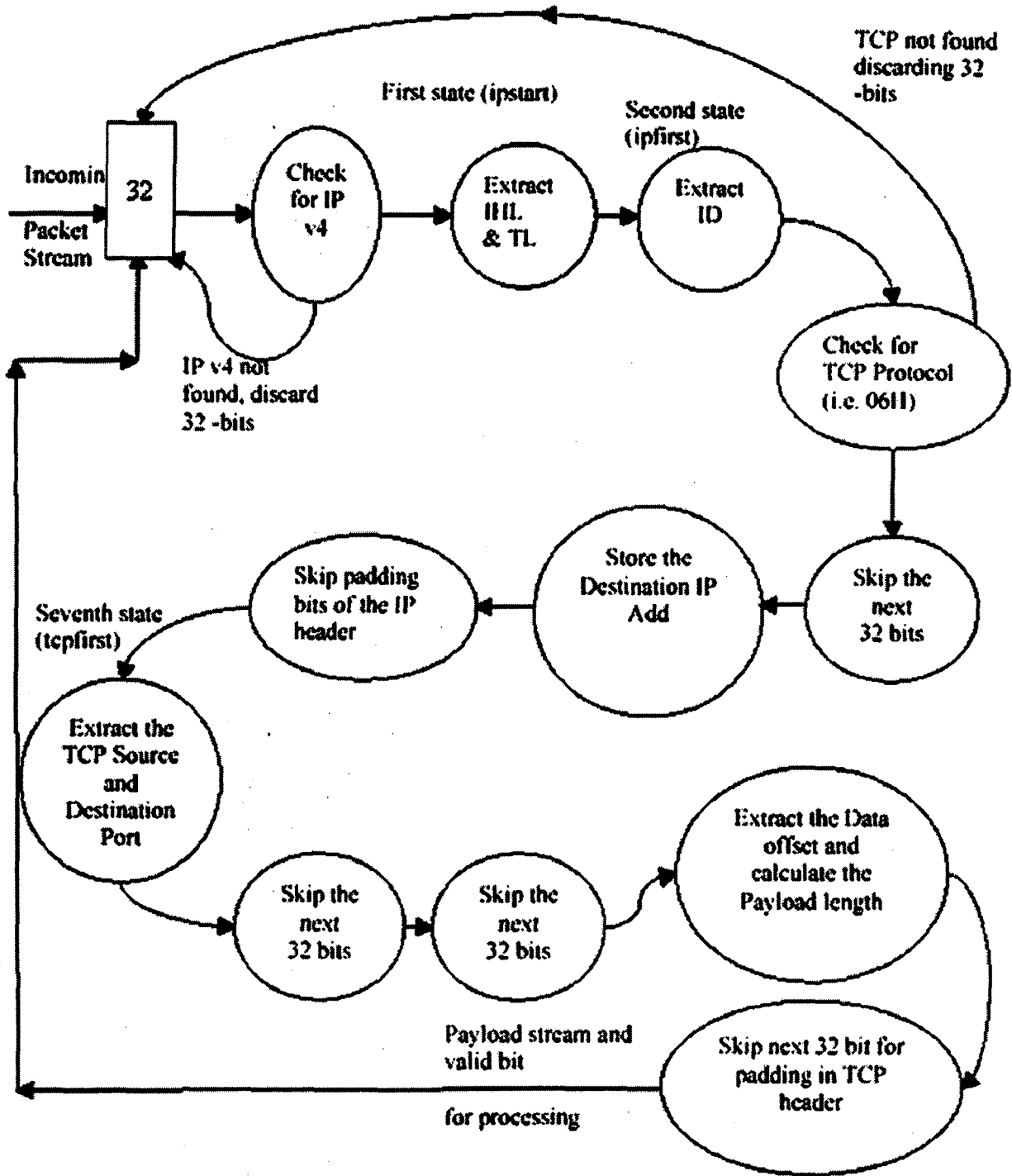


Fig. 3.2. TCP/IP Header extraction State diagram

4. Skip 32-bits. Go to next state (i.e. State 5).
5. Store the Destination IP Address. Go to next state (i.e. State 6).
6. Skip the padding bits (Used as a filter to guarantee that the data starts on a 32 bit boundary.) for the IP header. Go to next state (i.e. State 7).
7. Extract and store the TCP source and Destination Port numbers. Go to next state (i.e. State 8).
8. Skip 32-bits. Go to next state (i.e. State 9).

9. Skip 32-bits. Go to next state (i.e. State 10).
10. Extract the Data offset parameter. Calculate the Payload length which is nothing but the Total length – IP header length – TCP header length. Go to next state (i.e. State 11).
11. Skip the next 32-bits for the TCP header padding and go to next state (i.e. State 12).
12. Send a 32-bit stream and the valid bit (indicating valid payload on line). Go to next State (i.e. State 1).

### **Memory Controllers**

The SRAM controller provides two arbitrated interfaces for access to a 2 MB SRAM. A request and grant protocol is used to access SRAM [20].

The SDRAM controller provides three arbitrated interfaces to SDRAM. One is for reading only, one is for writing only, and one is for reading and writing. The read/write interface is used to access a bank of 64 MB SDRAM. The SDRAM controller also provides a simple request/grant interface with burst transfers.

### **Buffers**

Buffers are used throughout the system to store IP packets before processing.

### **Tri-Mode Ethernet MAC (TEMAC) UltraController-II**

As shown in the above block diagram the Tri-Mode Ethernet MAC (TEMAC) UltraController-II module is the Xilinx core [17]. This module is a minimal footprint, embedded network processing engine based on the PowerPC™ 405 (PPC405) processor core and the TEMAC core embedded within a Virtex™-4 FX Platform FPGA. The TEMAC UltraController-II module connects to an external PHY through Gigabit Media Independent Interface (GMII) and Management Data Input/Output (MDIO) interfaces and supports tri-mode (10/100/1000 Mb/s) Ethernet. Software running from the processor cache reads and writes through an On-Chip Memory (OCM) interface to two FIFOs that act as buffers between the different clock domains of the PPC405 OCM and the TEMAC. The TEMAC UltraController-II module uses minimal resources: one PPC405, one TEMAC, two Virtex-4 FIFOs, 20 slice flip-flops, and 18 look-up tables (LUTs). Because of the minimal footprint design, a greater number of FPGA logic resources remain available to the user.

### **Header Check**

Most of the 168 Snort header only rules look for specific TCP/UDP port numbers [3]. Each of the 168 header-only rules are checked in parallel using TCAM, and a rule match is declared if any of the headers match.

## TCAM

Ternary Content Addressable Memory (TCAM) is a type of memory that can perform parallel search at high speeds. A TCAM consists of a set of entries. The top entry of the TCAM has the smallest index and the bottom entry has the largest. Each entry is a bit vector of cells, where every cell can store one bit. Therefore, a TCAM entry can be used to store a string. A TCAM works as follows: given an input string, it compares this string against all entries in its memory in parallel, and reports one entry that matches the input [21].

## Ethernet Statistics core

The IP Ethernet Statistics core provides a user-configurable collection of statistical counters that are used to gather network traffic statistics for Xilinx Ethernet Media Access Controller (MAC) [22].

### 3.2. FPGA Kit used

The FPGA Kit used is a Xilinx Vertex-4 ML403 Evaluation Platform [23], as shown in Fig 3.3. Some of the salient features of the board are:

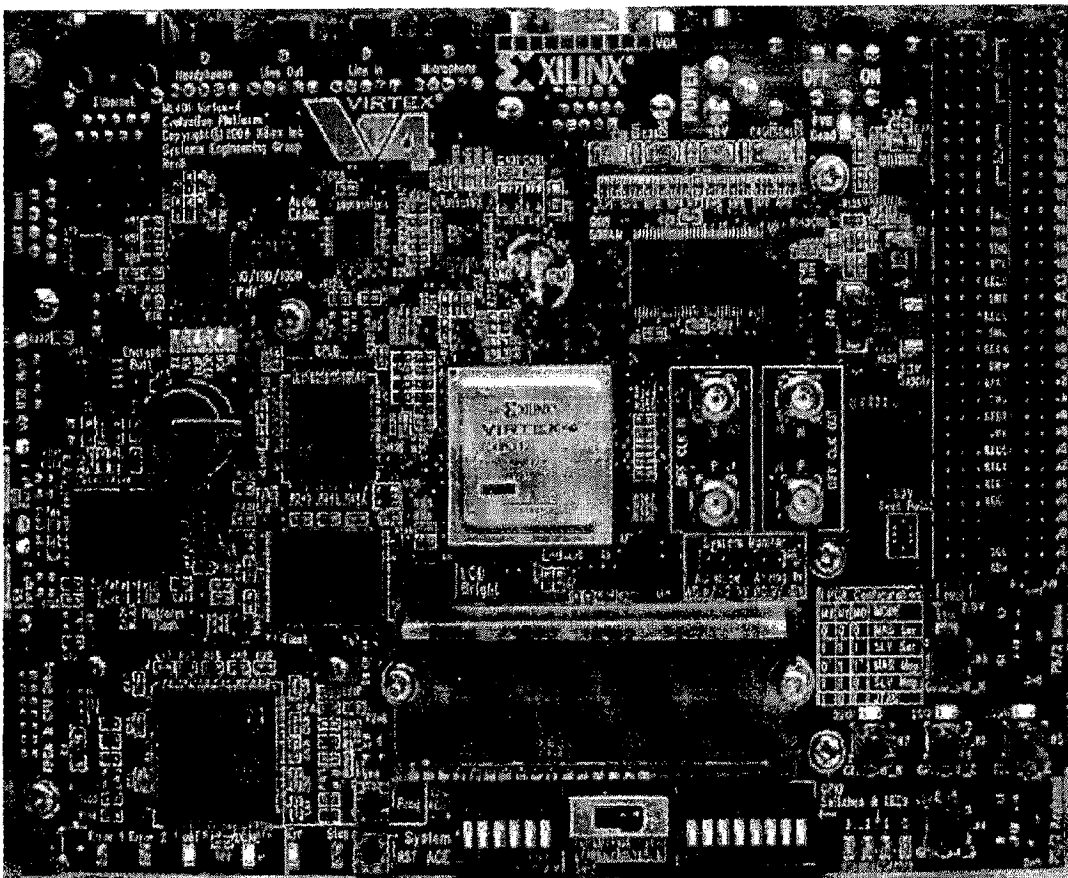


Fig 3.3. Xilinx Vertex-4 ML403 Evaluation Platform

- 64 MB DDR SDRAM, 32 bit interface running up to 266-MHz data rate.
- General purpose LEDs and push buttons.
- RS-232 serial port.
- 10/100/1000 Ethernet Port (RJ-45 Connector)
- One 4Kb IIC EEPROM.
- PS/2 mouse and keyboard connectors.
- JTAG configuration port for use with Parallel Cable III and Parallel Cable IV cable.
- System ACE and Compact Flash Connector.

The Virtex-4 family includes three platforms; Virtex-4 LX for logic, Virtex-4 SX for very high performance signal processing, and Virtex-4 FX for embedded processing and high-speed serial connectivity. Each version has a different mix of the special features and comes in a range of density to cover a variety of application sizes (as shown in Fig 3.4)

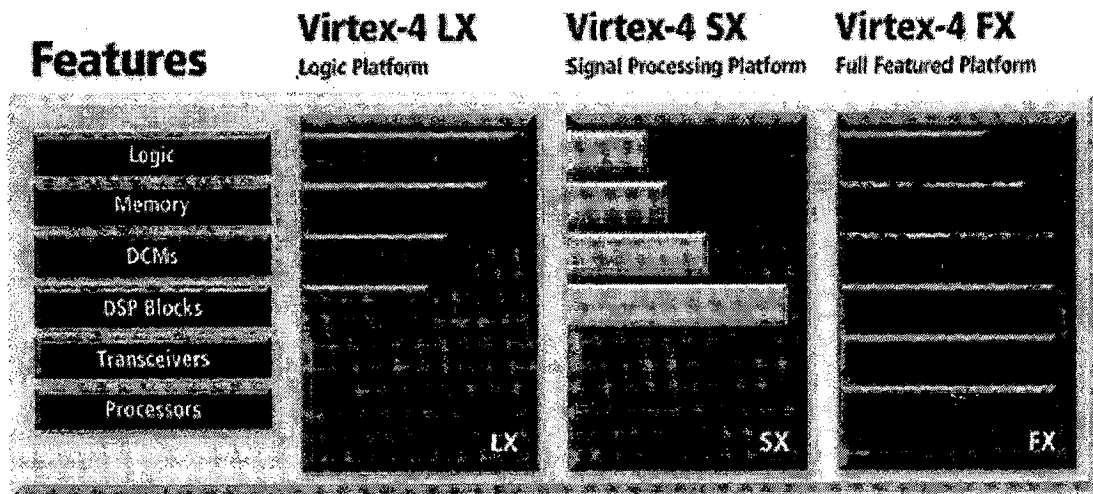


Fig 3.4. Virtex-4 family

The ML403 Evaluation platform has Xilinx Devices: XC4VFX12-FF668-10C (FPGA+PowerPC).

