FPGA based Implementation for Multi-gigabit high precision network measurements

Gavaletakis Antonios

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, Heraklion, GR-70013, Greece

Thesis Advisors: Prof. *Apostolos Traganitis*
Co-advisor: Dr. *Stefanos Papadakis*

University of Crete
Computer Science Department

FPGA based Implementation for Multi-gigabit high precision network measurements

Thesis submitted by
Gavaletakis Antonios
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author:	Gavaletakis Antonios
Committee approvals:	Traganitis Apostolos Professor, Thesis Supervisor
	Tsakalides Panagiotis Professor, Committee Member
	Mouchtaris Athanasios Assistant Professor, Committee Member
Departmental approval:	Angelos Bilas Professor, Director of Graduate Studies

Heraklion, July 2014

Abstract

In recent years network measurements have been recognized as an area of major interest.There are various software implementations for network measurements, which employ the technique of packet generation. Despite the simplicity and low cost of the software solutions, there are some major drawbacks, like the lack of accuracy, especially at high rates, and the high computational resources requirement leading to the high energy consumption. Therefore in order to have trustworthy results the use of high cost hardware is essential.

An alternative solution is to use specialized hardware that incorporates FPGAs that combine high computational capabilities with low energy consumption. One reasonably priced and academic community driven platform is the NetFPGA PCI card. It provides 4 Gigabit Ethernet ports and a powerful FPGA that may be exploited for the development of any network application.

Using the NetFPGA platform we developed an affordable, high precision and accuracy multi-gigabit network measurements system, that outperforms any software solution, requires minimum computational capabilities by the host computer and it consumes very low energy. Moreover we have shown the lack of accuracy in the measurements of the popular software solutions, even in low rates, establishing the need of a hardware based solution for reliable results.

Περίληψη

Οι μετρήσεις δικτύων έχουν αναγνωριστεί ως ένας τομέας υψηλού ενδιαφέροντος τα τελευταία χρόνια. Υπάρχουν ποικίλες εφαρμογές σε λογισμικό για μετρήσεις δικτύων, οι οποίες εφαρμόζουν την τεχνική της γέννησης πακέτων. Το λογισμικό παρόλο που είναι απλό και χαμηλού κόστους έχει σημαντικά μειονεκτήματα όπως η έλλειψη ακρίβειας, κυρίως σε υψηλούς ρυθμούς μετάδοσης δεδομένων, και οι υψηλές απαιτήσεις σε υπολογιστικούς πόρους που συνεπάγονται την υψηλή κατανάλωση ενέργειας. Για τους λόγους αυτούς η χρήση ακριβού υλικού είναι απαραίτητη ώστε να επιτυγχάνονται αξιόπιστα αποτελέσματα.

Μια εναλλακτική λύση είναι η χρήση εξειδικευμένου υλικού με FPGAs που συνδυάζουν υψηλές υπολογιστικές δυνατότητες με χαμηλή κατανάλωση ενέργειας. Μια χαμηλού κόστους πλατφόρμα, αναπτυγμένη από την ακαδημαϊκή κοινότητα είναι η PCI κάρτα NetFPGA. Παρέχει 4 Gigabit Ethernet πόρτες και φέρει μία ισχυρή FPGA που μπορεί να αξιοποιηθεί για την δημιουργία δικτυακών εφαρμογών.

Χρησιμοποιώντας την πλατφόρμα της NetFPGA, αναπτύξαμε ένα φτηνό και υψηλής ακρίβειας, multi-gigabit σύστημα δικτυακών μετρήσεων, το οποίο υπερτερεί έναντι οποιασδήποτε λύσης σε λογισμικό, απαιτεί ελάχιστες υπολογιστικές δυνατότητες από τον υπολογιστή και καταναλώνει πολύ λίγη ενέργεια. Επίσης δείξαμε την έλλειψη ακρίβειας των μετρήσεων από λογισμικό, ακόμα και σε χαμηλούς ρυθμούς μετάδοσης δεδομένων, αποδεικνύοντας ότι για αδιαμφισβήτητα αποτελέσματα είναι αναγκαία μια λύση βασισμένη σε υλικό.

Acknowledgements

Αρχικά θα ήθελα να ευχαριστήσω τον κύριο Στέφανο Παπαδάκη για την ιδέα της υλοποίησης της εργασίας αυτής και για την συνεχή βοήθεια και υποστήριξη του όλο αυτό το διάστημα. Οι συμβουλές του ήταν καθοριστικές για την διεκπεραίωση της εργασίας.

Στις ευχαριστίες αυτές δεν πρέπει να ξεχάσω τα καταπληκτικά παιδιά του εργαστηρίου του TNL στο FORTH. Η καθημερινή τους βοήθεια ήταν ανιδιοτελής και ανεκτίμητη. Ιδιαίτερα θέλω να ευχαριστήσω τους Μανώνη Σουρλίγκα, Λάμπρο Κατσίκα, Νίκο Βερβελάκη και Αντώνη Μακρογιαννάκη.

Πολλές ευχαριστίες επίσης δίνω στους γονείς μου και στον αδερφό μου για την υποστήριξη τους σε όλα αυτά τα χρόνια των ακαδημαϊκών μου σπουδών.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, a growing number of users are connected to the Internet, wired networks are expanded, new backbone networks are being installed, and the existing backbones are further developed, thus making network measuring imperative. Wired networks exist -at a small scale- in companies, in laboratories or at homes while, at a large scale, between countries and continents, and serve file transfers or telephony over IP. Most of the times, performance is the major criterion with which we set the cost of the service between a client and a company.

As a result, the packet processing like packet classification, manipulation, and forwarding has to be done at very high rates. The only way to guarantee that wired networks work properly is to test them under worst-case conditions i.e., with maximum traffic load. In particular, traffic generators are needed for generating traffic and fully utilizing links in order to evaluate a packet processing performance under these conditions.

There are a lot of implementations of packet generators in software that provide high configurability although they have a lot disadvantages. First of all, the software implementations are not 100% accurate in generating packets at high rates as the software performance depends on the hardware of the computer. What is more, generating high rates demands a lot of computer resources. The more computer resources are used the more energy is demanded, so the power consumption is an important issue. Last but not least, it is the cost for reliable measurement that requires a very powerful processor and generally a very fast computer that makes the cost really high.

With the use of NetFPGA hardware we expect that all these issues will be resolved. The NetFPGA projects are independent of computer's capabilities and are able to process packets at very high rates, so it could guarantee the high quality of network measurements. In addition, we suppose that the power consumption of the card would be always the same, at low level comparing to the computer power consumption. This is because the execution code on the FPGA always consumes the same energy.

This thesis has been inspired by the need for an affordable multi-gigabit high precision network measurements program. This can be achieved by implementing a configurable traffic generator which offers full Gigabit Ethernet link utilization. In addition, as high precision measurements are needed, both the part of the packets generation and the part of the receiving and analysing packets should be implemented in advance; thus, enabling it to work at rates of gigabit at high-performance.

To date, there has been a wide variety of tools generating traffic—both commercial and open-source solutions. One example of open source software solution is iPerf [1]. What we expect to reveal after conducting the research is a striking disadvantage of this tool is its performance at high-speed measurements. As it does not achieve higher link utilization rates because of its dependence on the hardware it leads to inaccurate measurements results.

On the other hand, commercial products like the IxChariot of IXIA [2] have better results than the open-source free-ware programs on testing packet processing with high loads. Unfortunately they are depended on the traffic patterns sent and they have the disadvantage of the high cost.

To conclude, all these disadvantages of inaccurate measurements and high power consumption that the software implementations have, as well as the high cost of the commercial products can be solved with this implementation of Multi-gigabit high precision network measurement project on the NetFPGA 1G platform. The goal of this project is to implement a Packet Generator that provides reliable flow generation without the pre knowledge of other flows. Because of the technology of the FPGA the power consumption is dramatically less than similar implementations in software.

1.1 Background

1.1.1 Measuring techniques

There are several methods of measuring the performance of a network either wired or wireless [3], [4],[5]. The packet generation used involves the injection of probe packets into the network for measuring the statistics results. Specifically, there is a system (for example a network interface of a computer) that is generating a flow of packets of the same header and payload size, on the one end, while on the other we have a system that receives this flow of packets. The metrics are calculated by observing the size, the amount and the arrival time of the packets. The metrics could be throughput, packets per second jitter and packet lost.

In a packet network, the term throughput characterizes the amount of data that the network can transfer per unit of time and is measured in bits per seconds (bps). Jitter or Packet Delay Variation (PDV) is the difference in end-to-end one-way delay between selected packets in a flow with any lost packets being ignored [6]. Due to the devices' performance and the actual state of the network, the statistics can be variable and might not achieve the theoretical limits.

1.1.2 The NetFPGA board

For the implementation of the packet generator the NetFPGA board is used. NetFPGA (network Field-Programmable Gate Array) is the low-cost reconfigurable hardware platform optimized for high-speed networking. It includes all the logic resources, memory, and Gigabit Ethernet interfaces necessary to build a complete switch, router or a security device. Because the entire data path is implemented in hardware, the system can support back-to-back packets at full Gigabit line rates and has a processing latency measured in only a few clock cycles. [7]

For this thesis the first version of NetFPGA is used, NetFPGA-1G, which features a FPGA Xilinx Virtex-II Pro 50 at 125MHz frequency clock with 53,136 logic cells and 4 external RJ45 plugs for gigabit Ethernet connections. The wire-speed processing on all ports at all time using FPGA is 8Gbps throughput

$$[1\,Gbits * 2(bi - directional) * 4(ports) = 8\,Gbps\,throughput]$$

Moreover, it has 4.5 MB of Static Random Access Memory (SRAM), 64 MB of Double-Date Rate Random Access Memory (DDR2 DRAM) at 400MHz with asynchronous clock, suitable for packet buffering. In addition, its PCI (Peripheral Component Interconnect) interface provides a fast reconfiguration of the FPGA, without using JTAG cable, and CPU access to memory-mapped registers and memory on the NetFPGA hardware. Finally, two SATA-style connectors to Multi-Gigabit I/O (MGIO) on right-side of PCB allows multiple NetFPGAs within a PC to be chained together. All these components are showed at figure 1.1. In the following chapters it is explained how these components are used for this project.

1.2 Motivation

Up to date, there have been several implementations of packet generation either on software or hardware platforms[8],[2], all of which have very serious shortcomings.

On the one hand, compared to hardware, software programs are much easier to develop and as a result, there are plenty of software implementations for the user to choose and they can all be executed without applying any changes on the computer. On the other hand, the software implementations depend on the features of the computers (CPU performance, memory, operating system etc.) that they are running on, which necessitates a more powerful computer, which, in turn, affects cost. There are free implementations but there is risk of low performance with unclear statistics. On the other hands, commercial products have better statistical results but the cost of software licences is high. Nowadays, low-power consumption is a major issue not covered by software implementation as the production of high-rate flows demands the full utilization of a CPU and results in high consumption either way.

Hardware implementation always yields high performance. The speed-up of the hardware implementations compared to the software implementations is substan-

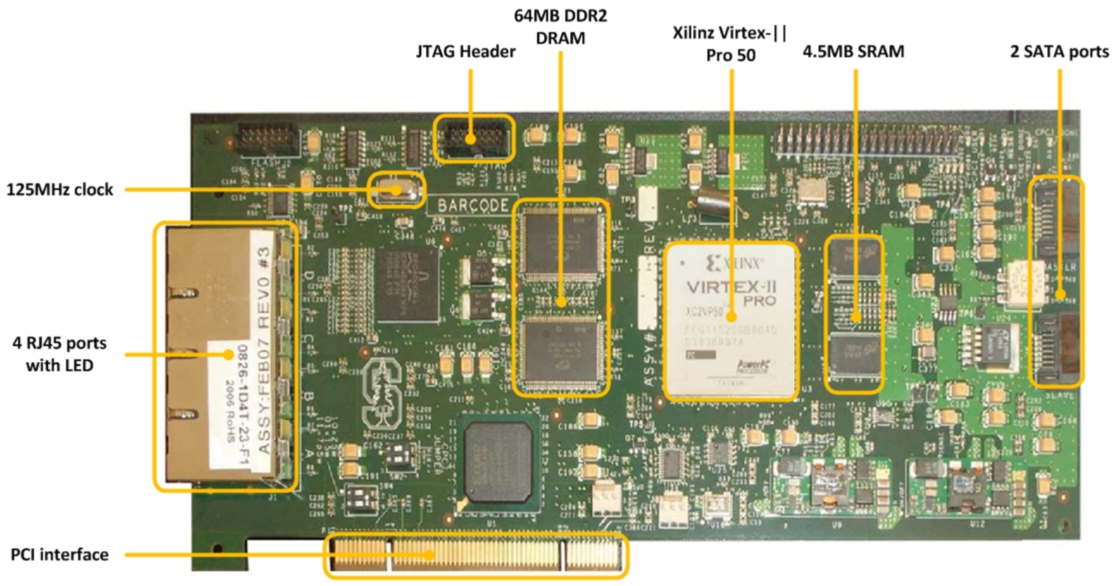


Figure 1.1: The components of the NetFPGA 1G board

tial as the low-level programming better exploits the hardware. For applications that need high performance and precision, for example, producing packet flows and measuring flows at high rates, hardware implementations are the most appropriate. In addition, the power consumption is lower compared to their equivalent software implementations. However, the most serious drawbacks of the hardware implementations are that it needs more time to implement a project in hardware than in software, and that the cost is pretty high depending on how powerful the hardware used is. That is to say, there is a trade-off between high hardware performance and low software cost.

A possible solution could be an implementation in NetFPGA, that combines high performance, low consumption of energy and low cost. The performance of NetFPGA is the same whichever computer is used, so the advantage is that it can be used with low performance computer and as a result with low cost computer, with the same precise results. Also, using the FPGA technology, the power consumption remains the same whether we are measuring the network or not. By implementing a packet generation system in hardware and an easy-to-use GUI(Graphical User Interface) in Java, we combine the performance of the hardware with the high configurability of the software. This implementation is as effective as any commercial product out in the market right now.

1.3 Objective

The aim of this work is to build a packet generator of multiple simultaneous flows by fully utilize the capabilities of NetFPGA-1G. One of the goals is to build a reliable system that will be able to construct gigabit Ethernet flows with high precision and will be able to measure the receiving packets at the same time. It is important the generated flows to have constant throughput and jitter, so the high precision of packet generation is needed. It is of equal importance that the processing of the receiving packets is expedited quickly enough so that no packets are lost and no results get distorted.

Up to date, there have been similar implementations [9] that are based on pre-captured flow from a network. In order to generate a flow, it is needed to have been pre-stored. Therefore, a large database of stored net-flows is mandatory. This is important limitation that does not allow us to easy generate whatever flow we want. Aim of this implementation is not to have this limitation, by generating and the sending of the flow without need of a prefabricated filing base. The goal is, the generated packets to be produced only from the preferences of the user without the pre-capture of any traffic flow.

For the facilitation of the user, we want to built a GUI that provides all the necessary choices for configuration of the generated flows and of course a visualization of the results of the flows. As we want the system of the NetFPGA and the GUI to be more independent and to executed in different computers, we plan this two components to communicate through network.

Considering the GUI, the user will have the possibility to simulate UDP flows and flows of audio and video over IP networks. By changing the size of the payload and consequently the size of the packet, the user could receive measurements of maximum throughput (maximum payload size) and maximum packets per second (minimum payload size).

1.4 Thesis Organization

The rest of the thesis is organized as follows; Chapter 2 presents some background on networks protocols and headers that are used in this project and on NetFPGA reference design. The design and architecture of the hardware implementation are discussed in Chapter 3. The results of the implementation are presented in Chapter 4. Chapter 5 summarizes and concludes this thesis by listing future enhancements that can be implemented.

Chapter 2

Development Platform and Networks Layers

2.1 NetFPGA and Reference Switch project

The goal is to built a traffic generator using the NetFPGA 1G. In order to achieve this goal, the first step is to understand how the reference project of the Reference Switch works and how it is programmed and compiled.

All the functionality of the Reference Switch project is written in the User Data Path (UDP) module. It has sixteen interfaces in total, four input interfaces for the Peripheral Component Interconnect (PCI) and four input interfaces for the network, four output interfaces for the PCI and another four output interfaces for the network. In addition, it has interfaces for Static Random-Access Memory SRAM and Dynamic Random-Access Memory (DRAM) memories. The User Data Path is 64 bits wide running at 125MHz which is enough to serve a 8 Gbps flow

$$4ports * 2bi - directional * 1Gbps = 8Gbps\ .$$

$$\frac{User\,Data\,Width}{Clock\,Period} = 8Gbps\ .$$

The modules of the UDP work according to the reference pipelining as described below. Packets pass between modules using First Input First Output (FIFO), just like a simple push interface with four signals: WR, RDY, DATA, and CTRL. When a module wants to send a packet to the next module it checks if the one bit wire RDY is on. When it is on, the one bit wire WR indicates that the sent data is valid. The 8 bit wire CTRL indicates the type of the data. The accepted values of CTRL are three; 0xFF, which indicates the module header at DATA wire; 0x00, which indicates the data of packet at the DATA wire; and 0x(number), which indicates the end of the packet's DATA and the number of the valid byte of the data using one bit right-shifted as many places as the valid data are as the table 2.1 shows.

Ctrl = 0x80, 8 valid bytes
Ctrl = 0x40, 7 valid bytes
Ctrl = 0x20, 6 valid bytes
Ctrl = 0x10, 5 valid bytes
Ctrl = 0x08, 4 valid bytes
Ctrl = 0x04, 3 valid bytes
Ctrl = 0x04, 3 valid bytes
Ctrl = 0x02, 2 valid bytes
Ctrl = 0x01, 1 valid bytes
Ctrl = 0x00, 0 valid bytes

Table 2.1: Ctrl's example values

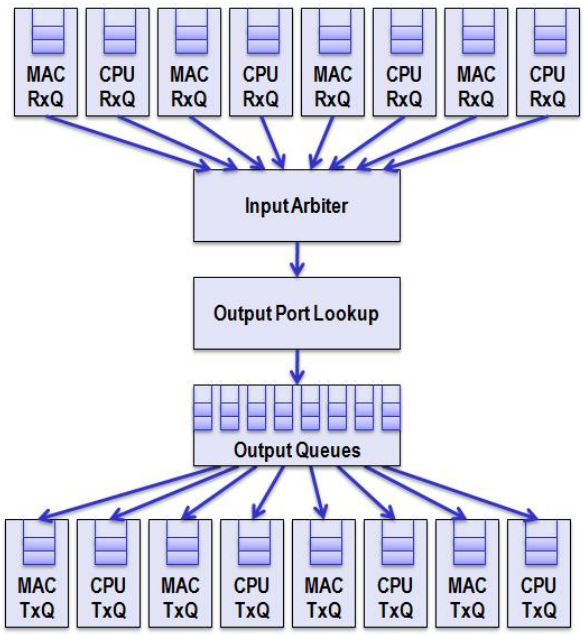
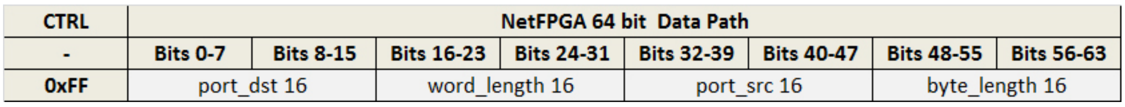


Figure 2.1: Reference pipeline of Reference Switch project

The UDP of the Reference Switch consists of 3 modules, the "Input Arbiter", "Output Port Lookup" and "Output Queues" as the figure 2.1 shows. When packets come from the network, they are stored in the input MAC RxQ queues. When packets come from the PCI interface, they are stored in the input CPU RxQ queues. Afterwards the packets enter the UDP. The packets that enter the UDP have a module header in the beginning as the figure 2.2 shows.This header stores the length of the packet in bytes in the 48–63 bits bits of the module header, the source port as a binary one-hot-encoded number in 32–47 bits for example port 0 is MAC port 0 and port 1 is CPU port 0, and the 64-bit word packet length in 16–31 bits. When the packet enters the UDP, Input Arbiter is responsible for pushing it to the Output Port Lookup by selecting sequentially an Rx queue to service. The Output Port Lookup module decides which output port(s) a packet goes out of and writes the output ports selection as an one-hot-encoded number into 0–15 bits of the module header.When the Output Queue module receives the packet, it looks at the module header to decide in which output queue to store the packet in order to push it into its destination Tx Queue. [10]



CTRL	NetFPGA 64 bit Data Path							
-	Bits 0-7	Bits 8-15	Bits 16-23	Bits 24-31	Bits 32-39	Bits 40-47	Bits 48-55	Bits 56-63
0xFF	port_dst 16		word_length 16		port_src 16		byte_length 16	

Figure 2.2: Module header fields:

2.2 Headers in detail

2.2.1 ARP Link Layer

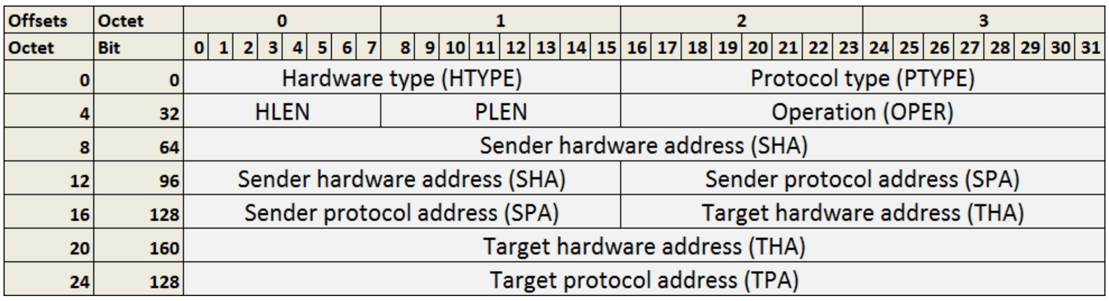
Address Resolution Protocol (ARP) is a request and reply telecommunication protocol that converts Protocol Addresses (e.g., IP addresses) to Local Network Addresses (e.g., Ethernet addresses). It is communicated within the boundaries of a single network and never routed across internetwork nodes. ARP header is placed into the Link Layer of the Internet Protocol Suite.