We have developed our prototype based on the architecture given in Fig 3.1. The prototype we have developed is programmed in VHDL. FPGA Logic is implemented on a Xilinx Vertex-4 ML403 Evaluation Platform [23], as shown in Fig 3.2.

We have used Integrated Software Environment (ISE) 8.2i and Embedded Development Kit(EDK) 8.2i for designing and implementation of the overall system on the kit. The simulations of all functions were conducted by the ModelSim. For performance evaluation of our prototype system, we applied Snort header ruleset and used Traffic Generator for generating network traffic for experiments.

Through JTAG cable the header ruleset stored in SDRAM can be updated without affecting the rest of the design. The overall design occupies only part of the FPGA resources available on the Xilinx ML403 Evaluation Platform.

#### **4.1. Integrated Software Environment (ISE) 8.2i**

The Integrated Software Environment (ISE™) is a Xilinx development system product that is required for implementing designs onto Xilinx programmable logic devices. It allows takes the design from design entry level to programming the Xilinx device. Various steps in the ISE design flow are:

##### **Design Entry**

Design entry is the first step in the ISE design flow. During design entry, source files are created based on the design objectives. The top-level design file is created using a Hardware Description Language (HDL), such as VHDL, Verilog, or ABEL, or using a schematic. Multiple formats can be used for the lower-level source files in the design. Project Navigator is used to create new project as shown in Fig 4.1 to 4.3.

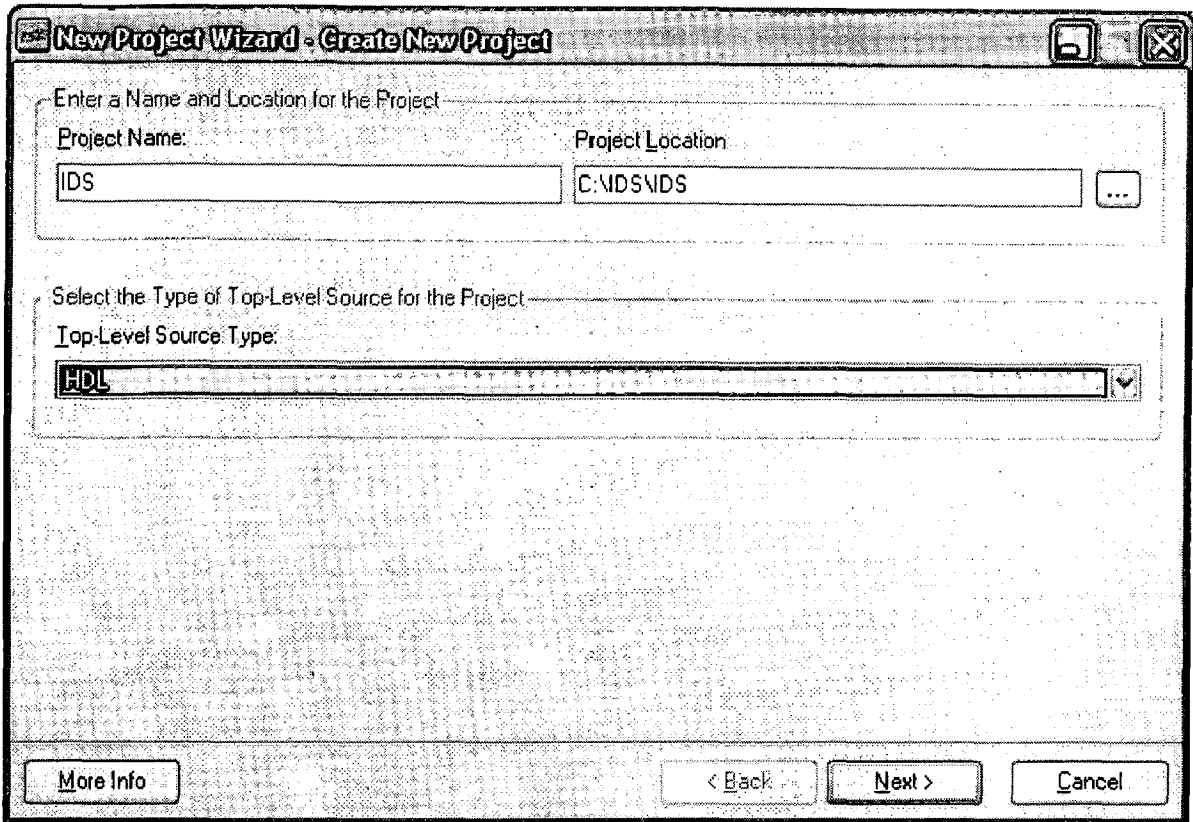


Fig 4.1 Creating New project in ISE Project Navigator

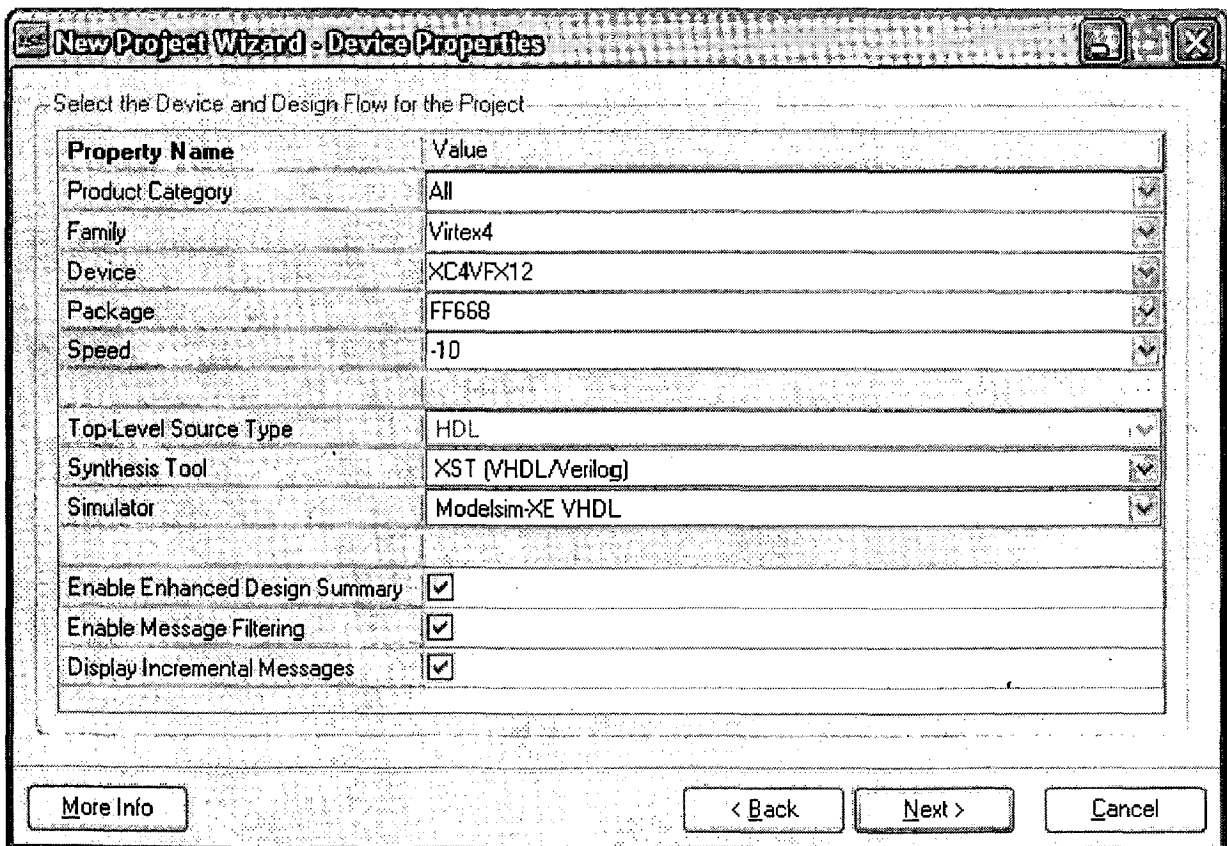


Fig 4.2 Selecting Device Properties

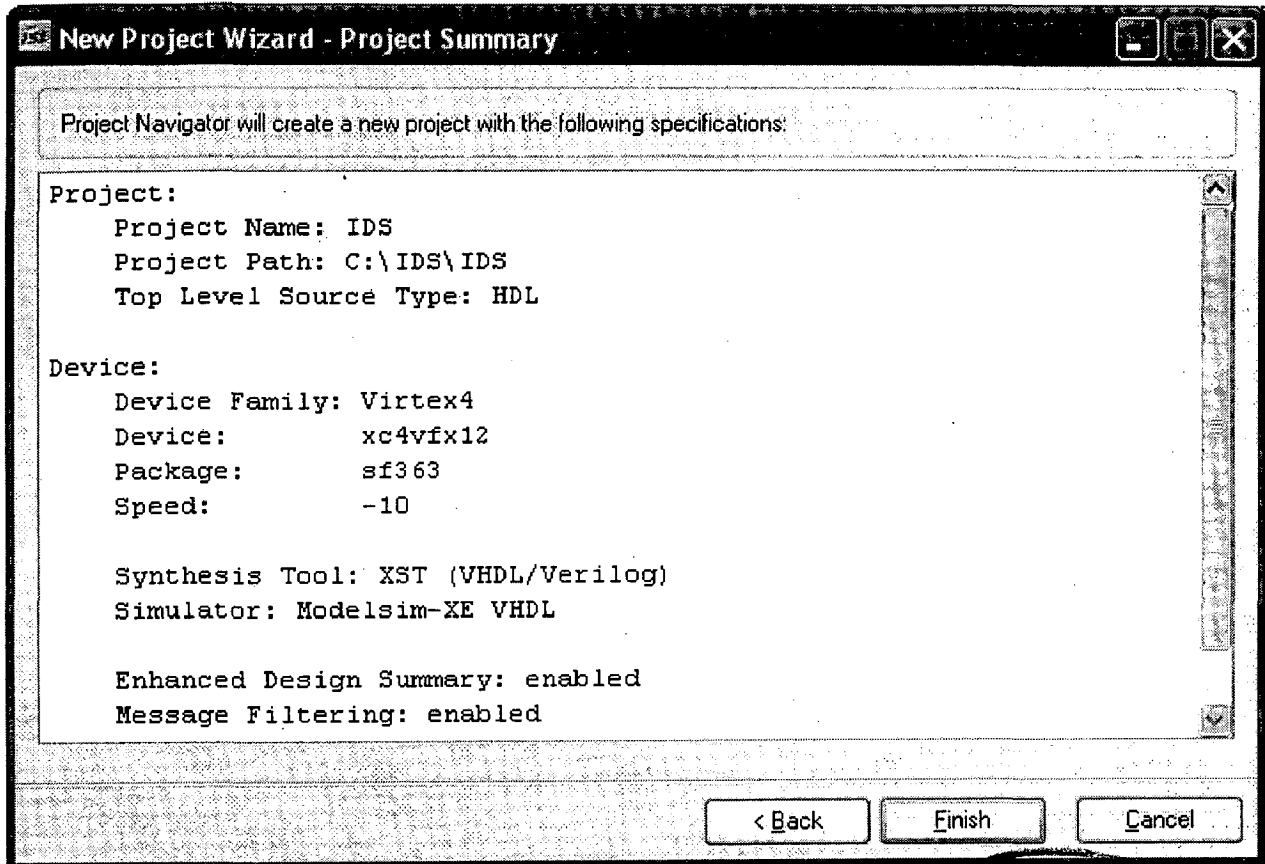
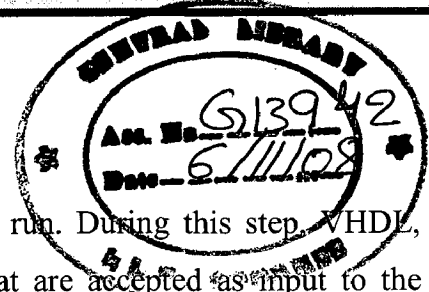


Fig 4.3 Project Summary



### Synthesis

After design entry and optional simulation, the synthesis is run. During this step, VHDL, Verilog, or mixed language designs become netlist files that are accepted as input to the implementation step.

### Implementation

After synthesis, the design implementation is run, which converts the logical design into a physical file format that can be downloaded to the selected target device. From Project Navigator, we can run the implementation process in one step, or we can run each of the implementation processes separately. Implementation processes vary depending on whether we are targeting a Field Programmable Gate Array (FPGA) or a Complex Programmable Logic Device (CPLD).

### Verification

The functionality of the design can be verified at several points in the design flow. Simulator software is used to verify the functionality and timing of the design or a portion of it. The simulator interprets VHDL or Verilog code into circuit functionality and displays logical results of the described HDL to determine correct circuit operation.

## Device Configuration

After generating a programming file, next step is configuring the device. During configuration, the configuration files are generated and the programming files are downloaded from a host computer to a Xilinx device.

## IMPACT

IMPACT (iMPACT) is a tool featuring batch and GUI operations as shown in Fig 4.4 and 4.5; it allows performing two basic functions: **Device Configuration** and **File Generation**. Configuration is the process of loading design-specific information into one or more FPGA, PROM, or CPLD devices to define the functional operations of the logical blocks, their interconnections, and the chip I/O.