The ARP protocol is used when a network device does not have the mapping of the IP to media access control address (MAC address). ARP broadcasts a request packet in a special format to all the machines on the Local Area Network (LAN) to see if one machine knows that it has that IP address associated with it. A machine that recognizes the IP address as its own returns a reply so indicating.[11]

ARP may also be used as a simple announcement protocol. This is useful for updating other hosts' mapping of a hardware address when the sender's IP address or MAC address has changed. Such an announcement, also called a gratuitous ARP message. A gratuitous ARP request is an ARP request packet where the source and destination IP are both set to the IP of the machine issuing the packet and the destination MAC is the broadcast address. Ordinarily, no reply packet will occur. An alternative is to broadcast an ARP reply with the sender's hardware and protocol addresses (SHA and SPA) duplicated in the target fields (TPA=SPA, THA=SHA) [12].

The principal packet structure of ARP packets is shown in the 2.3 figure which illustrates the case of IPv4 networks running on Ethernet. In this scenario, the packet has 48-bit fields for the sender hardware address (SHA) and target hardware address (THA), and 32-bit fields for the corresponding sender and target protocol addresses (SPA and TPA). Thus, the ARP packet size in this case is 28 bytes.

- **Hardware type (HTYPE)**: 16 bits. This field specifies the network protocol type. Ethernet is 1.
- **Protocol type (PTYPE)** 16 bits.This field specifies the internetwork protocol for which the ARP request is intended. For IPv4, this has the value 0x0800.
- **Hardware length (HLEN)**: 8 bits. Length (in octets) of a hardware address. Ethernet addresses size is 6.
- **Protocol length (PLEN)**: 8 bits. Length (in octets) of addresses used in the upper layer protocol. IPv4 address size is 4.
- **Operation (OPER)**: 16 bits. Specifies the operation that the sender is performing: 1 for request, 2 for reply.
- **Sender hardware address (SHA)**: 48 bits. Media address of the sender.
- **Sender protocol address (SPA)**: 32 bits. Internetwork address of the sender.
- **Target hardware address (THA)**: 48 bits. Media address of the intended receiver. This field is ignored in requests.
- **Target protocol address (TPA)**: 32 bits. Internetwork address of the intended receiver.



Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Hardware type (HTYPE)																Protocol type (PTYPE)															
4	32	HLEN								PLEN								Operation (OPER)															
8	64	Sender hardware address (SHA)																															
12	96	Sender hardware address (SHA)																Sender protocol address (SPA)															
16	128	Sender protocol address (SPA)																Target hardware address (THA)															
20	160	Target hardware address (THA)																															
24	128	Target protocol address (TPA)																															

Figure 2.3: ARP header's fields

2.2.2 IEEE 802.3 Ethernet frame

Ethernet is the most common local area networking technology, and, with gigabit and 10 gigabit Ethernet, is also being used for metropolitan-area and wide-area networking. Ethernet refers to the family of LAN products covered by the IEEE 802.3 standard that defines the carrier sense multiple access collision detect (CSMA/CD) protocol. 802.3 specifies the physical media and the working characteristics of Ethernet. Four data rates are currently defined for operation over optical fiber and twisted-pair cables: 10Base-T Ethernet (10 Mb/s), Fast Ethernet (100 Mb/s), Gigabit Ethernet (1000 Mb/s) and 10-Gigabit Ethernet (10 Gb/s).

The IEEE 802.3 standard provides MAC (Layer 2) addressing, duplexing, differential services, and flow control attributes, and various physical (Layer 1) definitions, with media, clocking, and speed attributes.

The figure 2.4 shows the complete Ethernet frame, as transmitted, for the payload size up to the Maximum transmission unit (MTU) of 1500 octets. Some implementations of Gigabit Ethernet support larger frames, known as jumbo frames.

- **Preamble:** 64bits. It consists of seven bytes all of the form 10101010. It is used by the receiver to allow it to establish bit synchronization

- **Destination MAC address:** 48 bits. This field specifies the receiver MAC address of the packet. A destination MAC address of ff:ff:ff:ff:ff:ff indicates a Broadcast, meaning the packet is sent from one host to any other on that network.

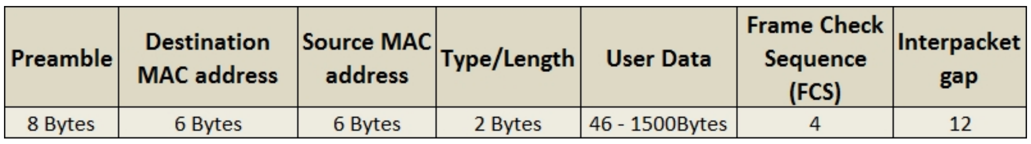
- **Source MAC address:** 48 bits. This field specifies the sender MAC address of the packet.

- **Type / Length field:** 16bits. It can be used for two different purposes. If the type/length field has a value 1500 or lower, it's a length field, otherwise it's a type field and is followed by the data for the upper layer protocol. When the length/type field is used as a length field the length value specified does not include the length of any padding bytes.

- **User Data:** 46 octets - 1500 octets. Non-standard jumbo frames allow for larger maximum payload size.

- **Frame check sequence (FCS):** 32bits. It is a 4-octet cyclic redundancy check which allows detection of corrupted data within the entire frame.

- **Interpacket gap:** 96bits. Idle time between packets. After a packet has been sent, transmitters are required to transmit a minimum of 96 bits (12 octets) of idle line state before transmitting the next packet.



Preamble	Destination MAC address	Source MAC address	Type/Length	User Data	Frame Check Sequence (FCS)	Interpacket gap
8 Bytes	6 Bytes	6 Bytes	2 Bytes	46 - 1500Bytes	4	12

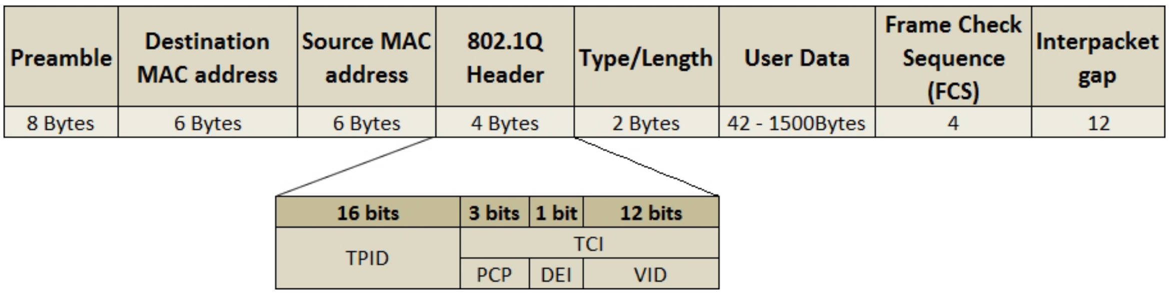
Figure 2.4: 802.3 Ethernet packet

2.2.3 IEEE 802.1Q Ethernet frame

IEEE 802.1Q, or VLAN Tagging, is an IEEE standard allowing multiple bridged networks to transparently share the same physical network link without leakage. IEEE 802.1Q is used to refer to the encapsulation protocol used to implement this mechanism over Ethernet networks.

802.1Q adds a 32-bit field between the source MAC address and the EtherType/length fields of the original Ethernet frame, leaving the minimum frame size unchanged at 64 bytes (octets) and extending the maximum frame size from 1,518 bytes to 1,522 bytes. The minimum payload size is 42-octets. Two bytes are used for the tag protocol identifier (TPID), the other two bytes for tag control information (TCI). The TCI field is further divided into PCP, DEI, and VID. The figure 2.5 shows the complete Ethernet frame with VLAN tagging enabled.

- **Tag protocol identifier (TPID):** 16 bits. It set to a value of 0x8100 in order to identify the frame as an IEEE 802.1Q-tagged frame. This field is located at the same position as the EtherType/length field in untagged frames, and is thus used to distinguish the frame from untagged frames.
- **Tag protocol identifier (TPID):** 16bits.
  - **Priority code point (PCP):** 3 bits. This field refers to the IEEE 802.1p priority. It indicates the frame priority level. Values are from 0 (best effort) to 7 (highest); 1 represents the lowest priority. These values can be used to prioritize different classes of traffic (voice, video, data, etc.).
  - **Drop eligible indicator (DEI):** 1 bit. May be used separately or in conjunction with PCP to indicate frames eligible to be dropped in the presence of congestion.
  - **VLAN identifier (VID):** 12 bit. It specifies the VLAN to which the frame belongs. The hexadecimal values of 0x000 and 0xFFF are reserved.



Preamble	Destination MAC address	Source MAC address	802.1Q Header	Type/Length	User Data	Frame Check Sequence (FCS)	Interpacket gap
8 Bytes	6 Bytes	6 Bytes	4 Bytes	2 Bytes	42 - 1500Bytes	4	12

16 bits	3 bits	1 bit	12 bits
TPID	TCI		
	PCP	DEI	VID

Figure 2.5: 802.1Q Ethernet packet

2.2.4 IPv4 Network Layer

IPv4 is a protocol that is used for routing the data packets between different auto-organized networks. It operates on a best effort delivery model, in that it does not guarantee delivery, nor does it assure proper sequencing or avoidance of duplicate delivery. The internet protocol uses five key fields/values in the header in providing its service:IP address, Type of Service, Time to Live, and Header Checksum.

Each device that participate in a computer network should have an IP address. With this way the hosts and the network interfaces are identified. An address indicates where the device is.

The Type of Service mechanism is used to indicate the quality of the service desired by changing the priority of the packet queuing and routing on the routers.

Time to Live is an indication of an upper bound on the lifetime of an internet datagram. It is set by the sender of the datagram and reduced at the points along the route where it is processed. If the time to live reaches zero before the internet datagram reaches its destination, the internet datagram is destroyed. The time to live mechanism can be thought of as a self destruct time limit.

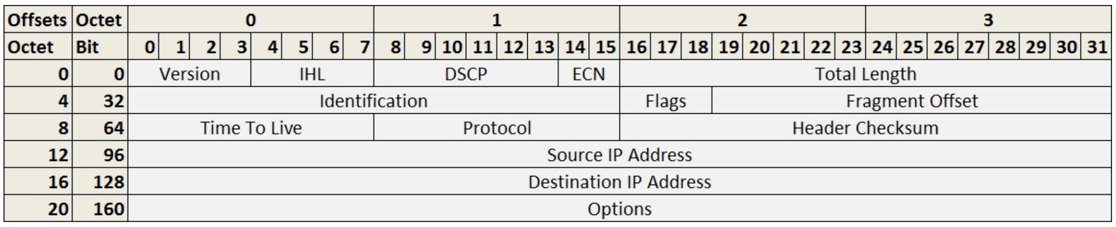
The Header Checksum provides a verification that the information used in processing internet datagram has been transmitted correctly. The data may contain errors. If the header checksum fails, the internet datagram is discarded at once by the entity which detects the error.

The internet protocol does not provide a reliable communication functionality. There are no acknowledgements either end-to-end or hop-by-hop. There is no error control for data, no retransmissions and no flow control. [13]

The header of the IPv4 is consist of 20 bytes and 13 fields as it is shown at the figure 2.6.[14]

- **Version:** 4 bits. Indicates the version of the IP. Always assigned to 4.
- **Internet Header Length (IHL):** 4 bits. Size of IP header in 32-bit words. The minimum value is 5, which is a length of $5 \times 32 = 160$ bits $= 20$ bytes. The maximum value is 15 words, $15 \times 32 = 480$ bits $= 60$ bytes.
- **Differentiated Services Code Point (DSCP):** 6 bits. Originally defined as the Type of service field.
- **Explicit Congestion Notification (ECN):** 2 bits. ECN is an optional feature that allows end-to-end notification of network congestion without dropping packets.
- **Total Length:** 16 bits.It defines the entire packet (fragment) size, including header and data, in bytes. The minimum-length packet is 20 bytes (20-byte header + 0 bytes data) and the maximum is 65,535 bytes. The largest datagram that any host is required to be able to reassemble is 576 bytes, but most modern hosts handle much larger packets.

- **Identification:** 16 bits. An identifying value assigned by the sender to aid in assembling the fragments of a datagram.
- **Various Control Flags:** 3 bits. Bit 0: reserved, must be zero Bit 1: 0 = May Fragment, 1 = Don't Fragment. Bit 2: 0 = Last Fragment, 1 = More Fragments.
- **Fragment Offset:** 13 bits. This field indicates where in the datagram this fragment belongs. The fragment offset is measured in units of 8 octets (64 bits). The first fragment has offset zero.
- **Time to Live:** 8 bits. This field indicates the maximum time the datagram is allowed to remain in the internet system. If this field contains the value zero, then the datagram must be destroyed. This field is modified in internet header processing.The intention is to cause undeliverable datagrams to be discarded, and to bound the maximum datagram lifetime.
- **Protocol:** 8 bits. This field indicates the next level protocol used in the data portion of the internet datagram.
- **Header Checksum:** 16 bits. A checksum on the header only. Since some header fields change (e.g., time to live), this is recomputed and verified at each point that the internet header is processed.
- **Source Address:** 32 bits. This field is the IPv4 address of the sender of the packet.
- **Destination Address:** 32 bits. This field is the IPv4 address of the receiver of the packet.



Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags			Fragment Offset												
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options																															

Figure 2.6: IP header's fields

2.2.5 UDP Transport Layer

User Datagram Protocol(UDP) together with Transmission Control Protocol(TCP) are the two types of Internet Protocol. UDP is not a connection oriented protocol as TCP. Data can be sent bidirectionally with no more effort. Multiple messages are sent as packets in chunks using UDP. UDP protocol is used in DNS, TFTP, SNMP, RIP, VOIP packets and it is very simple because it does not have an inherent order as all of its packets are independent from each other. Moreover, it does not use acknowledgements to check the missing packets; it does not have flow control and, lastly, it does not need handshake for establishing the connection. If ordering or reliability is required, it has to be managed by the application layer. [15]

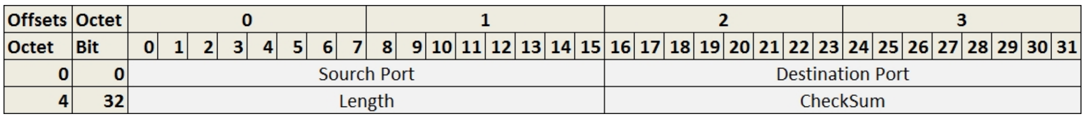
The header of UDP is 8 byte length and consists of 4 fields of 16 bits, Source port, Destination port, length and checksum as it shows the figure 2.7.

- **Source Port:** 16bits. It indicates the port of the sending process, and may be assumed to be the port to which a reply should be addressed in the absence of any other information. If not used, a value of zero is inserted.

- **Destination Port:** 16bits. This field identifies the receiver's port and is required. Similar to source port number, if the client is the destination host then the port number will likely be an ephemeral port number and if the destination host is the server then the port number will likely be a well-known port number.

- **Length:** 16bits. Length is the length in octets of this user datagram including this header and the data.The minimum value of the length is eight.

- **Checksum:** 16bits. The checksum field is used for error-checking of the header and data. It is computed from the IP header, UDP header and the data. If the checksum is cleared to zero, then check summing is disabled. If the computed checksum is zero, then this field must be set to 0xFFFF.



Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Sourch Port																Destination Port															
4	32	Length																CheckSum															

Figure 2.7: UDP header's fields

2.2.6 RTP Application layer

RTP is designed to support end-to-end real-time, transfer of stream data. It is used in video and audio application transferring over multicast or unicast network services that require timely delivery of information and can tolerate some packet loss to achieve this goal. RTP is independent of the underlying transport and network layers. As a result, TCP communication supports the RTP protocol, although the majority of the RTP implementations are built on the UDP. The protocol provides facilities for jitter compensation and detection of packet loss and out of sequence arrival in data, which are common during transmissions on an IP network. RTP allows data transfer to multiple destinations through IP multicast. RTP is regarded as the primary standard for audio/video transport in IP networks and is used with an associated profile and payload format [16]. The RTP header has a minimum size of 12 bytes. After the header, optional header extensions may be present. This is followed by the RTP payload, the format of which is determined by the particular class of application. The fields in the header are as as it shows the figure 2.8[17]:

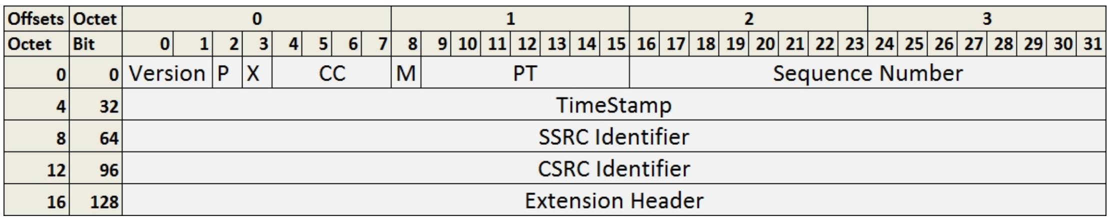
- **Version:** 2 bits. RTP version number. Always set to 2.
- **P, Padding:** 1 bit. If set, this packet contains one or more additional padding bytes at the end which are not part of the payload. The last byte of the padding contains a count of how many padding bytes should be ignored. Padding may be needed by some encryption algorithms with fixed block sizes or for carrying several RTP packets in a lower-layer protocol data unit.
- **X, Extension:** 1 bit. If set, the fixed header is followed by exactly one header extension.
- **CC, CSRC count:** 4 bits. The number of CSRC identifiers that follow the fixed header.
- **M, Marker:** 1 bit. The interpretation of the marker is defined by a profile. It is intended to allow significant events such as frame boundaries to be marked in the packet stream. A profile may define additional marker bits or specify that there is no marker bit by changing the number of bits in the payload type field.
- **PT, Payload Type:** 7 bits. Identifies the format of the RTP payload and determines its interpretation by the application. A profile specifies a default static mapping of payload type codes to payload formats. Additional payload type codes may be defined dynamically through non-RTP means. An RTP sender emits a single RTP payload type at any given time; this field is not intended for multiplexing separate media streams.
- **Sequence Number:** 16 bits. The sequence number increments by one for each RTP data packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence. The initial value of the sequence number is random (unpredictable) to make known-plaintext attacks on encryption more difficult, even if the source itself does not encrypt, because the packets may flow through a translator that does.
- **Timestamp:** 32 bits. The timestamp reflects the sampling instant of the first octet in the RTP data packet. The sampling instant must be derived from a clock that increments monotonically and linearly in time to allow synchronization and jitter calculations. The resolution of the clock must be sufficient for the desired synchronization accuracy and for measuring packet arrival jitter (one tick per video frame is typically not sufficient). The clock

frequency is dependent on the format of data carried as payload and is specified statically in the profile or payload format specification that defines the format, or may be specified dynamically for payload formats defined through non-RTP means. If RTP packets are generated periodically, the nominal sampling instant as determined from the sampling clock is to be used, not a reading of the system clock. As an example, for fixed-rate audio the timestamp clock would likely increment by one for each sampling period. If an audio application reads blocks covering 160 sampling periods from the input device, the timestamp would be increased by 160 for each such block, regardless of whether the block is transmitted in a packet or dropped as silent.

- **SSRC, Synchronization source:** 32 bits. Identifies the synchronization source. The value is chosen randomly, with the intent that no two synchronization sources within the same RTP session will have the same SSRC. Although the probability of multiple sources choosing the same identifier is low, all RTP implementations must be prepared to detect and resolve collisions. If a source changes its source transport address, it must also choose a new SSRC to avoid being interpreted as a looped source.

- **CSRC, Contributing source:** 32 bits. An array of 0 to 15 CSRC elements identifying the contributing sources for the payload contained in this packet. The number of identifiers is given by the CC field. If there are more than 15 contributing sources, only 15 may be identified. CSRC identifiers are inserted by mixers, using the SSRC identifiers of contributing sources. For example, for audio packets the SSRC identifiers of all sources that were mixed together to create a packet are listed, allowing correct talker indication at the receiver.

- **Extension header: (optional)** The first 32-bit word contains a profile-specific identifier (16 bits) and a length specifier (16 bits) that indicates the length of the extension (EHL=extension header length) in 32-bit units, excluding the 32 bits of the extension header.



Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version		P	X	CC				M	PT							Sequence Number															
4	32	TimeStamp																															
8	64	SSRC Identifier																															
12	96	CSRC Identifier																															
16	128	Extension Header																															

Figure 2.8: RTP header's fields

Chapter 3

Implementation

The system's implementation is described in this chapter. The first step is to define the modules and their functionality. Next, comes the definition of the names and the size of the register that the end user can use in order to communicate with the card and to modify the generated flow. Last but not least is the description of the GUI, which is the most important element for the end user.

3.1 Packet Generator User Data Path

The User Data Path of the Reference Switch is used as reference project in order to implement our system. Our design, as shown in figure 3.1, is divided into two subsystems, the subsystems of the receiver and the subsystems of the transmitter.