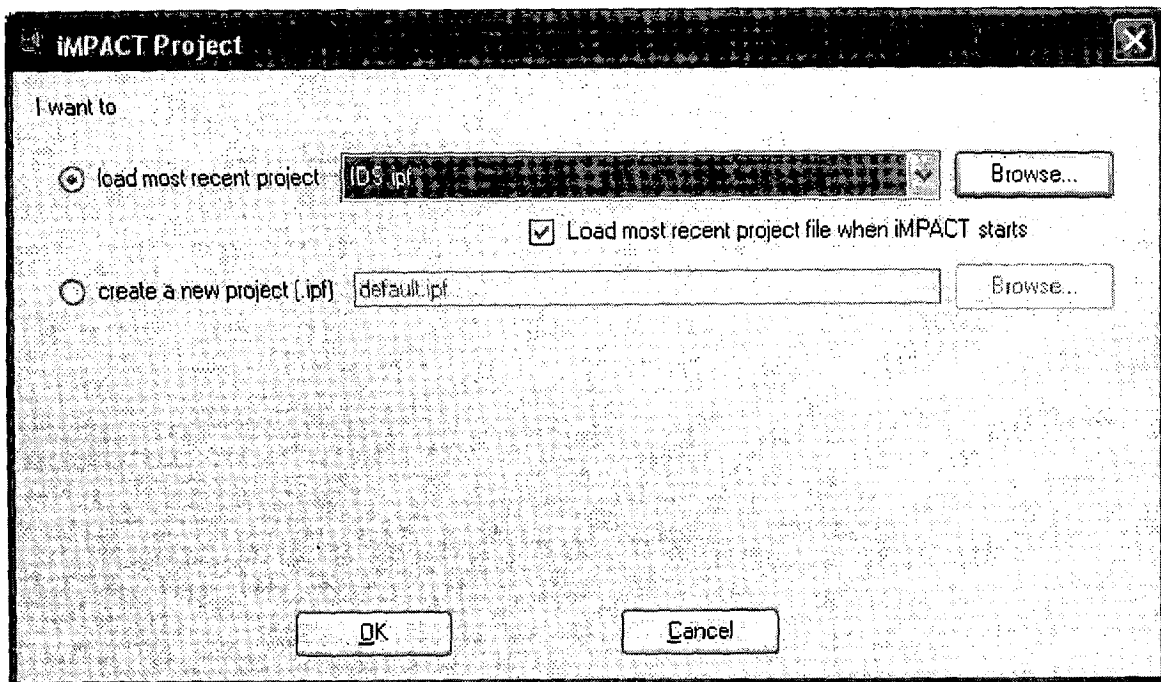


Fig 4.4 Loading your project in IMPACT



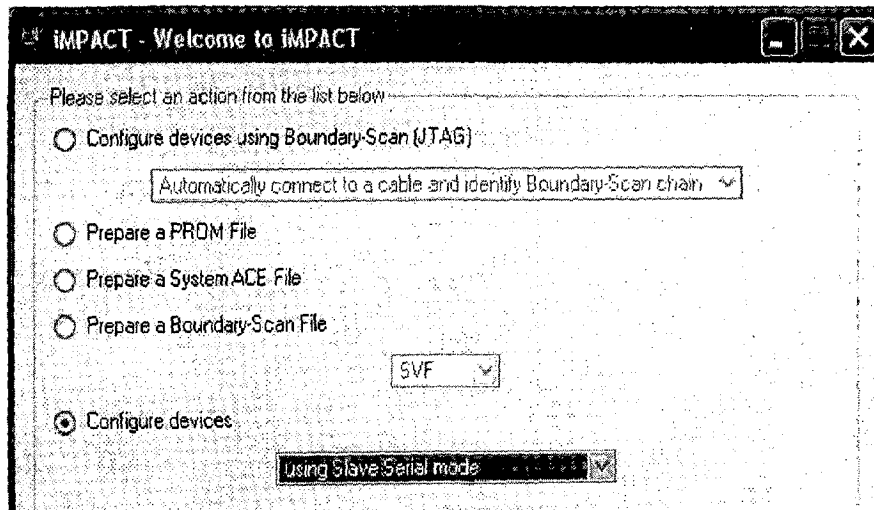


Fig 4.5 Options for configuring devices through IMPACT.

### Platform Cable USB

Platform Cable USB is a high-performance download cable that attaches to your hardware for the purpose of programming or configuring any of the following Xilinx® devices:

- ISP Configuration PROMs
- CPLDs
- FPGAs

Platform Cable USB attaches to a desktop or laptop PC with an off-the-shelf High-Speed USB A-B cable and derives all operating power from the hub port controller. No external power supply is required.

### 4.2. Embedded Development Kit(EDK) and Xilinx Platform Studio (XPS)

EDK [17] is an integrated software solution for designing embedded processing systems and implementing on a Xilinx FPGA device. The components of the Xilinx EDK are:

- Hardware (Intellectual Property) for the Xilinx Embedded Processors and their peripherals.
- Drivers, Libraries and a Micro Kernel for Embedded Software Development.
- Software Development Kit (SDK), Eclipse based IDE.
- GNU compiler and debugger for C development for MicroBlaze and PowerPC.

## **Xilinx Platform Studio**

Xilinx Platform Studio (XPS) is the design development software provided in the Xilinx Embedded Development Kit (EDK). XPS consists of an interface and all the underlying tools needed to develop the hardware and software components of an embedded processor system. We can also perform system verification within the XPS environment.

## **Base System Builder**

The Base System Builder (BSB) automates basic hardware and software platform configuration tasks common to most processor designs. The BSB lets us pick from the peripherals available on that board, automatically match the FPGA pinout to the board, and create a completed platform and test application ready to download and run on the board. This gives a hardware platform to use as a starting point from which we can add more processors and peripherals if needed, including custom peripherals, using the tools provided in Xilinx Platform Studio (XPS).

In all cases, BSB lets us select the following system attributes:

- Processor type (MicroBlaze or PowerPC, depending on the selected target FPGA device)
- Processor and bus clock frequency (BSB automatically infers and configures a Digital Clock Manager (DCM) primitive when needed)
- Standard processor buses (all peripherals are automatically connected via appropriate buses)
- Debug interface
- Cache configuration
- Memory size and type (both on-chip block RAM (BRAM) and controllers for off-chip memory devices)
- Common peripherals (such as general purpose I/O, Universal Asynchronous Receiver-Transmitter (UART), and timer)
- Interrupt sources (from among the applicable selected peripherals)

When targeting one of the supported embedded processor development boards, BSB narrows the choices of peripherals that control off-chip devices to those features provided on the board. Any deselected peripherals are omitted from the processor system design to minimize FPGA use. The BSB further provides the following board-specific services:

- Automatic selection of the on-board FPGA

- Selection of clock rates supported by the on-board oscillators
- Automatic setting of reset polarity
- Automatic generation of FPGA pinout to match the board connections, for the selected set of peripherals

For each option, functional default values are pre-selected in XPS. Upon exit of the BSB, a Microprocessor Hardware Specification (MHS) file is created and loaded into the XPS project. We can further enhance the design in XPS or continue to implement the design using the Xilinx implementation tools.

Optionally, the BSB can also create one or more software projects. Each project contains a sample application and linker script that can be compiled and run on the hardware on the target development board. Applications are designed to illustrate system aliveness and perform simple and basic testing of some hardware components. XPS supports multiple software projects for every hardware system, each of which contains its own source files and linker script.

The major steps carried out for the implementation of design on ML403 Evaluation Platform in XPS are given in Fig 4.6 to 4.17:

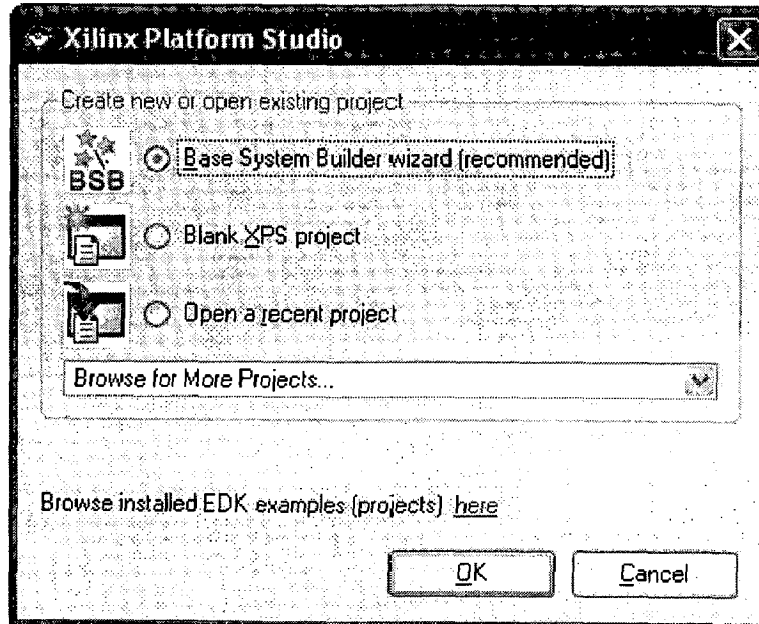


Fig 4.6 Selecting BSS

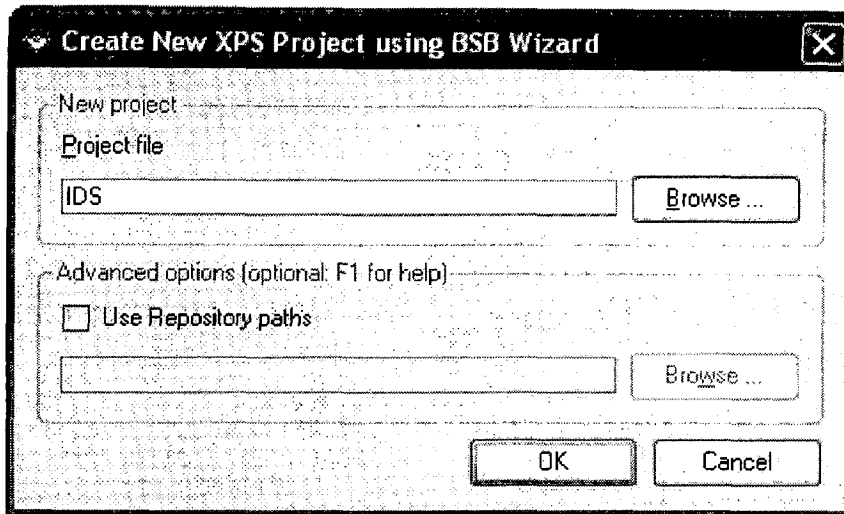


Fig 4.7 Creating new project- IDS

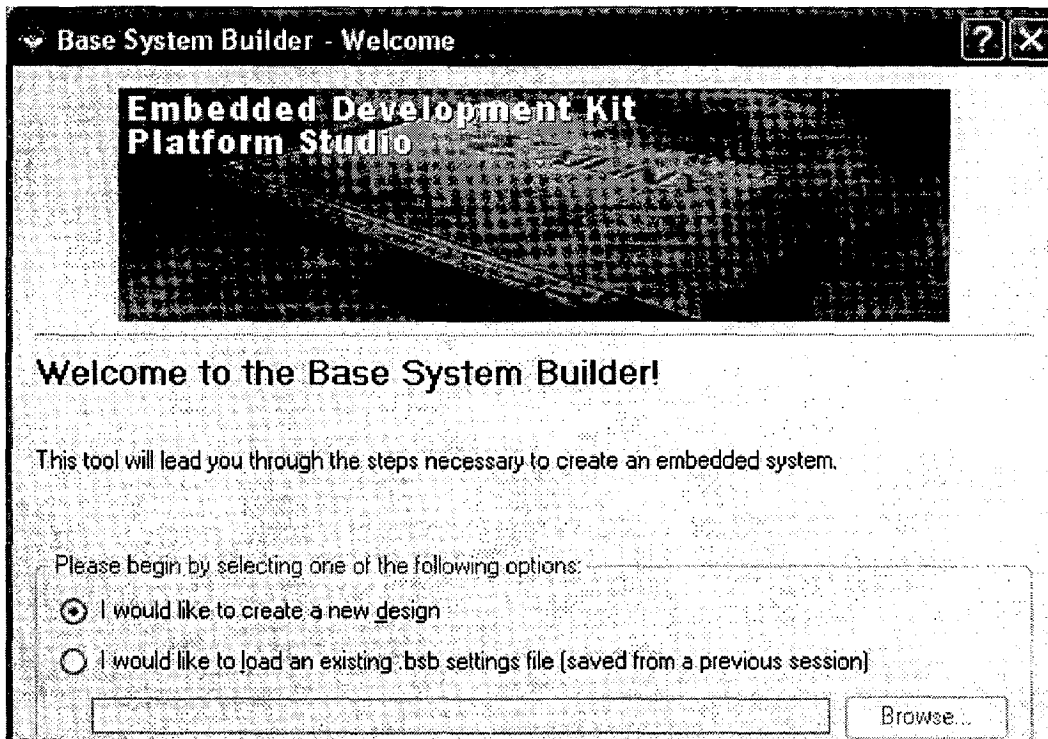


Fig 4.8 Selecting the design option

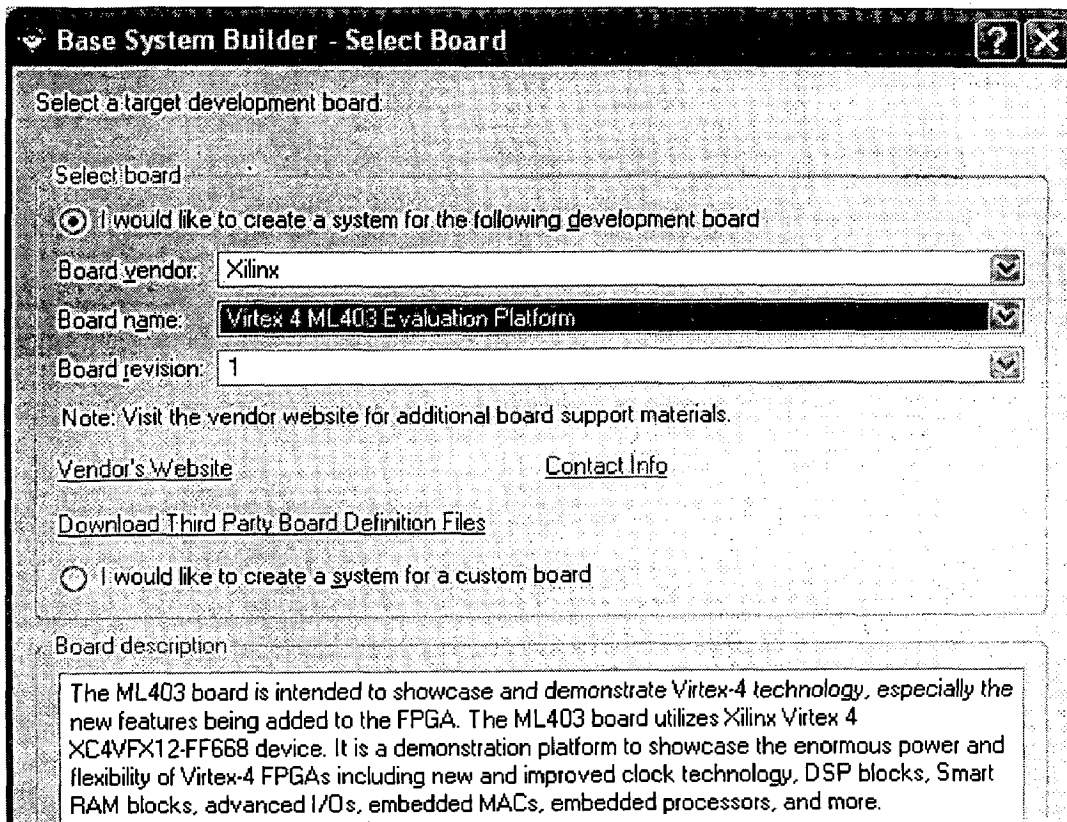


Fig 4.9 Selecting the target development board

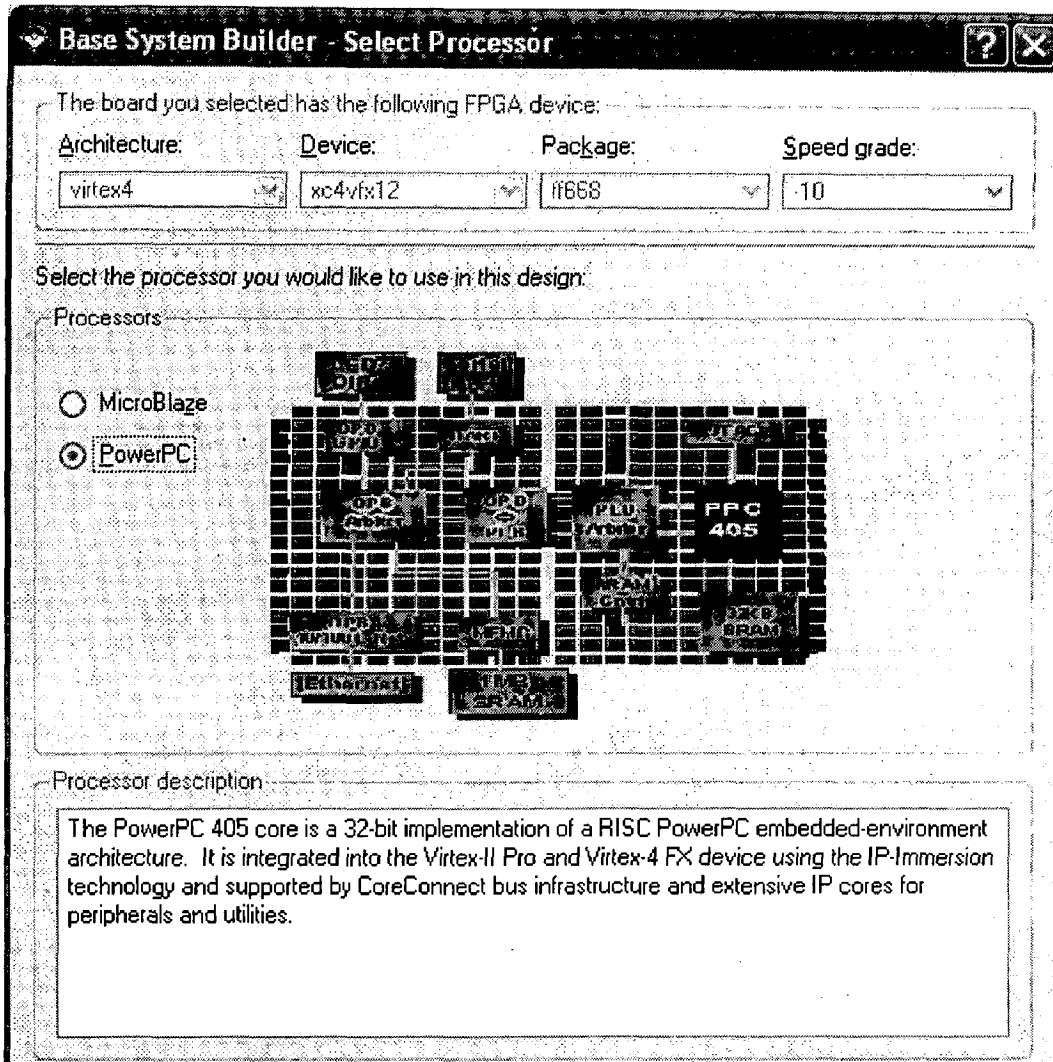


Fig 4.10 Selecting the embedded processor

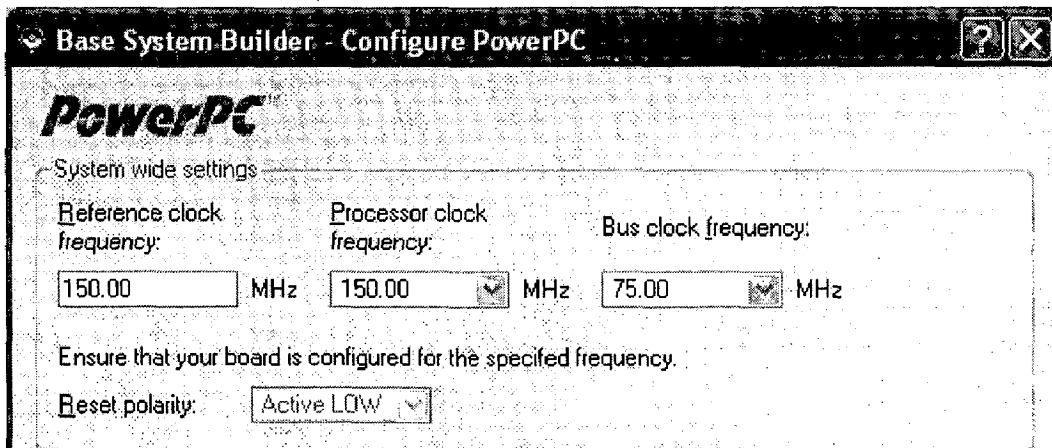


Fig 4.11 Selecting clock frequency

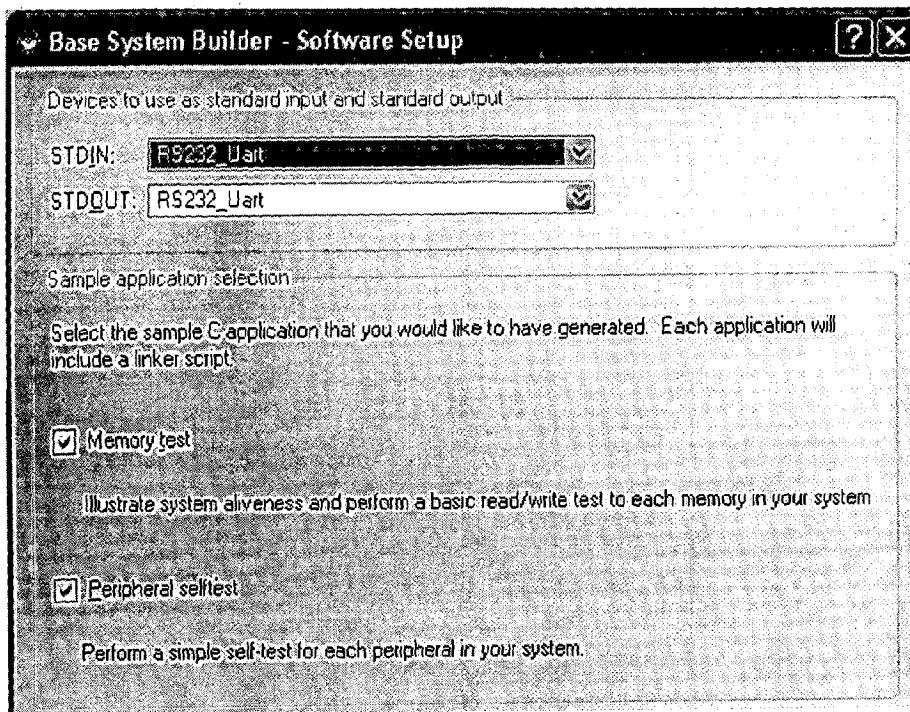


Fig 4.12 Software setup

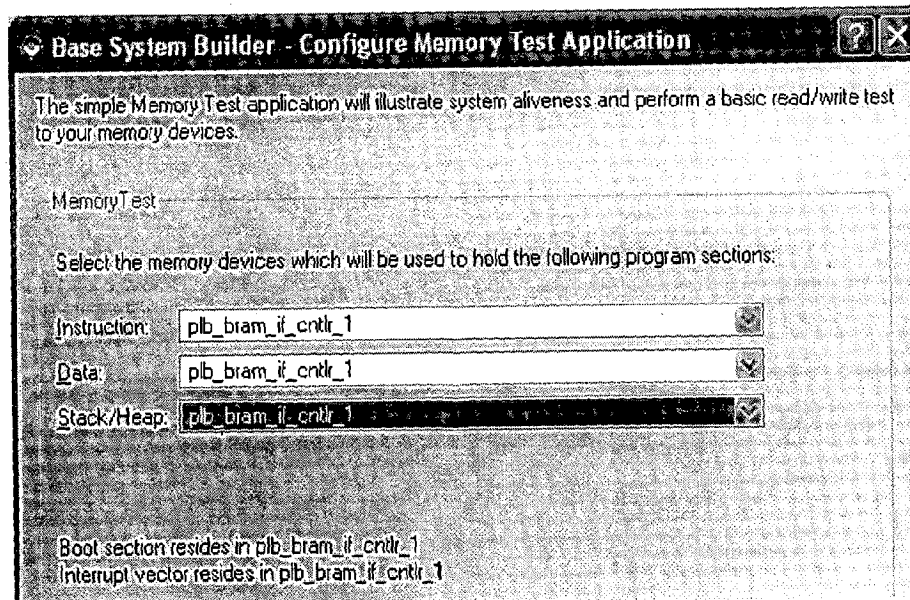


Fig 4.13 Configuring memory test applications

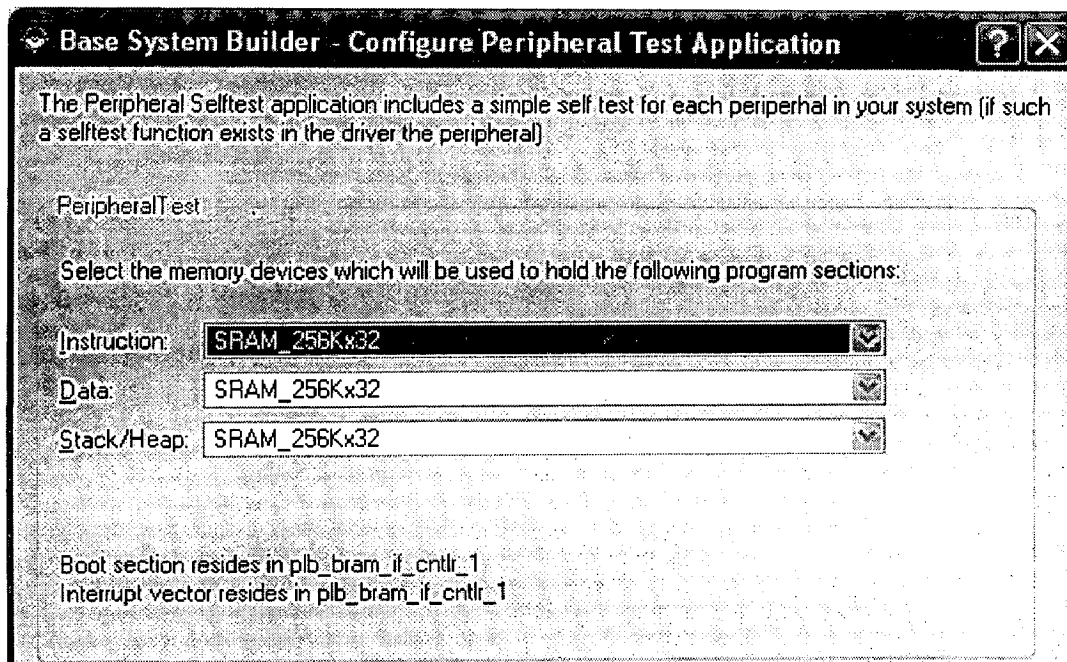


Fig 4.14 Configuring peripheral test applications

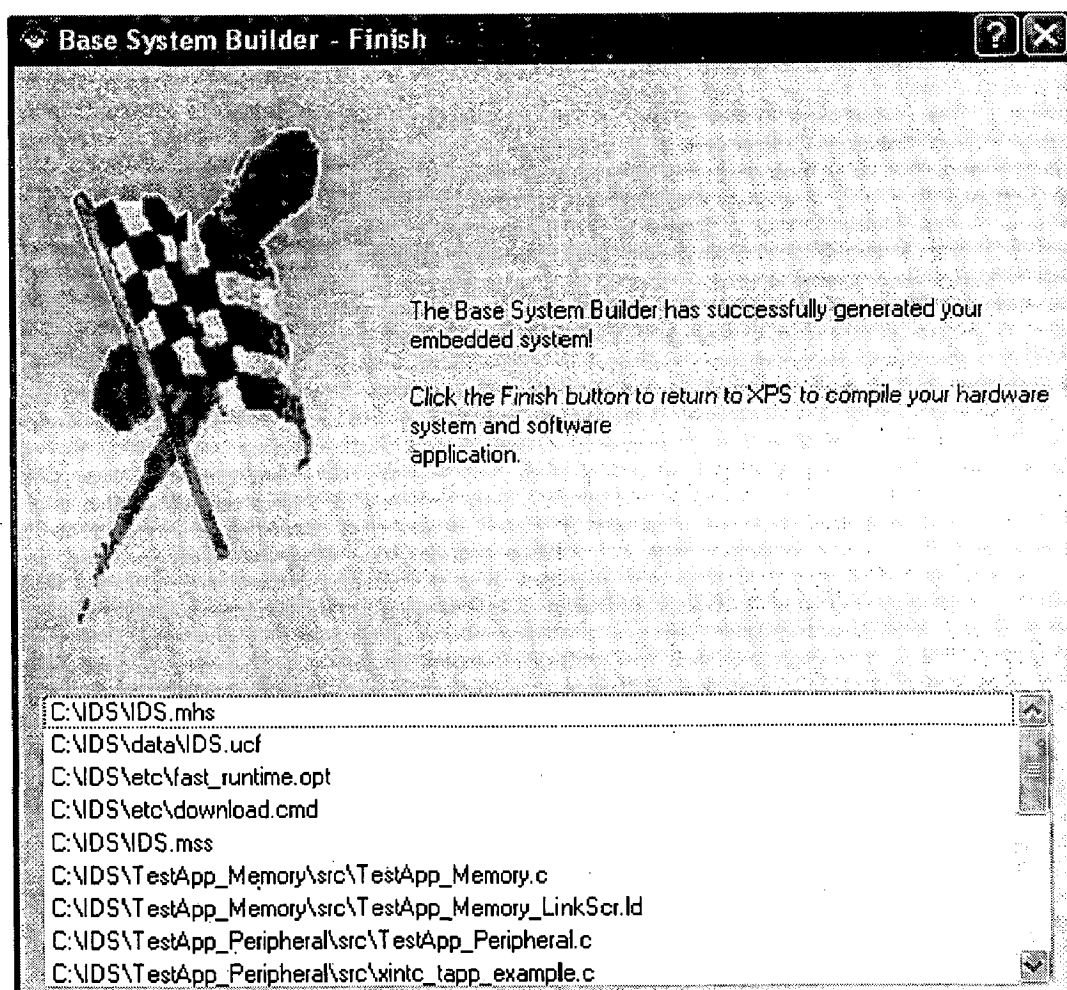


Fig 4.15 Finished message