The part of the receiver consists of the Input Arbiter module, the Statistics modules and the Packet Parser. All input packets are temporarily stored into the appropriate MAC Input Queue depending on the physical Input MAC Port that the packets entered (packet MAC1 stored into MAC Input Queue 1, etc). Each of the four MAC Input Queues is connected with one instance of Statistics module for having its own statistics results. All the flows that are captured from the Input Queues are aggregated into one flow through the Input_Arbiter module and this aggregated flow is fed to a Statistic module for generating aggregated statistics. The Statistics module announces statistics of the average Throughput per second, Bit per second and Jitter, to the user through the register system. An essential functionality of the system is to reply to ARP requests. When a packet is received, it is analyzed by the Packet Parser module. If it is identified as an ARP request, the module of Packet Parser sends a signal of 1 bit (*ARP_request*) directly to Packet Generator module for producing an ARP reply packet. The packet that is produced is an ARP Gratuitous reply.

The part of transmitter consists of the modules Rater, Packet_Generator, Send and My_Output_Queues. The packet rate is set by the value of a register in Rater module. This defines the period of the *signal_generation* signal that later in connected with the Packet_Generator module which indicates the preparation and sent of the packet. The type and the fields of the headers are filled with the registers' values of the Register System at the Packet_Generator module. The prepared headers with the packet payload are forwarded to the My_Output_Queue module in chunks of 64 bits by the the Send module. My_Output_Queue module stores temporarily the packets chunks of 64 bits into the appropriate MAC_Output_Queues according to the physical port that should be sent. The destination information of each packet is stored in the module header at the respective preamble

In order to extend the capabilities of the system a mechanism is implemented for generating four independent and fully parametrized flows. This is achived with the use of time slots. In each instance of time, a packet of only one flow is generated. The indication of the flow is done with a round robin register which is called *current_flow*. It changes whenever a flow finishes the transmission of a packet or whenever it is on standby as there is no running request for generating new packets. The duration of the preparation and transmission of each flow varies depending on the packet's size. The duration is proportional to the packet size. The minimum transmission time of the smaller packet (60 bytes) is 8 cycles as in each cycle 8 bytes are sent,

$$\frac{(4+60)bytes}{8bytes/cycle} = 8cycles$$

$$8 * 8ns = 64ns$$

The maximum transmission time of the biggest packet (1514 bytes) is 190 cycles,

$$\frac{(4+1514)bytes}{8bytes/cycle} = 190cycles$$

$$190 * 8ns = 1520ns$$

The information of the source, destination physical port and the size of the packet, is stored in the 32 bits module header of the NetFPGA, that adds the overhead of the 4 extra bytes in the calculation of the useful data transfer.

Initially, the aim was to implement four instances from the Rater and Packet_Generator modules. This approach was aborted, however, because of the tremendous size of the design. Eventually, however, the time slotted design yields the same quality of generation of packets and the same statistics result with a smaller design, as it doesn't instantiate four times the modules of the Rater and Packet_Generator but extends the functionality of the existing modules.

Ultimately, the My_Output_Queue module is responsible for storing the packets to the Tx Queues depending on its module header. Whenever a packet enters into the My_Output_Queue module, its module header is identified and then allocated to a Tx Queue.

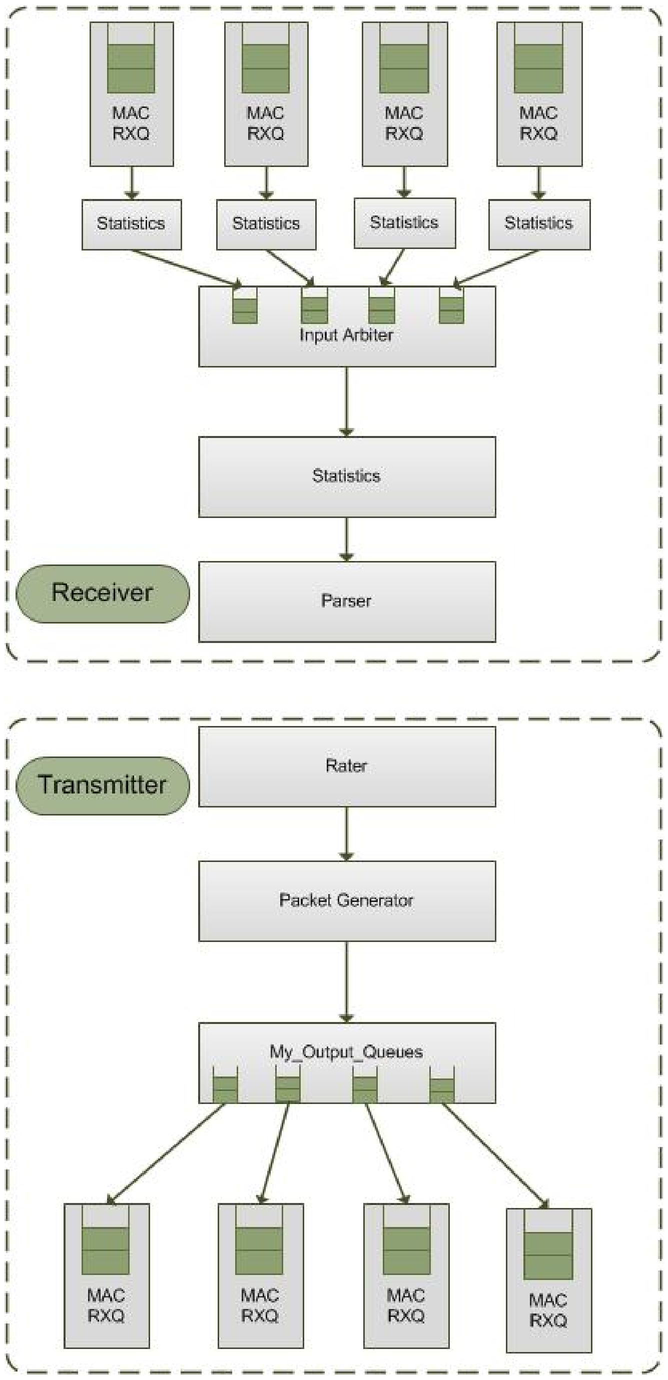


Figure 3.1: Project's User Data Path pipeline

3.2 Modules in Details

3.2.1 Statistics Module

The goal of the Statistics module is to calculate and send to the host computer, through the register system, all the statistics information i.e. Packets per second (pps), Bits per second (bps), and Jitter(ns/sec) that it calculates for each receiving flow. The statistics results are derived from the incoming packets size, the arrival time and the arrival frequency.

State Machine

The module has one state machine that keeps the states of the incoming packet. As shown in the figure 3.2, the states are *WAIT*, *CTRL_FF*, *CTRL_00*, *CTRL_XX*. The *WAIT* state is the initial state of the FSM. The *CTRL_FF* indicates the arrival of the module header and lasts one cycle. The *CTRL_00* is the state where the useful data are received and finally, *CTRL_XX* is the state that indicates the end of the packet. The state machine is initialized at *WAIT* and is on standby for new packets. The advent of a new packet is indicated by the signal of *in_wr* which indicates input valid data, so the state machine changes its state at *CTRL_FF*. At this state the arrival time and the size of the packet are stored by reading the module header. When the value of CTRL is 0x00 the state is changed to *CTRL_00* which indicates the packet useful data. When the last chunk of the packet enters the module the state changes to *CTRL_XX*. This is indicated by the CTRL value, being different from 0x00. The CTRL value indicates the valid bytes of the packet. At this state the number of the received packets is increased by one. When it finishes receiving the packet, the state machine reverts to *WAIT*.

For the metrics of Packets per second (pps), Bits per second (bps), and Jitter, the timing of each second is crucial. To calculate the time a 32 bit register called *counter* is used which increases by +one value for each positive edge of the clock. When this register is 125000000, the counter resets to zero and it marks the end of a second (125MHz Clock frequency, 8ns clock period).

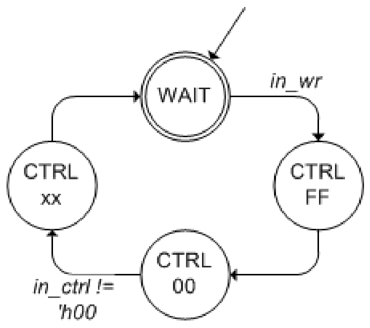


Figure 3.2: Statistic module's FSM

Module Functionality

When a packet is received (state == *CTRL_FF*), the value of the signal data in this state contains the information of source and destination port, and the size of the packet in bytes. The packet size is added into a 32-bit register

$$total\_bits = total\_bits + pack\_size * 8$$

in order to accumulate the total amount of bits that passed into the module. When the counter has counted a period of a second (value 125000000) it is reset to zero, and the value of total_bits equals the bps (bits per second). Next, the value of the total_bits is copied to the register system for access by the user interface.

The calculation of the pps(packet per second) takes place at the ***ctrl_xx*** state when the whole packet has been received. The 32-bit packets register that stores the amount of packets that are received by the module, is increased by one. Like in the bps calculation, when the counter indicates the end of the second, the packets' value equals the pps and it is written into the register system of the NetFPGA. After that the packet register is reset to zero.

Last, the calculation of the jitter demands the knowledge of the arrival time of the last two packets. So, it uses three registers, a 32-bit register (*packet1*) that stores the arrival time of the first packet, a 32-bit register (*packet2*) that stores the arrival time of the second packet and a third 32 bit register (*absolute_ value*) that stores the absolute value of the arrivals' difference of the packets. In the period of one second all the absolute differences are summed into a 32-bit register (*sum_jitter*) and then their average is calculated. For the calculation of the average a separate Division module calculates in 41 cycles the division of two 41-bit numbers. The quotient is written to the register system for user access. Arrival time is related to the counter; therefore, it considers wraparound issues. Finally, a minimum of three packets is needed so as to calculate the jitter, as it is necessary for the packet per second to be greater than three, otherwise the jitter is equal with zero.

3.2.2 Division Module

This module performs a serial division operation, producing one bit of the answer per clock cycle. The dividend's and the divisor's values are unsigned quantities. The division module calculates the division between two 41-bit numbers. The input of the module is two wires of the 41-bits *in_ dividend* and *in_ divider* that represent the numbers that will be divided. The divide operation begins by providing a pulse on the *in_ start* input. The quotient is then provided with 41-bit divider clock cycles as this is the size of the input wires of dividend and divisor. The *in_ start* pulse stores the input parameters in registers, so they do not need to be maintained at the inputs throughout the operation of the module. If an *in_ start* pulse is given to the Division module during the time when it is already working on a previous divide operation, it will abort the operation it was performing and begin working on the new one. Attempting to divide by zero will simply produce a result of all ones. This core is so simple that no checking for this condition is provided. A possible divide by zero condition should be concerned by the user, by comparing the divisor to zero and flagging it. The *ready* pulse indicates the ending of the division process and the correct outputs are written at the 41-bit output wires quotient and remainder.

3.2.3 Packet Parser Module

The packet's header parsing is performed at the Packet Parser module. This module receives the chunks of the packets and identifies the type and the packet headers.

The generated measurements packets can be UDP and ARP packets, so as a result, the expected received packets are one of the these two packet versions. If the following header after the MAC header is not type of IPv4 datagram (EtherType=0x0800) of ARP (EtherType=0x8100) and if the following header after the IP header is not a UDP header (Protocol=17), the packet is dropped and is not further analyzed.