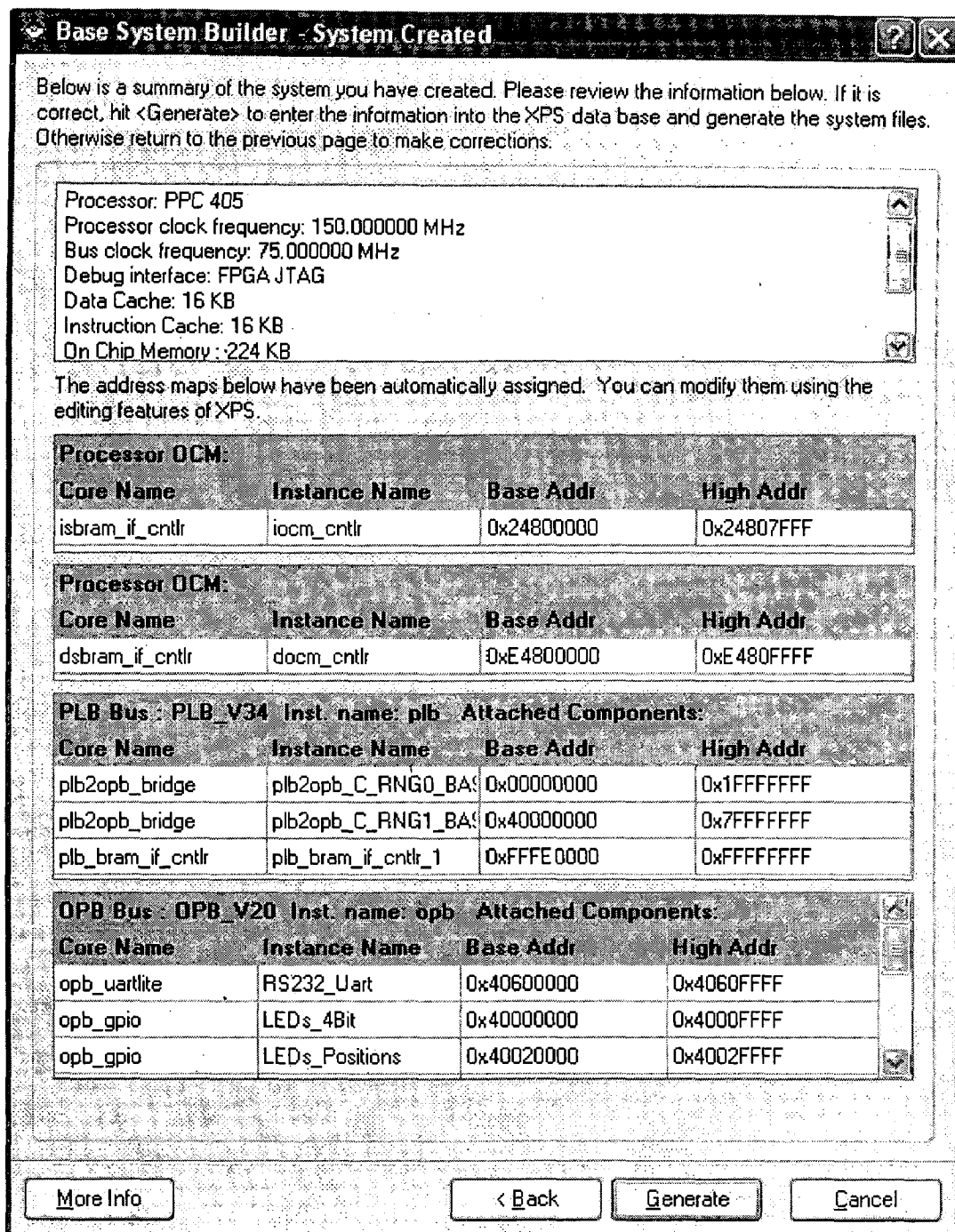


Fig 4.16 Details of System created

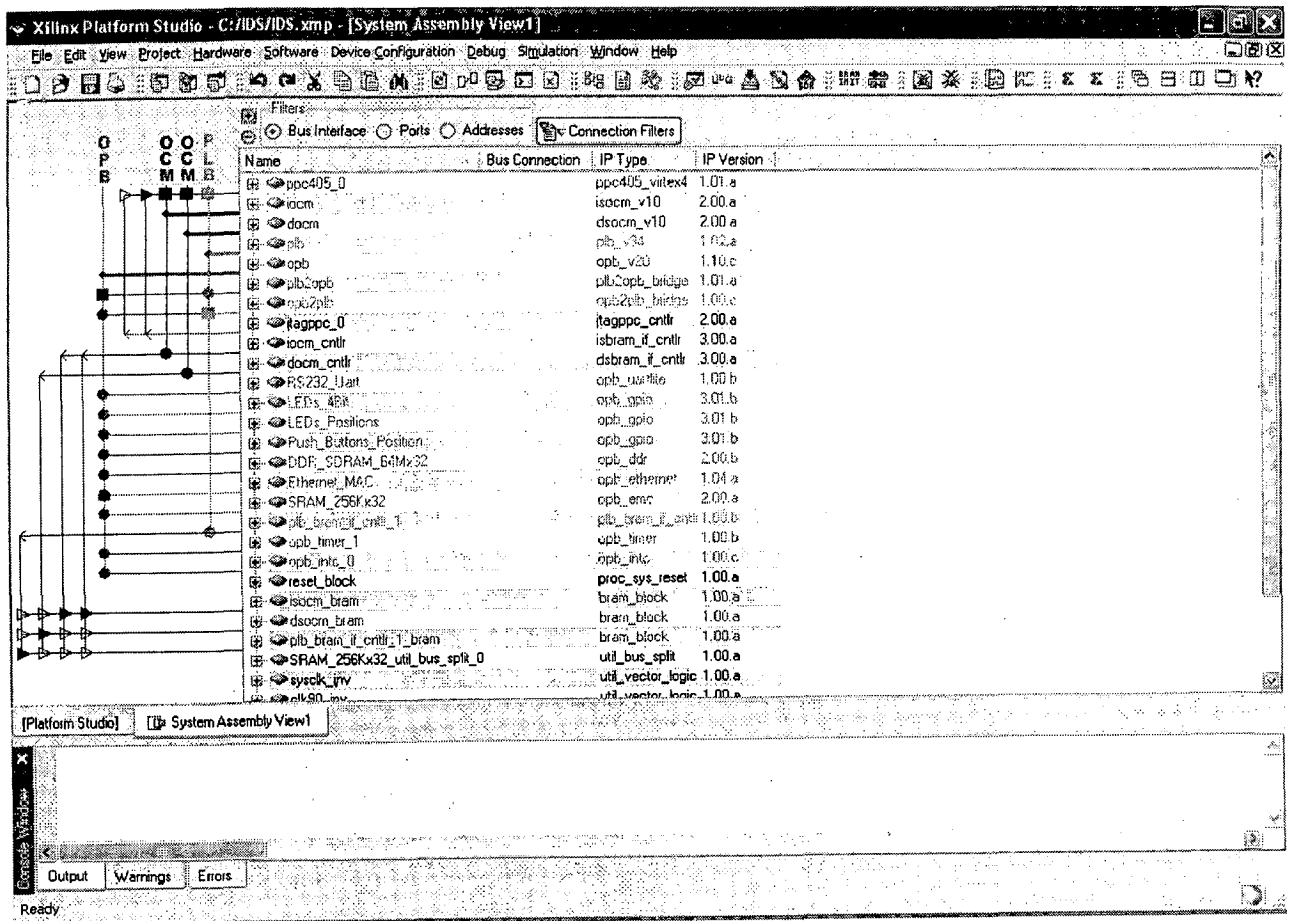


Fig 4.17 System Assembly View

### 4.3. ModelSim

ModelSim is a simulation and debugging tool for VHDL, Verilog, and mixed-language designs. The following diagram shows the basic steps for simulating a design within a ModelSim project.

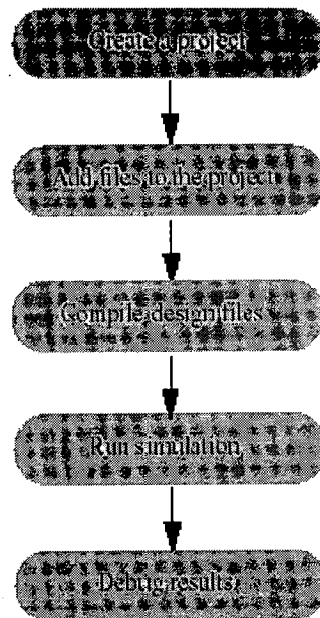


Fig 4.18 Steps for simulating a design with ModelSim

ModelSim offers numerous tools for debugging and analyzing the design. Several of these tools are:

- Setting breakpoints and stepping through the source code
- Viewing waveforms and measuring time
- Viewing and initializing memories

### 5.1. Lab Setup

The setup to evaluate performance of the designed and implemented IDS is as shown in Fig 5.1.

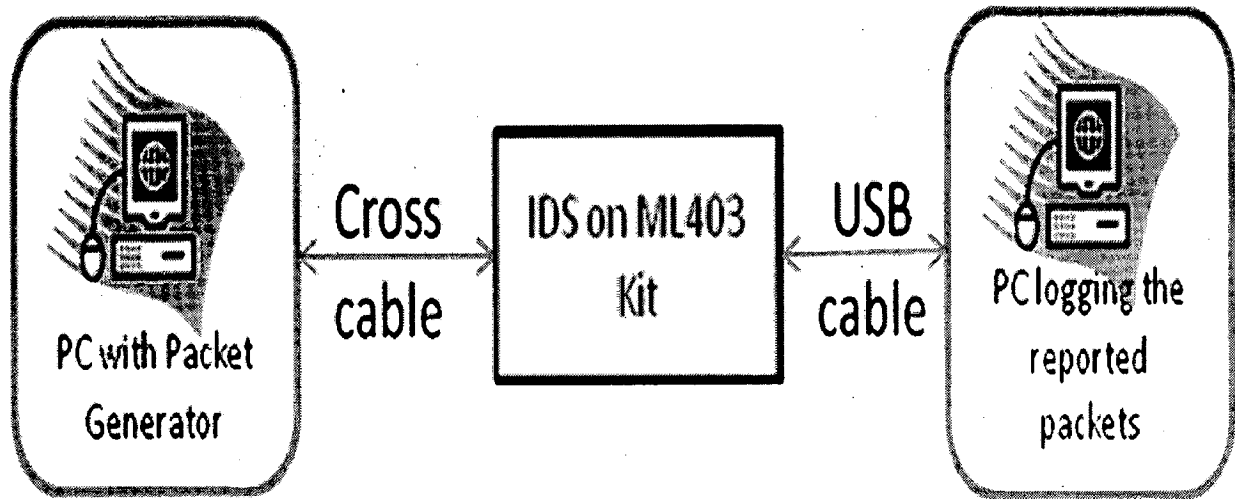


Fig 5.1. Lab setup for evaluating the IDS.

An IP packet generator is used to generate the network packets with desired source and destination IP address and port numbers. Only TCP protocol is used for checking the results.

After analysing Snort ruleset we have noted that majority of rulesets have width of 32 bytes, as shown in Fig 5.2. Table 5.1 shows example header ruleset. For this we have designed the TCAM for 32 bytes of data entry width. Similarly other rulesets can be catered for by TCAMs working in parallel with adequate data width.

Table 5.1: Example Header Rule Set

ID	Source IP	Destination IP	Protocol	Source Port	Destination Port
1	any	192.168.0.0/16	tcp	$\geq 1024$	2589
2	any	192.158.0.0/16	tcp	10101	any
3	any	192.168.50.2	tcp	any	443
4	192.168.0.0/16	any	udp	49230	60000
5	any	any	tcp	any	110
6	any	any	tcp	146	1000:1300

The TCAM output is used to enable the transmission of buffered packet's IP address for logging through serial interface to the PC maintaining performance log of this IDS.

The system compares and logs all packets while working on Gigabit Ethernet. The incoming packet rate is limited due to the physical Ethernet interface chosen as 1Gbps Ethernet interface. But to compare with other works in the field the calculated throughput of TCAM (based on SRL16 with 32 byte width) is given in Table 5.2. In a real world system not all of the traffic entering the system will need to be searched.

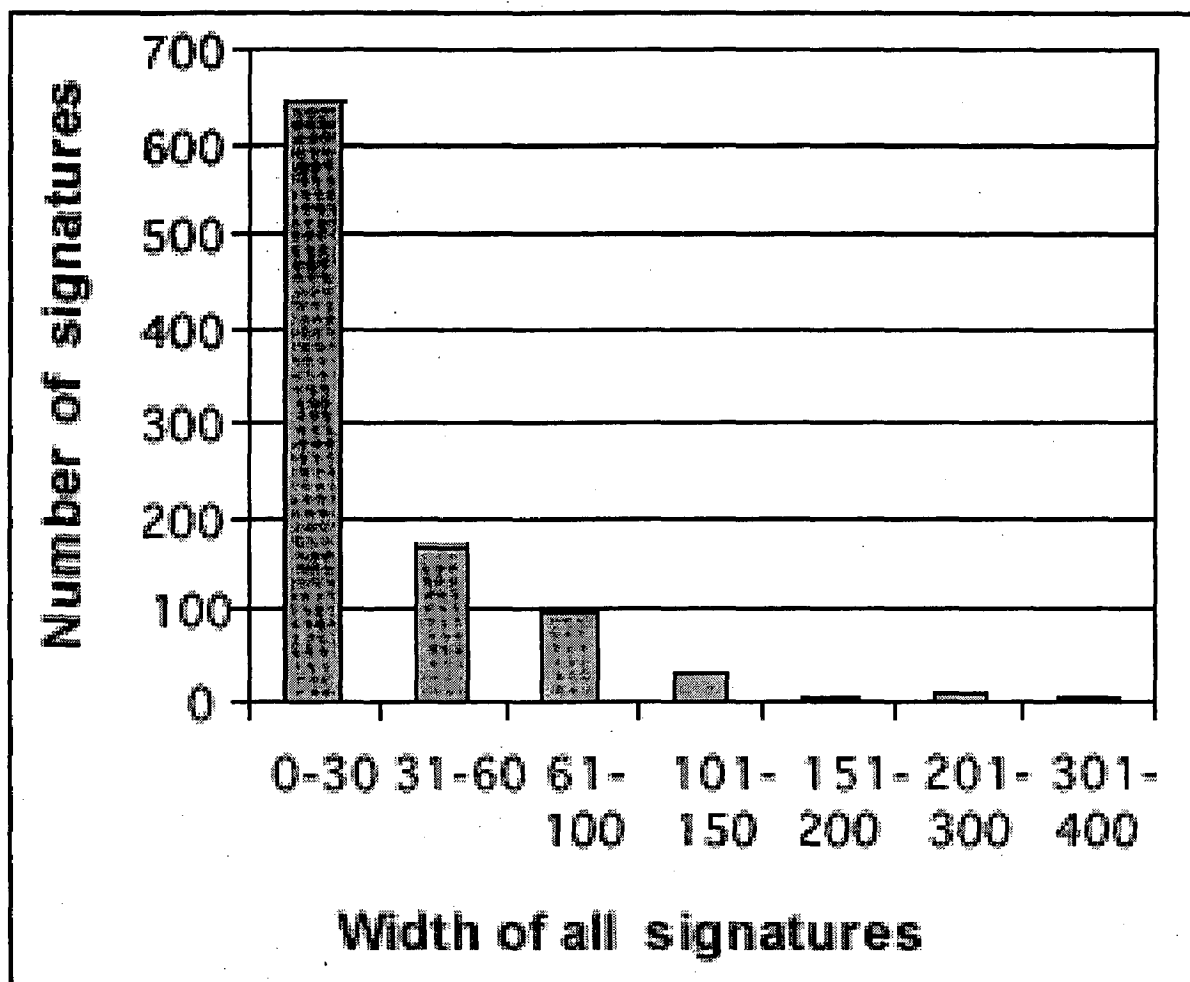


Fig 5.2. Signature width for all rules in Snort database.

In addition to the gigabit speed performance, the area left on the FPGA is sufficient to cater for content matching rulesets to be incorporated on the same kit.

## 5.2. Resource Utilization

Xilinx tools gives summary of each IP core generated or used, as given in Fig 5.3 for the Gigabit Ethernet MAC, which utilized maximum resources.

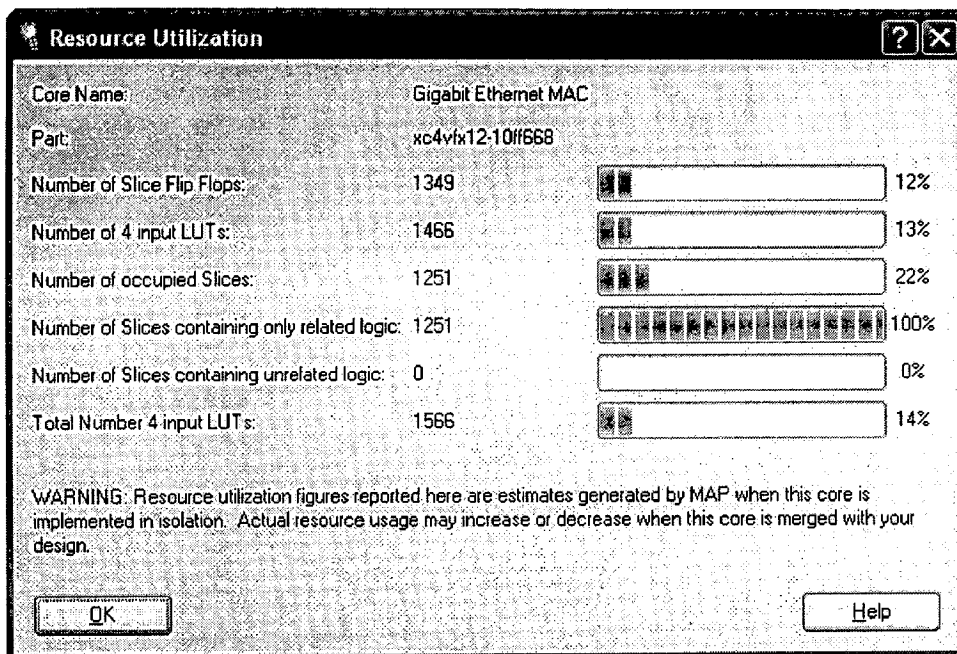


Fig 5.3.Resource utilization of Gigabit Ethernet MAC.

Finally when the complete design is ready and compiled using ISE Project Navigator, total resource utilization is given as part of summary report. The over all resource utilization for our design is only 34% (Though only header matching is taken into account, the area left is sufficient to add payload matching modules.)

## 5.3. Traffic Analysis

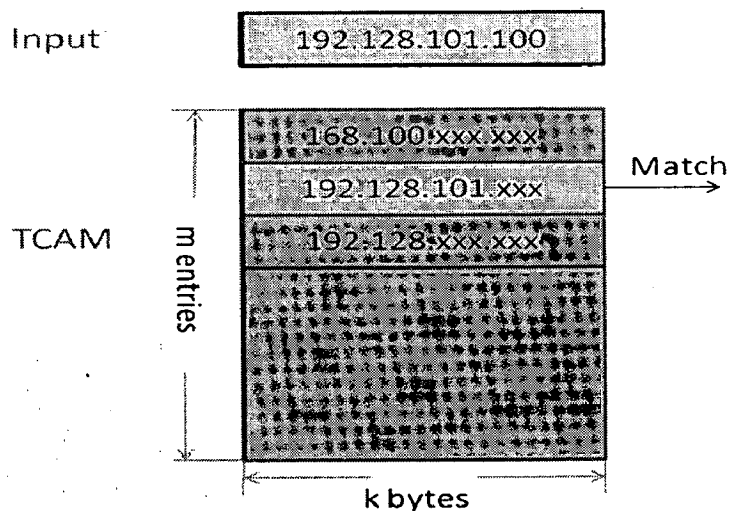


Fig 5.4. TCAM

As shown in Fig 5.4 the TCAM gives the index and match flag to the first match in the memory. TCAMs have Fully associative memory and compares input string with all the entries in parallel. If multiple matches, report index of the first match. Each cell takes one of three logic states : ‘0’, ‘1’, and ‘x’(don’t care). Current TCAM technology gives us fast time match up to 4ns. For 32 byte wide TCAM with depth of 1024 minimum cycle time is 16ns [17].

Table 5.2. Calculated Throughput of the Individual FPGA Modules.

Module	Throughput(maximum in Gbps))
Ruleset matching module(TCAM)	16
Gigabit Ethernet MAC	1

Gokhale et.al [24] used CAM to implement snort rules on a Virtex XCV1000E FPGA. Their hardware delivered a throughput of 2.2Gbps. Cho et. al [25] generated structural VHDL for deep packet filtering on an FPGA. Their design runs at 90MHz on an Altera EP20K device and achieves a throughput of 2.88Gbps. Attig et. al.[26] have implemented a Bloom filter circuit on a Virtex E2000 FPGA. Their circuit operates at 62.8MHz and provides a throughput of 502Mbps.

**Affect of Packet size**

Since there is a large overhead for processing each packet’s header, the biggest influence on receive performance is the packet size, which determines the number of packet arrivals per second. As the packet size reduces the performance of the system reduces. Appreciable affects are below 512 byte size packets, worst being at 64 byte size packets.

We have presented a design and implementation of complete Intrusion Detection System (IDS) on a FPGA. It is noted that with improved TCAMs with current technology there is a fast time match up to 4ns. The designed IDS on Virtex4 kit successfully works on Gigabit Ethernet.