The kind of the packet is important as whenever there is an ARP request, an ARP reply packet should be sent in order to inform the network devices that an IP is assigned with a MAC address. An ARP reply from the NetFPGA interface is important in order to the network devices (routers) forward the NetFPGA generated packets to the destination. Whenever a packet is fully received and marked as an ARP request packet, a pulse is sent to packet generator through the one-bit wire arp_request. The module of the Packet Generator receives this signal and produces a Gratuitous ARP reply for informing the network devices with the MAC address of the NetFPGA interface, so that the network devices knows that they should transmit packets sent to that MAC address on that port[12]. The FSM of the model is working as follows:

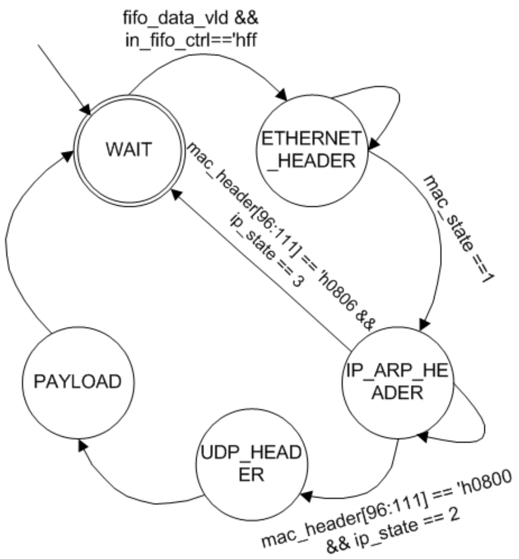


Figure 3.3: Packet Parser module FSM

State Machine

The incoming packets travels from module to module in 64-bit chunks. As a results the Packet Parser Module expects to receive a specific sequence of data value each time. Depending on the state and the received data values the registers of the headers are reconstructed and the packet type is defined. The state machine consists of four states as the figure 3.3 illustrates: *WAIT*, *ETHERNET_HEADER*, *IP_ARP_HEADER*, *UDP_HEADER*, *PAYLOAD*. The *WAIT* state is the initial state of the FSM. Because of buffering all the packets are temporarily stored in a FIFO. When the FIFO buffer is not empty, which means that data are received from the ethernet and the CTRL value of the data is 0xFF, the state changes to *ETHERNET_HEADER*. At this state it is expected to receive the Ethernet header data with the first 16 bits of the next header. This state lasts two period clocks. In the first period the bits from 0 to 63 of the mac header are received and in the second period the remaining bits of mac header (bits 64-111) are received. In addition, the first 16 bits of the next header are received which depends on the value of the mac_type. If the value of mac_type (mac_header[96:111]) is 0x0800, the next header is an IP header; if it is 0x8100, the packet is an ARP packet. The process of the packet continues at the state of IP_ARP_HEADER. At the state of *IP_ARP_HEADER*, depending on the mac_type value, the rest of the part is either filled with the IP header or the ARP. If the incoming packet has an IP header after 3 clock periods, which is needed to receive all the IP header, the state changes to the *UDP_HEADER* state. If it is an ARP packet, upon receiving the ARP header (4 clock periods), the process of the packet ends and the next state is the *WAIT*. At the *UDP_HEADER* state, the rest of the part of the udp_header (bits 48-63) is received and the next state is the *PAYLOAD*. At this state the 64 bits of the chunk is stored as payload and at the next period the state changes back to *WAIT*.

3.2.4 Rater Module

The Rater Module is responsible for the frequency of producing packets.The rate of the packet productions is chosen through the register system. Two 32 bit registers, counter and limit, are responsible for the timing of the module. With every positive edge of the clock, counter is increased by +one until it reaches the value that the user has set at limit register. As the figure 3.4, when the counter reaches the limit value, the counter is reset and the *signal_ out* signal is turned to high for one period of the clock and indicates the generation of a new packet.

This module does not use the ready signal for checking if the next module is ready for receiving data or not. The reason is that the packet generation period time should always be constant and independence of the other modules' state. Whenever it is time to send a packet, the Rater indicates this by raising the *signal_ out* signal. In the Packet Generator Module implementation section 3.2.5, there is a description of the mechanism of receiving this signal, and how notifications are not lost even when the module is busy.

As mentioned earlier, for generating four independent and fully parameterized

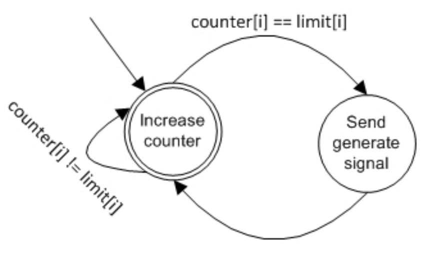


Figure 3.4: Rater module FSM

flows, four different functionalities of Rater module are needed. This is achieved by four duplicates of all the registers and the wires of this module, resulting in four different Rater's signals that will produce four different flows of packets (*signal_ out[0], signal_ out[1], signal_ out[2], signal_ out[3]*). Specifically, there are four Rater modules that work in parallel. The reason that they are implemented, instead of duplicating the whole module, is because of the size of the final project. To duplicate only the registers and the functionality of a module occupies less space than to duplicate the whole module.

3.2.5 Packet Generator Module

The Packet Generator module purpose is to construct the UDP packets that will be sent to the network. The packet's format is defined through the register system as it is described in the paragraph 3.3) by giving the parameters for each flow. The fields of the header are filled according to user selections and IEEE standards as are presented in the tables 3.2, 3.3, 3.4, 3.5, 3.6.

State Machine

The Packet Generator Module consists of two FSMs. The first FSM is responsible for the packet generation and the second is responsible for dividing and sending the packet into 64-bits chunks to the next module. The main state machine consists of four states: ***WAIT***, ***GENERATE_PACKATE***, ***ALL_TO_ONE***, ***SEND***. The states of the second FSM, which are implemented in a different module (Send module), are ***WAIT***, ***FPGA_HEADER***, ***HEADER***, ***HEADER_PAYLOAD***, ***PAYLOAD***, ***END***.

Both state machines are initialized into ***WAIT*** state. When a new packet is sent, the main FSM changes its state to ***GENERATE_PACKETS***.

In ***GENERATE_PACKETS*** state, the registers of the table3.1 that represent the headers of data link, network and transport layers are defined by the user's options. The user's choices are assigned to the registers though the register system. Depending on the choices, different types of packet format are constructed. This state lasts one clock.

ALL_TO_ONE is the next state, in which the total header size is calculated, and also all the header registers are aggregated into one register (all_together[0:463]) in order to be ready for cutting into pieces of 64 bits. Like the previous state, *ALL_TO_ONE* state lasts only one cycle.

The last state (*SEND*) of the state machine sends the header data to the next module (Send module) that is responsible for cutting the headers and payload into chunks of 64 bits and sends them to the next module (Output queue). The state machine is initialized to *WAIT* state when Send module finishes its work. The second FSM is described in the 3.2.6 subsection of the implementation of Send module.

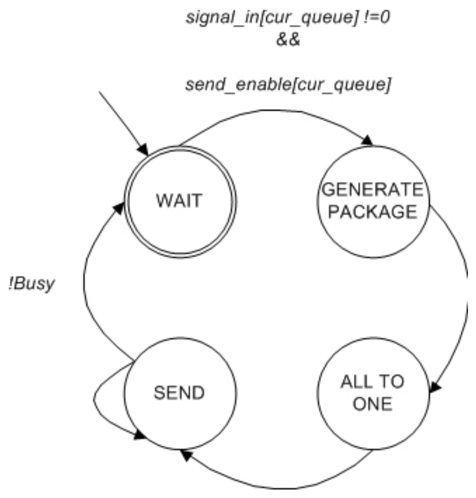


Figure 3.5: Packet Generator module FMS

Module Functionality

As mentioned in paragraph 3.2.4 of the Rater module description, Packet Generator module has a mechanism that even when the module is busy by constructing and sending a packet, it does not lose track of the requests for generating further packets. This is achieved by using a 10-bits register for each four flows which is called *signal_in*. This means that 1024 requests can be on standby for generation. Every time a request comes from the Rater module for generating a packet by the *generate_signal* wire this register is augmented by +one. Whenever a generation request is served, by sending a new packet of this flow, the register is decreased by -one. As long as the flow's value of each *signal_in* is bigger than zero, it means that there are new packets that are waiting to be sent. Because of this mechanism the Rater module and the generation module are not interdependent.

In order to fully utilize the card's capabilities and to extend the project's abilities, four multiple flows are implemented with the use of the time division. In each time slot, a packet of a flow is generated and sent. There is a round robin register, which is called *current_flow* that indicates which distinct flow will be generated. The policy of the time shifting is very simple and maximises the time slots. Whenever there are requests for generating packets of a flow and the register *current_flow* is equal with this flow, a new packet of the *current_flow* is generated and sent. If there is no flow with generation requests, then the *current_flow* register changes its value in the next flow. After sending a packet, the *current_flow* always changes value. So the duration of the time slots are dependent on the packet's size. The larger the packet is, the longer it takes.

Table 3.1: Size of each header register

reg/wire	Size in bytes	Name
reg	28	arp_header
reg	4	payload
reg	14 or 18	mac_header
reg	20	ip_header
reg	8	udp_header
reg	12	rtp_header
reg	8	module_header

As shown in table 3.1, each header has a register which should be filled. The size of each register is depended on the header size that it represents. There are header fields that always have the same value; there are fields that are filled by the user's parameters, and then there are some fields that should be calculated in order to be assigned to the header registers. The tables 3.2, 3.3, 3.4, 3.5, 3.6 show the values that are assigned to each header depending on the user configurations and the IEEE standards.