The Protocol Wrapper including the packet extractor which extracts TCP/IP header bytes and forwards them for further processing by header ruleset matching module works for packets of varying length.

The ruleset in the TCAM on the kit can be updated through configuration memory while the system is running and this configuration memory is programmed through JTAG cable thus providing partial reconfigurable feature to the IDS on the FPGA kit. We have included only header ruleset for intrusion detection; similarly TCAM can be used for content matching rulesets.

The system presently logs only IP addresses of the defaulting packets through serial interface on a separate computer, it can be extended to store complete defaulting packet for further analysis.

If a kit with at least two Ethernet ports is used then additional feature of real time intrusion prevention can also be designed with this IDS. We have implemented the IDS using the FX-12 kit which provides 12,312 logic cells within 5,472 slices. The Virtex-4 FX-20 is the FPGA next in size providing 82% more LUTs than the one we used. With this FPGA it will be possible to implement the complete Intrusion Detection and Prevention system on one kit.



## REFERENCES

---

- [1] A. K. Tummala and P. Patel, "Distributed IDS using Reconfigurable Hardware", *In Proceedings of 21st International Parallel and Distributed Processing Symposium, (IPDPS-2007)*, pages 1-6, 26-30 March 2007.
- [2] Paul E. Proctor. "The Practical Intrusion Detection Handbook". *Prentice Hall*, 2000.
- [3] Snort Homepage. Available at <http://www.snort.org>, last accessed on June, 2008.
- [4] R. Franklin, D. Carver and B. L. Hutchings, "Assisting Network Intrusion Detection with Reconfigurable Hardware", *In Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 111, 22 – 24 September 2002.
- [5] L. Schaelicke, T. Slabach, B. Moore, and C. Freeland, "Characterizing the Performance of Network Intrusion Detection Sensors", *In Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection (RAID-2003)*, pages 1-19, September 2003.
- [6] S. Li, J. Torresen and O. Soraasen, "Exploiting Reconfigurable Hardware for Network Security", *In Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 292, 09 – 11 April 2003.
- [7] D. Moore, C. Shannon, G. M. Voelker and S. Savage, "Internet Quarantine: Requirements for Containing Self-Propagating Code", *In Proceedings of 22nd Annual Joint Conference of IEEE Computer and Communication societies (INFOCOM 2003)*, volume 3, pages 1901- 1910 , 30 March - 3 April 2003.
- [8] E. Ahmed, and J. Rose, "The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density", *In Proceedings of the ACM/SIGDA Eighth international Symposium on Field Programmable Gate Arrays*, pages 3-12, 10 – 11 February 2000.
- [9] J. Cong and Y. Ding, "Combinational logic synthesis for LUT based field programmable gate arrays", *ACM Trans. Des. Autom. Electron. Syst*, volume 1, issue 2, pages 145– 204, 1996.
- [10] IBM coreconnect architecture, available at <http://www.ibm.com/chips/products/coreconnect> last accessed on June, 2008
- [11] K. Compton and S. Hauck, "An Introduction to Reconfigurable Computing", *In IEEE Computer*, April 2000.
- [12] A. S. Tanenbaum, "Computer Networks", 4<sup>th</sup> edition, *Prentice-Hall, Inc*, 2002.
- [13] J. F. Kurose and K. Ross, "Computer Networking: A Top-Down Approach Featuring the Internet", 2<sup>nd</sup> edition *Addison-Wesley Longman Publishing Co., Inc*, 2003.
- [14] D. Comer, "Internetworking with TCP/IP: Principles, Protocols, and Architecture", *Prentice-Hall, Inc*, 1998.
- [15] L. Qiu, G. Varghese, and S. Suri, "Fast Firewall Implementations for Software-Based and Hardware-Based Routers", *In SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS*

*International Conference on Measurement and Modeling of Computer Systems*, pages 344-345, 2001.

[16] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time", *Computer Networks, Amsterdam, Netherlands*, pages 2435 – 2463, 1999.

[17] The Xilinx Corporation, Web reference [www.xilinx.com](http://www.xilinx.com), last accessed on June, 2008.

[18] Treck Inc, Web reference, <http://www.treck.com>, last accessed on June, 2008.

[19] F. Braun, J. Lockwood, and M. Waldvogel, "Protocol Wrappers for Layered Network Packet Processing in Reconfigurable Hardware," *IEEE Micro*, volume 22, issue 1, pages 66-74, Jan.-Feb. 2002.

[20] Embedded System Tools Reference Manual, available at [http://www.xilinx.com/ise/embedded/est\\_rm.pdf](http://www.xilinx.com/ise/embedded/est_rm.pdf) Last accessed on June, 2008

[21] F. Yu and R. Katz. "Efficient Multi-Match Packet Classification and Lookup with TCAM", In *12th Annual Proceedings of IEEE Hot Interconnects, Stanford, CA*, pages 50-59, August 2004.

[22] Ethernet statistics IP core, available at [http://www.xilinx.com/ML403/ethernet\\_statistics\\_ds323.pdf](http://www.xilinx.com/ML403/ethernet_statistics_ds323.pdf). last accessed on June, 2008

[23] ML40x EDK Processor Reference Design, available at <http://www.xilinx.com/bvdocs/userguides/ug082.pdf> last accessed on June, 2008.

[24] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards Gigabit Rate Network Intrusion Detection Technology", In *Proceedings of International Conference on Field Programmable Logic and Applications*, pages 404–413, 2002.

[25] Y. H. Cho, and W. M. Smith, "Specialized Hardware for Deep Packet Filtering", In *Proceedings of the 12th International Conference on Field Programmable Logic and Applications*, pages 452-461, September 2002.

[26] M. Attig, S. Dharmapurikar and J. Lockwood, "Implementation Results of Bloom Filters for String Matching", In *proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, pages 322-323, 2004.

[27] C. Kruegel and G. Vigna, "Anomaly Detection of Web-Based Attacks" In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 251-261, 2003.

[28] B. Madhusudan and J. Lockwood "Design of a System for Real-Time Worm Detection", In *12th Annual Proceedings of IEEE Hot Interconnects*, pages 77-83, August 2004.

## **PUBLICATIONS**

---

Submitted paper titled “Design and Implementation of Intrusion Detection System (IDS) with Field Programmable Gate Array (FPGA)” in the IEEE Colloquium and the third IEEE International Conference on Industrial and Information Systems (ICIIS 2008) to be held at Indian Institute of Technology, Kharagpur, India during 08 – 10 December 2008. The paper submission no is 293.

---

```
//Vendor: Xilinx
//Version : 9.2i
//Filename : tcam.v
Timestamp : 04/21/2008 10:00:27

//
//Design Name: tcam
//Device: xc4vfx12-10ff668
//
// Module tcam
// Generated by Xilinx Architecture Wizard
//

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
-- synthesis translate_off
Library XilinxCoreLib;
-- synthesis translate_on
ENTITY tcam IS
    port (
        clk: IN std_logic;
        data_mask: IN std_logic_VECTOR(31 downto 0);
        din: IN std_logic_VECTOR(31 downto 0);
        we: IN std_logic;
        wr_addr: IN std_logic_VECTOR(7 downto 0);
        busy: OUT std_logic;
        match: OUT std_logic;
        match_addr: OUT std_logic_VECTOR(255 downto 0);
        single_match: OUT std_logic);
END tcam;

ARCHITECTURE tcam_a OF tcam IS
-- synthesis translate_off
component wrapped_tcam
    port (
        clk: IN std_logic;
        data_mask: IN std_logic_VECTOR(31 downto 0);
        din: IN std_logic_VECTOR(31 downto 0);
        we: IN std_logic;
        wr_addr: IN std_logic_VECTOR(7 downto 0);
        busy: OUT std_logic;
        match: OUT std_logic;
        match_addr: OUT std_logic_VECTOR(255 downto 0);
        single_match: OUT std_logic);
end component;

-- Configuration specification
for all : wrapped_tcam use entity XilinxCoreLib.cam_v5_1(behavioral)
generic map(
    c_has_en => 0,
    c_wr_addr_width => 8,
    c_din_width => 32,
    c_has_wr_addr => 1,
    c_data_mask_width => 32,
    c_cmp_din_width => 32,
```

```

        c_has_read_warning => 0,
        c_width => 32,
        c_mem_init => 0,
        c_has_cmp_data_mask => 0,
        c_has_we => 1,
        c_enable_rlocs => 0,
        c_addr_type => 1,
        c_ternary_mode => 1,
        c_match_resolution_type => 0,
        c_has_single_match => 1,
        c_depth => 256,
        c_has_multiple_match => 0,
        c_read_cycles => 1,
        c_mem_type => 0,
        c_has_data_mask => 1,
        c_reg_outputs => 0,
        c_has_cmp_din => 0,
        c_mem_init_file => "o.mif",
        c_cmp_data_mask_width => 32,
        c_match_addr_width => 256);

-- synthesis translate_on
BEGIN
-- synthesis translate_off
U0 : wrapped_team
    port map (
        clk => clk,
        data_mask => data_mask,
        din => din,
        we => we,
        wr_addr => wr_addr,
        busy => busy,
        match => match,
        match_addr => match_addr,
        single_match => single_match);

-- synthesis translate_on

END team_a;

```

```

//Vendor: Xilinx
//Version : 9.2i
//Filename : eth.v
Timestamp : 04/28/2008 10:50:07

```

```

//
//Design Name: eth
//Device: xc4vfx12-10ff668
//
// Module eth
// Generated by Xilinx Architecture Wizard

```

```

module eth_clk(CLKIN_IN,
    RST_IN,
    CLKIN_IBUFG_OUT,
    CLK0_OUT,
    LOCKED_OUT);

```

```

input CLKIN_IN;
input RST_IN;
output CLKIN_IBUFG_OUT;
output CLK0_OUT;
output LOCKED_OUT;

wire CLKFB_IN;
wire CLKIN_IBUFG;
wire CLK0_BUF;
wire [6:0] GND1;
wire [15:0] GND2;
wire GND3;

assign GND1 = 7'b0000000;
assign GND2 = 16'b0000000000000000;
assign GND3 = 0;
assign CLKIN_IBUFG_OUT = CLKIN_IBUFG;
assign CLK0_OUT = CLKFB_IN;
IBUFG CLKIN_IBUFG_INST (.I(CLKIN_IN),
    .O(CLKIN_IBUFG));
BUFG CLK0_BUF_INST (.I(CLK0_BUF),
    .O(CLKFB_IN));
DCM_ADV DCM_ADV_INST (.CLKFB(CLKFB_IN),
    .CLKIN(CLKIN_IBUFG),
    .DADDR(GND1[6:0]),
    .DCLK(GND3),
    .DEN(GND3),
    .DI(GND2[15:0]),
    .DWE(GND3),
    .PSCLK(GND3),
    .PSEN(GND3),
    .PSINCDEC(GND3),
    .RST(RST_IN),
    .CLKDV(),
    .CLKFX(),
    .CLKFX180(),
    .CLK0(CLK0_BUF),
    .CLK2X(),
    .CLK2X180(),
    .CLK90(),
    .CLK180(),
    .CLK270(),
    .DO(),
    .DRDY(),
    .LOCKED(LOCKED_OUT),
    .PSDONE());
defparam DCM_ADV_INST.CLK_FEEDBACK = "1X";
defparam DCM_ADV_INST.CLKDV_DIVIDE = 2.0;
defparam DCM_ADV_INST.CLKFX_DIVIDE = 1;
defparam DCM_ADV_INST.CLKFX_MULTIPLY = 4;
defparam DCM_ADV_INST.CLKIN_DIVIDE_BY_2 = "FALSE";
defparam DCM_ADV_INST.CLKIN_PERIOD = 6.66667;
defparam DCM_ADV_INST.CLKOUT_PHASE_SHIFT = "NONE";
defparam DCM_ADV_INST.DCM_AUTO CALIBRATION = "TRUE";
defparam DCM_ADV_INST.DCM_PERFORMANCE_MODE = "MAX_SPEED";
defparam DCM_ADV_INST.DESKEW_ADJUST = "SYSTEM_SYNCHRONOUS";
defparam DCM_ADV_INST.DFS_FREQUENCY_MODE = "LOW";
defparam DCM_ADV_INST.DLL_FREQUENCY_MODE = "LOW";
defparam DCM_ADV_INST.DUTY_CYCLE_CORRECTION = "TRUE";
defparam DCM_ADV_INST.FACTORY_JF = 16'hF0F0;

```

```

defparam DCM_ADV_INST.PHASE_SHIFT = 0;
defparam DCM_ADV_INST.STARTUP_WAIT = "FALSE";
endmodule

```

```

module eth_stat (
    host_stats_lsw_rdy, host_miim_sel, rx_small, host_reset, ref_reset, rx_frag, host_req, rx_byte,
    host_stats_msw_rdy, tx_byte, rx_clk, rx_reset,
    tx_reset, ref_clk, tx_clk, host_clk, host_addr, host_rd_data, increment_vector
);
    output host_stats_lsw_rdy;
    input host_miim_sel;
    input rx_small;
    input host_reset;
    input ref_reset;
    input rx_frag;
    input host_req;
    input rx_byte;
    output host_stats_msw_rdy;
    input tx_byte;
    input rx_clk;
    input rx_reset;
    input tx_reset;
    input ref_clk;
    input tx_clk;
    input host_clk;
    input [9 : 0] host_addr;
    output [31 : 0] host_rd_data;
    input [4 : 19] increment_vector;

    // The synopsys directives "translate_off/translate_on" specified
    // below are supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity
    // synthesis tools. Ensure they are correct for your synthesis tool(s)

    // synopsys translate_off

    wire NlwRenamedSig_OI_host_stats_lsw_rdy;
    wire \BU2/N3 ;
    wire \BU2/N2 ;
    wire \BU2/U0/ethernet_statistics_32bit/N7 ;
    wire \BU2/U0/N987 ;
    wire \BU2/U0/N986 ;
    wire \BU2/U0/N978 ;
    wire \BU2/U0/N975 ;
    wire \BU2/U0/N974 ;
    wire \BU2/U0/ethernet_statistics_32bit/_mux0058_map135 ;
    wire \BU2/U0/N973 ;
    wire \BU2/U0/N972 ;
    wire \BU2/U0/N971 ;
    wire \BU2/U0/N970 ;
    wire \BU2/U0/ethernet_statistics_32bit/_mux0059_map240 ;
    wire \BU2/U0/N969 ;
    wire \BU2/U0/N968 ;
    wire \BU2/U0/N967 ;
    wire \BU2/U0/N966 ;
    wire \BU2/U0/ethernet_statistics_32bit/count_read_0_1_2 ;
    wire \BU2/U0/N965 ;
    wire \BU2/U0/ethernet_statistics_32bit/_or0003_map313 ;
    wire \BU2/U0/N991 ;
    wire \BU2/U0/N964 ;

```

```

wire \BU2/U0/N963 ;
wire \BU2/U0/N962 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0059 ;
wire \BU2/U0/N961 ;
wire \BU2/U0/N960 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0055 ;
wire \BU2/U0/N959 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0058 ;
wire \BU2/U0/N958 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0056 ;
wire \BU2/U0/N957 ;
wire \BU2/U0/N956 ;
wire \BU2/U0/N955 ;
wire \BU2/U0/N954 ;
wire \BU2/U0/N953 ;
wire \BU2/U0/N952 ;
wire \BU2/U0/N951 ;
wire \BU2/U0/N950 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0060 ;
wire \BU2/U0/N949 ;
wire \BU2/U0/ethernet_statistics_32bit/count_read_01 ;
wire \BU2/U0/N947 ;
wire \BU2/U0/N946 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0058_map155 ;
wire \BU2/U0/N941 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0055_map199 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0059_map227 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0059_map225 ;
wire \BU2/U0/N936 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<0>_map114 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<1>_map101 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<2>_map88 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<3>_map75 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<5>_map62 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<4>_map49 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<6>_map36 ;
wire \BU2/U0/ethernet_statistics_32bit/_or0007_map179 ;
wire \BU2/U0/ethernet_statistics_32bit/_or0007_map178 ;
wire \BU2/U0/ethernet_statistics_32bit/_or0007_map172 ;
wire \BU2/U0/ethernet_statistics_32bit/_or0007_map171 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0055_map215 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0062 ;
wire \BU2/U0/N929 ;
wire \BU2/U0/N927 ;
wire \BU2/U0/ethernet_statistics_32bit/_or0007_map181 ;
wire \BU2/U0/N926 ;
wire \BU2/U0/N925 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<6>_map41 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<5>_map67 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<4>_map54 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<3>_map80 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<2>_map93 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<1>_map106 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<0>_map119 ;
wire \BU2/U0/ethernet_statistics_32bit/_or0003 ;
wire \BU2/U0/ethernet_statistics_32bit/_or0003_map309 ;
wire \BU2/U0/N992 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0056_map292 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0055_map196 ;
wire \BU2/U0/ethernet_statistics_32bit/_or0007_map166 ;

```



```

wire \BU2/U0/ethernet_statistics_32bit/_or0007_map162 ;
wire \BU2/U0/ethernet_statistics_32bit/_or0007_map159 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0058_map141 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0058_map124 ;
wire \BU2/U0/ethernet_statistics_32bit/_xor0029 ;
wire \BU2/U0/N990 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<0>_map108 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<1>_map95 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<2>_map82 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<3>_map69 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<5>_map56 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<4>_map43 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0061<6>_map30 ;
wire \BU2/U0/ethernet_statistics_32bit/round_robin_sequence_3_1_3 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0057_map15 ;
wire \BU2/U0/ethernet_statistics_32bit/N26 ;
wire \BU2/U0/ethernet_statistics_32bit/_or0013 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0057_map10 ;
wire \BU2/U0/N989 ;
wire \BU2/U0/ethernet_statistics_32bit/_or0012 ;
wire \BU2/U0/ethernet_statistics_32bit/_mux0057_map8 ;
wire \BU2/U0/N988 ;
wire \BU2/U0/ethernet_statistics_32bit/round_robin_sequence_3_2_4 ;
wire \BU2/U0/ethernet_statistics_32bit/count_read_0_3_5 ;
wire \BU2/U0/N91 ;
wire \BU2/U0/N89 ;
wire \BU2/U0/N87 ;
wire \BU2/U0/N85 ;

```

assign

```

host_stats_lsw_rdy = NIwRenamedSig_OI_host_stats_lsw_rdy,
host_addr_142[9] = host_addr[9],
host_addr_142[8] = host_addr[8],
host_addr_142[7] = host_addr[7],
host_addr_142[6] = host_addr[6],
host_addr_142[5] = host_addr[5],
host_addr_142[4] = host_addr[4],
host_addr_142[3] = host_addr[3],
host_addr_142[2] = host_addr[2],
host_addr_142[1] = host_addr[1],
host_addr_142[0] = host_addr[0],
host_rd_data[31] = host_rd_data_143[31],
host_rd_data[30] = host_rd_data_143[30],
host_rd_data[29] = host_rd_data_143[29],
host_rd_data[28] = host_rd_data_143[28],
host_rd_data[27] = host_rd_data_143[27],
host_rd_data[26] = host_rd_data_143[26],
host_rd_data[25] = host_rd_data_143[25],
host_rd_data[24] = host_rd_data_143[24],
host_rd_data[23] = host_rd_data_143[23],
host_rd_data[22] = host_rd_data_143[22],
host_rd_data[21] = host_rd_data_143[21],
host_rd_data[20] = host_rd_data_143[20],
host_rd_data[19] = host_rd_data_143[19],
host_rd_data[18] = host_rd_data_143[18],
host_rd_data[17] = host_rd_data_143[17],
host_rd_data[16] = host_rd_data_143[16],
host_rd_data[15] = host_rd_data_143[15],
host_rd_data[14] = host_rd_data_143[14],
host_rd_data[13] = host_rd_data_143[13],

```

```

host_rd_data[12] = host_rd_data_143[12],
host_rd_data[11] = host_rd_data_143[11],
host_rd_data[10] = host_rd_data_143[10],
host_rd_data[9] = host_rd_data_143[9],
host_rd_data[8] = host_rd_data_143[8],
host_rd_data[7] = host_rd_data_143[7],
host_rd_data[6] = host_rd_data_143[6],
host_rd_data[5] = host_rd_data_143[5],
host_rd_data[4] = host_rd_data_143[4],
host_rd_data[3] = host_rd_data_143[3],
host_rd_data[2] = host_rd_data_143[2],
host_rd_data[1] = host_rd_data_143[1],
host_rd_data[0] = host_rd_data_143[0],
increment_vector_144[4] = increment_vector[4],
increment_vector_144[5] = increment_vector[5],
increment_vector_144[6] = increment_vector[6],
increment_vector_144[7] = increment_vector[7],
increment_vector_144[8] = increment_vector[8],
increment_vector_144[9] = increment_vector[9],
increment_vector_144[10] = increment_vector[10],
increment_vector_144[11] = increment_vector[11],
increment_vector_144[12] = increment_vector[12],
increment_vector_144[13] = increment_vector[13],
increment_vector_144[14] = increment_vector[14],
increment_vector_144[15] = increment_vector[15],
increment_vector_144[16] = increment_vector[16],
increment_vector_144[17] = increment_vector[17],
increment_vector_144[18] = increment_vector[18],
increment_vector_144[19] = increment_vector[19];
VCC VCC_0 (
.P(NLW_VCC_P_UNCONNECTED)
);
GND GND_1 (
.G(NLW_GND_G_UNCONNECTED)
);
VCC \BU2/XST_VCC (
.P(BU2/N3 )
);
GND \BU2/XST_GND (
.G(BU2/N2 )
);

FDR \BU2/U0/ethernet_statistics_32bit/host_rd_data_6 (
.D(\BU2/U0/ethernet_statistics_32bit/_mux0054 [6]),
.R(\BU2/U0/ethernet_statistics_32bit/_or0005 ),
.C(host_clk),
.Q(host_rd_data_143[6])
);
FDR \BU2/U0/ethernet_statistics_32bit/host_rd_data_5 (
.D(\BU2/U0/ethernet_statistics_32bit/_mux0054 [5]),
.R(\BU2/U0/ethernet_statistics_32bit/_or0005 ),
.C(host_clk),
.Q(host_rd_data_143[5])
);
FDR \BU2/U0/ethernet_statistics_32bit/host_rd_data_4 (
.D(\BU2/U0/ethernet_statistics_32bit/_mux0054 [4]),
.R(\BU2/U0/ethernet_statistics_32bit/_or0005 ),
.C(host_clk),
.Q(host_rd_data_143[4])
);

```

```

FDR \BU2/U0/ethernet_statistics_32bit/host_rd_data_3 (
.D(\BU2/U0/ethernet_statistics_32bit/_mux0054 [3]),
.R(\BU2/U0/ethernet_statistics_32bit/_or0005 ),
.C(host_clk),
.Q(host_rd_data_143[3])
);
FDR \BU2/U0/ethernet_statistics_32bit/host_rd_data_2 (
.D(\BU2/U0/ethernet_statistics_32bit/_mux0054 [2]),
.R(\BU2/U0/ethernet_statistics_32bit/_or0005 ),
.C(host_clk),
.Q(host_rd_data_143[2])
);
FDR \BU2/U0/ethernet_statistics_32bit/host_rd_data_1 (
.D(\BU2/U0/ethernet_statistics_32bit/_mux0054 [1]),
.R(\BU2/U0/ethernet_statistics_32bit/_or0005 ),
.C(host_clk),
.Q(host_rd_data_143[1])
);
FDR \BU2/U0/ethernet_statistics_32bit/host_rd_data_0 (
.D(\BU2/U0/ethernet_statistics_32bit/_mux0054 [0]),
.R(\BU2/U0/ethernet_statistics_32bit/_or0005 ),
.C(host_clk),
.Q(host_rd_data_143[0])
);
FDR \BU2/U0/ethernet_statistics_32bit/enb (
.D(\BU2/U0/ethernet_statistics_32bit/_or0004 ),
.R(ref_reset),
.C(ref_clk),
.Q(\BU2/U0/ethernet_statistics_32bit/enb_141 )
);
VCC \BU2/U0/XST_VCC (
.P(\BU2/U0/N1 )
);
GND \BU2/U0/XST_GND (
.G(\BU2/U0/N0 )
);

```

```
// synopsys translate_on
```

```
endmodule
```

```
// synopsys translate_off
```

```
`timescale 1 ps / 1 ps
```

```
module glbl ();
```

```
parameter ROC_WIDTH = 100000;
parameter TOC_WIDTH = 0;
```

```
wire GSR;
wire GTS;
wire PRLD;
```

```
reg GSR_int;
reg GTS_int;
reg PRLD_int;
```

```
//----- JTAG Globals -----
```

```
wire JTAG_TDO_GLBL;
```

```

wire JTAG_TCK_GLBL;
wire JTAG_TDI_GLBL;
wire JTAG_TMS_GLBL;
wire JTAG_TRST_GLBL;

reg JTAG_CAPTURE_GLBL;
reg JTAG_RESET_GLBL;
reg JTAG_SHIFT_GLBL;
reg JTAG_UPDATE_GLBL;

reg JTAG_SEL1_GLBL = 0;
reg JTAG_SEL2_GLBL = 0;
reg JTAG_SEL3_GLBL = 0;
reg JTAG_SEL4_GLBL = 0;

reg JTAG_USER_TDO1_GLBL = 1'bz;
reg JTAG_USER_TDO2_GLBL = 1'bz;
reg JTAG_USER_TDO3_GLBL = 1'bz;
reg JTAG_USER_TDO4_GLBL = 1'bz;

assign (weak1, weak0) GSR = GSR_int;
assign (weak1, weak0) GTS = GTS_int;
assign (weak1, weak0) PRLD = PRLD_int;

initial begin
    GSR_int = 1'b1;
    PRLD_int = 1'b1;
    #(ROC_WIDTH)
    GSR_int = 1'b0;
    PRLD_int = 1'b0;
end

initial begin
    GTS_int = 1'b1;
    #(TOC_WIDTH)
    GTS_int = 1'b0;
end

endmodule

// synopsys translate_on

```