Table 3.2: Fields and values of MAC header

MAC Header Values		
802.3	mac_destination	User choice
	mac_source	User choice
802.1Q	QoS type	0x8100
	CoS value	User choice
	CFI	0
802.3	VID	0
	MAC TYPE	0x800 or 0x0806

Table 3.3: Fields and values of IP header

IP Header Values	
Version	4
IHL	5
Differentiated Services	User choice
Tolat length	Calcuated value
Identification	0
flags	0
Fragment Offset	0
TTL	255
Protocol	17
Header Checksum	Calculated value
Source IP address	User choice
Destination IP address	User choice

Table 3.4: Fields and values of UDP header

UDP Header Values	
Sourch UDP address	User choice
Destination UDP address	User choice
Length	Calculated value
Checksum	0

Table 3.5: Fields and values of RTP header

RTP Header Value	
Version	2
Padding	0
Extension	0
CSRC count	0
Marker	0
Payload Type	0 or 4 or 15 or 18
Sequence Number	Calculated value
Timestamp	Calculated value
SSRC	0xAD0F01AD

Table 3.6: Fields and values of ARP header

ARP Header Values	
Htype	1
Ptype	0x800
Hlen	6
Plen	4
Operation	User choice
SPA	User choice
THA	User choice
TPA	User choice

As shown in the above tables, there are some fields that are filled with the user's choices which comes from the register system and other fields that are always filled with the same value and should not be changed because of the IEEE standards. There are fields in the headers IP, UDP and RTP that should be calculated from the hardware in order to be assigned to the header, as well. At the IP header the fields that the NetFPGA should calculate are length and checksum. The length field contains the length of the IP, UDP header, plus the payload size in bytes. Finally, as the size of the headers is standard the length calculation is :

$$length = 28 + payload\_size\_bytes[cur\_queue\_stored] \tag{3.1}$$

The calculation of the checksum demands two cycles of calculations. At the beginning, the checksum field is initialised a zero.In the first cycle the header of the IP is cut in chunks of 16 bits and they are summed into the 19-bits temp register.

$$\begin{aligned} temp[0:18] = ((&ip\_header[0:15] + ip\_header[16:31]) + \\ &(ip\_header[32:47] + ip\_header[48:63])) + \\ &((ip\_header[49:79] + ip\_header[80:95]) + \\ &(ip\_header[96:111] + ip\_header[112:127])) + \\ &(ip\_header[128:143] + ip\_header[144:159]) \end{aligned} \tag{3.2}$$

In the second cycle, if the temp register produced a number bigger than 16 bits, the extra bits are summed up to a 16-bits result (sum register) which is then subtracted out of 0xFFFF.

$$sum[0:15] = temp[3:18] + temp[0:2] \tag{3.3}$$

$$ip\_header[80:95] = 0xFFFF - sum[0:15] \tag{3.4}$$

In the UDP header the field that the hardware should calculate is the length. The length field contains the UDP header's length plus the payload size in bytes.

Finally, as the size of the headers is standard the length calculation is :

$$length = 8 + payload\_size\_bytes[cur\_queue\_stored] \tag{3.5}$$

The Sequence Number and Timestamp values of the RTP header are calculated in the hardware. The Sequence Number of the RTP is initialized at zero. Every time a new packet is sent, the Sequence Number value is increased by +one. For the calculation of the timestamp a 16-bit counter (time_stamp) is used that is increased by +one in each clock. Whenever it is time to fill the header field, the current value is marked. The the time stamp precision is equal to the NetFPGA's clock period, which is 8ns.

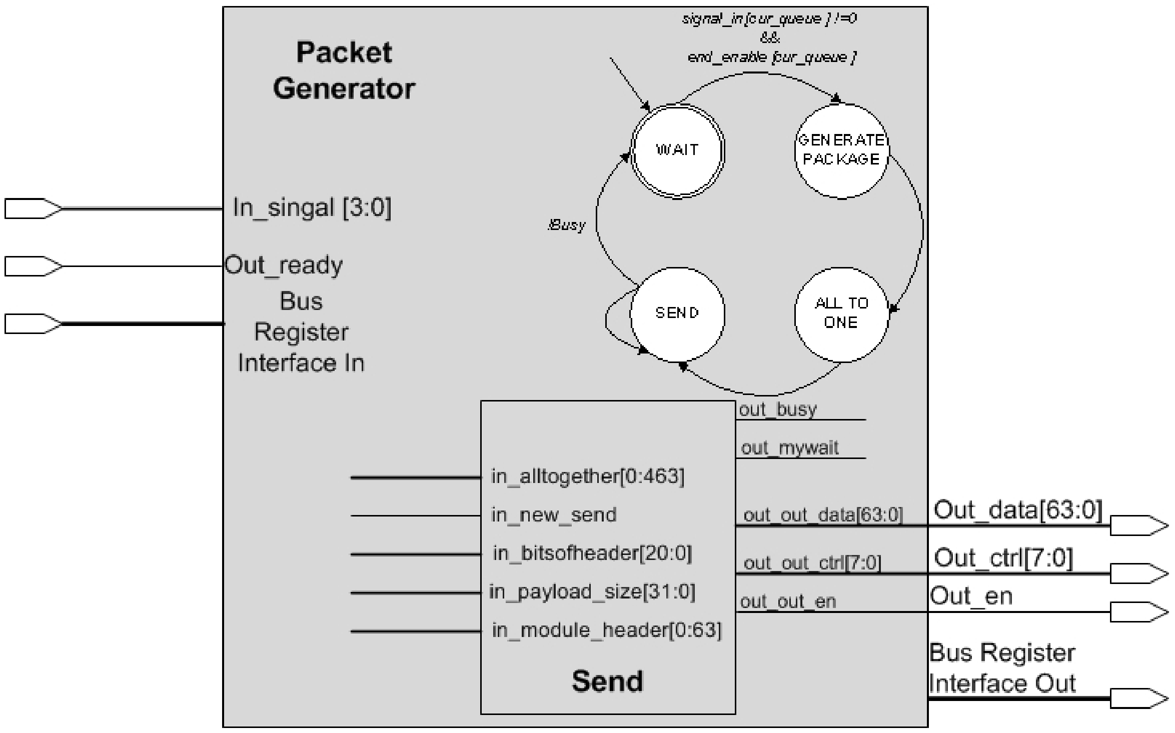


Figure 3.6: Packet generator module structure

During the Packet Generator FSM execution, the headers are filled as described above. Depending on the preferences of the user, the packet value of the header and the size change. The signals that can change the final header size are: cos_val, rtp_enable, arp_enable. If the cos_val is different to zero, the 802.1Q header is added above the MAC header. Also, when the rtp_enable is one, the RTP header is added above the UDP header. Finally, the arp_enable signal produces a packet with MAC and ARP header only. Whenever the packet is ready to be sent the signal new_send sends a pulse to the Send module. The valid registers data: alltogether[0:463], bitsofheader[20:0], payload_size[31:0] and module_header[0:63] are passed to the Send module. The Send module receives this information and sends the packet into 64-bits chunks to the next module. The figure 3.6 presents the Packet Generator module's wires and connection.

3.2.6 Send Module

After the construction of the packet's header, it is the turn of the packet to be divided into 64-bits chunks in order to be sent to the next module (My Output Queues). This work is achieved by the SEND module.

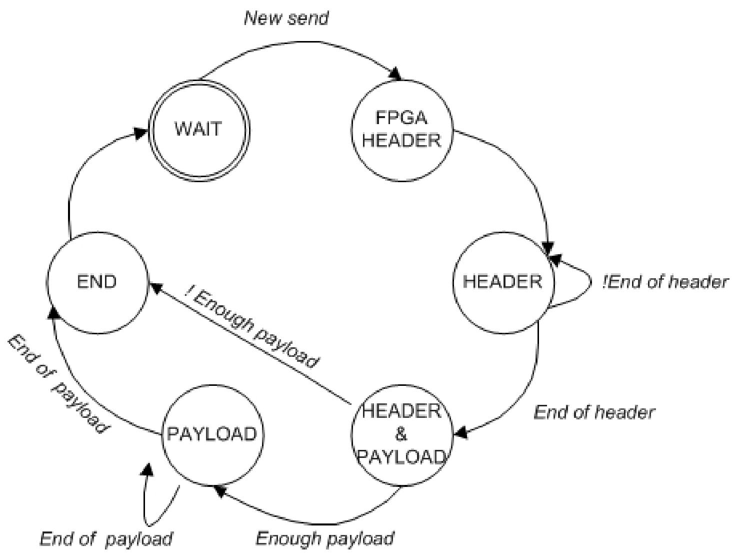


Figure 3.7: Send module's FSM

State Machine

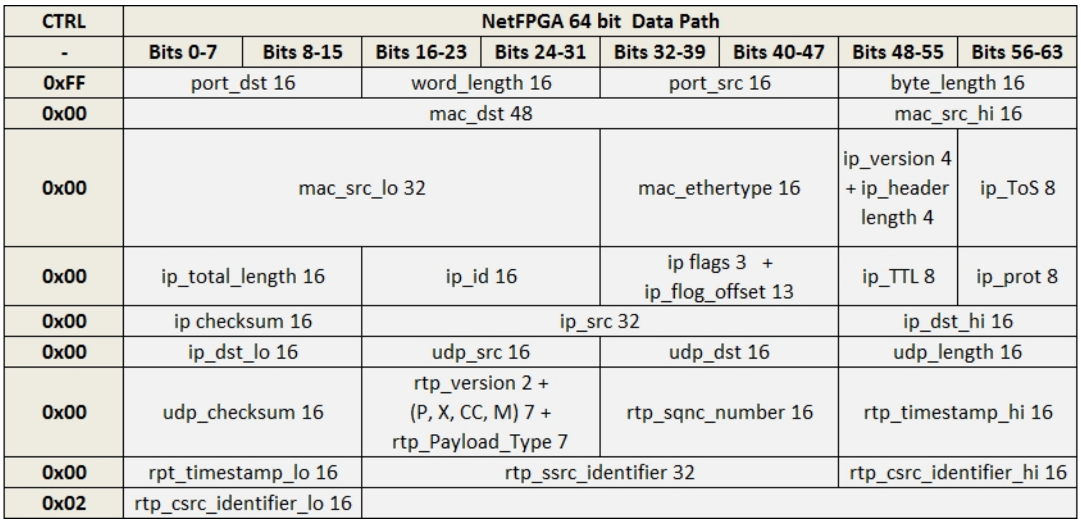
The Send module's FSM has six states: ***WAIT***, ***FPGA_HEADER***, ***HEADER***, ***HEADER_PAYLOAD***, ***PAYLOAD***, ***END***. Upon reset the FSM is at ***WAIT*** state. At this state, the state machine awaits for the input wires of the module, *in_ alltogether*, *in_ bitsofheader*, *in_payload_ size*, *in_ module_header* to take a valid value. These wires have all the necessary information for sending the packet into chunks to the next module. As the names of the wires indicate, *in_ alltogether* contains all the packet headers, *in_ bitsofheader* contains the header size in bits, payload_size indicates the payload size in bits, and ultimately *in_ module_ header* contains the packet module header. When the wires values are valid (*in_ new_ send* = 1), the state machine changes the state to ***FPGA_HEADER***. At this state the module header (64 bits) is stored at a FIFO before being sent to the next module where it can receive the data. ***FPGA_HEADER*** lasts only one clock cycle.

The next state is ***HEADER***. At this state the wire *in_ alltogether* that contains the packets' headers is cut in 64-bits chunks. Each piece is stored at the FIFO for forwarding to the next module when it will be available. The duration of this state depends on the size of the headers. The next state is *header_payload*.

At *header_payload* state the chunk that is stored into FIFO contains as much header information and payload information as needed for the compilation of 64 bits. This state lasts only 1 cycle. Because the header is cut in 64 bits, and there is a possibility the header may not be a multiple of 64 there may not be adequate bits for completing a 64-bits chunk. Based on this assumption the next state can either be *payload* or *end*.

At the *payload* state the chunk that will be stored into FIFO contains data only from payload. The payload value is a 32-bit register with a sequence number that is repeated as many times as necessary. If the payload is not a multiple of 64, it uses part of the register to fill the packet. Payload size should be greater than 18 Bytes (minimum packet size) and smaller than 1472 Bytes (depending on the packet type) for a UDP packet. Depending on the user's choices, different packet types and flows are generated.

The figure 3.8 shows the data value and the control value of an RTP packet with zero payload size transition.



CTRL	NetFPGA 64 bit Data Path							
-	Bits 0-7	Bits 8-15	Bits 16-23	Bits 24-31	Bits 32-39	Bits 40-47	Bits 48-55	Bits 56-63
0xFF	port_dst 16		word_length 16		port_src 16		byte_length 16	
0x00	mac_dst 48						mac_src_hi 16	
0x00	mac_src_lo 32				mac_ethertype 16		ip_version 4 + ip_header length 4	ip_ToS 8
0x00	ip_total_length 16		ip_id 16		ip flags 3 + ip_flog_offset 13		ip_TTL 8	ip_prot 8
0x00	ip checksum 16		ip_src 32				ip_dst_hi 16	
0x00	ip_dst_lo 16		udp_src 16		udp_dst 16		udp_length 16	
0x00	udp_checksum 16		rtp_version 2 + (P, X, CC, M) 7 + rtp_Payload_Type 7		rtp_sqnc_number 16		rtp_timestamp_hi 16	
0x00	rpt_timestamp_lo 16		rtp_ssrc_identifier 32				rtp_csrc_identifier_hi 16	
0x02	rtp_csrc_identifier_lo 16							

Figure 3.8: CTRL and DATA registers' Values during the RTP packet sent

3.2.7 My_output_queue Module

The Output_Queue module of the NetFPGA's Verilog library has the same functionality as this new my_output_queue module. It reads the module headers of the packets and it decides to which output queue the packets will be stored and forwarded. The library module has more functionalities like checking the size of the packets, storing the packets temporarily into sRAM and reading from it; processes that require a lot of cycles in order to be executed. Simulations reveal that the library module demands more time for processing a packet than the theoretical limits and as a result, we cannot fully utilize the card as it was impossible to generate 4 Gbps flows (four 1 Gbps flows). For example, for the minimum packet size of a 60-bit UDP packet the theoretical limit for processing and forwarding is 21 cycles for each packet.

$$(125.000.000/4)/1.488.095 = 21cycles$$

As the propagation delay is 9 cycles, it leaves only 12 cycles for packet processing. The library module needs 15 cycles for packet storing and processing. This means it is impossible to generate four flows with a full speed of 1Gbps. Because of these limitations, the module had to be redesigned and reimplemented.

Since this module implementation is more simplified than the library module's implementation, it needs less time to decide which output port to send the packets. The buffering process of reading and writing on the sRAM demands a lot of cycles. However,the buffering process was replaced with FIFO implemented in the FPGA, which is faster but smaller in size. It is not important to check for TTL fields or packet size as the packets are generated identically. Finally, the module's decision of the output port is done in 1 cycle.

State Machine

As the figure 3.9 shows, the state machine of my_output_queues module consists of the following states: *WAIT*, *DST_PORT_CALC* and *SEND*. The state machine is initialized into *WAIT* state. When the first part of the packet comes, it is stored temporarily into a FIFO. When the FIFO has stored packets, the state changes from *WAIT* to *DST_PORT_CALC*. At *DST_PORT_CALC* state the destination(s) output queues determine where the packet should be written. The packet will then be stored temporarily into that FIFO, but only if the FIFO of the output queues are not full. This state lasts only 1 cycle.

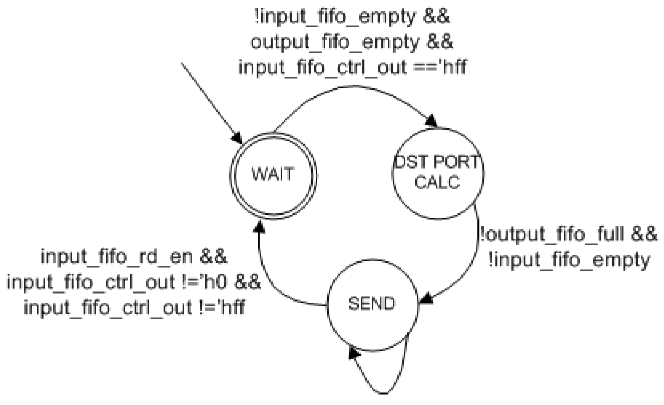


Figure 3.9: My Output Queues module FSM

After that, the state machine changes its state to *SEND*. At this state, all the packet chunks are written in the temporary FIFO as long as FIFO is not full. The state changes back to *WAIT* when all the packet chunks are forwarded and stored at the respective output(s) FIFO(s). This is indicated by the input_fifo_ctrl_out wire. When its value is different than 0x00 or 0xFF, the packet's end is indicated. As long as the output FIFOs have packets and the Output_Queue modules are available the packets are forwarded to them.

3.3 Packet Format

The Packet generator module can generate two different packet formats: UDP and ARP. Data Link layer header can be two different types, either 802.3 (TYPE 0x0800), or Quality of Service (QoS, 802.1p, TYPE 0x8100) by choosing values of Priority Code Point from 0 (best effort) to 7 (highest); 0 represents the lowest priority. These values can be used to prioritize different classes of traffic (voice, video, data, etc.) [18].

In addition, at Network layer, besides the source and destination IP, the user may also define the Differentiated Services Code Point (DSCP), originally defined as the Type of service field - ToS. The ToS field specifies a datagram's priority and requests from a router a low-delay, high-throughput, or highly-reliable service. Based on these ToS values, a packet would be placed in a prioritized outgoing queue, or take a route with appropriate latency, throughput, or reliability.

VoIP calls can be simulated as well, by adding a Real Time Protocol (RTP) header at the higher layer of the UDP. RTP is used to carry data that has real-time properties. The field of payload type identifies the format (G.711, G.723.1, G.728, etc) of the RTP payload and determines its interpretation by the application.

The generation of ARP packets is important for this project as it is needed to inform the network devices (router, etc) about the existence of a virtual interface, in order to be able to route the packets. The ARP packets that are produced are in reply to ARP requests and indicate the assignment between the IP and the source mac address of the port of the card.

The packet format is defined by the user by choosing and writing the registers through the register system. User defines the parameters of the Data Link layer, Network layer and Transport layer. The parameters that should be defined are presented in details at subsection 3.2.5.

3.4 Register System

The register system collates and generates addresses for all the registers and memories in an NetFPGA project. It is the way for the NetFPGA card to communicate with the host computer. Each module of the project has an xml file which declares the name and size of each module register. Some registers are for input, which should be written by the user in order to parameterize the generated flow and others are for output which gives the calculated data of the card back to user. The register are accessed through the GUI system by the user. The tables 3.7, 3.8, 3.9 specifies the functionality of each modules' registers.

Table 3.7: Packet Generator Module's registers

Packet Geretor Module		
Register Name	**Description**	**Size in bits**
payload_size_bytes	Payload size in bytes.	32
dstip	Destination IP address by decadal representation.	32
srcip	Source IP address by decadal representation.	32
dstport	Destination port (field of transport layer).	32
srcport	Source port (field of transport layer).	32
dstmac_high	High 16 bits of destination MAC address.	32
dstmac_low	Low 32 bits of destination MAC address.	32
srcmac_high	High 16 bits of source MAC address.	32
srcmac_low	Low 32 bits of sourch MAC address.	32
arp_enable	If equal to 1, activates the ARP packet generation.	32
arp_opcode	If equal to 1, specifies the ARP packet's type	32
rtp_enable	If equal to 1, enables RTP packet generation .	32
pt	If equal to 1, specifies the RTP flow's type	32
cos_value	Cos value.	32
tos	If different to 0, quality of service is enabled. It defines the TOS field's value.	32
fpga_dst_port	Packet's NetFPGA Destination port.	32
send_enable	If equal to 1 packets are generated	32
num_packets_generated (output)	Generated packets number.	32

Table 3.8: Rater Module's registers

Rater Module		
Register Name	**Description**	**Size in bits**
clk_limit_3	Packet generator rate limit of flow 3.	32
clk_limit_2	Packet generator rate limit of flow 2.	32
clk_limit_1	Packet generator rate limit of flow 1.	32
clk_limit_0	Packet generator rate limit of flow 0.	32
packets_generated_3 (output)	Number of generated packets of flow 3 of the Rater module.	32
packets_generated_2 (output)	Number of generated packets of flow 2 of the Rater module.	32
packets_generated_1 (output)	Number of generated packets of flow 1 of the Rater module.	32
packets_generated_0 (output)	Number of generated packets of flow 0 of the Rater module.	32

Table 3.9: Statistic Module's registers

Statistic Module		
Register Name	**Description**	**Size in bits**
bps (output)	Throughput in Bits per Second	32
pps (output)	Throughput in packets per Second	32
jitter (output)	Gives the jitter per Second	32

3.5 Packet Generator Graphical User Interface

GUI declares the generated flow by writing the aforementioned registers, as well as allows to read them and ultimately, receive the statistics that the card has calculated. The software is responsible for providing correct parameters to the hardware through the register system. Two software components have been implemented; one responsible for the read and write operation of the card's registers, and one responsible for the results' graphical representation.

The first is implemented in C. This program reads the auto-generated register.h file generated from the hardware compilation which, in turn, contains the card's register addresses. The registers are written, along with the records of the user's values, and read in order for the GUI to display the statistics results. The program input is the user's register declarations specified above and the output is the results of the statistics calculation. The program is responsible for writing and reading the appropriate registers.

The second program is implemented in Java. As shown in figure 3.11 a GUI is also implemented that simplifies the process of registers' definition. The user has unfilled text boxes and drop menus with the parameters that he should fill.

For being more ergonomic, functions such as store/load of previous parameters, calculation median, minimum and maximum value are implemented. Also, the GUI program is used for the graphical representation of the statistics results as well as the calculation of the the average, standard deviation, minimum and maximum of metrics. The possibility to store the results at a text file for further processing is provided.

The communication between the client program and the server is accomplished through TCP sockets. This way, it is not necessary for the end user to use a computer that has a NetFPGA card installed, but to know the IP address of a computer with a NetFPGA card. The figure 3.10 shows the connection between the computers of GUI and NetFPGA.

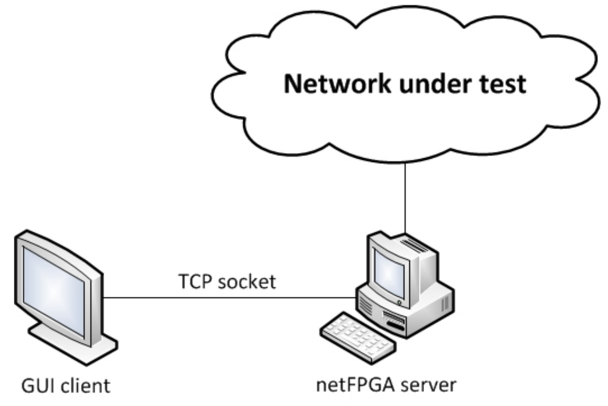


Figure 3.10: Connection between GUI client and NetFPGA server

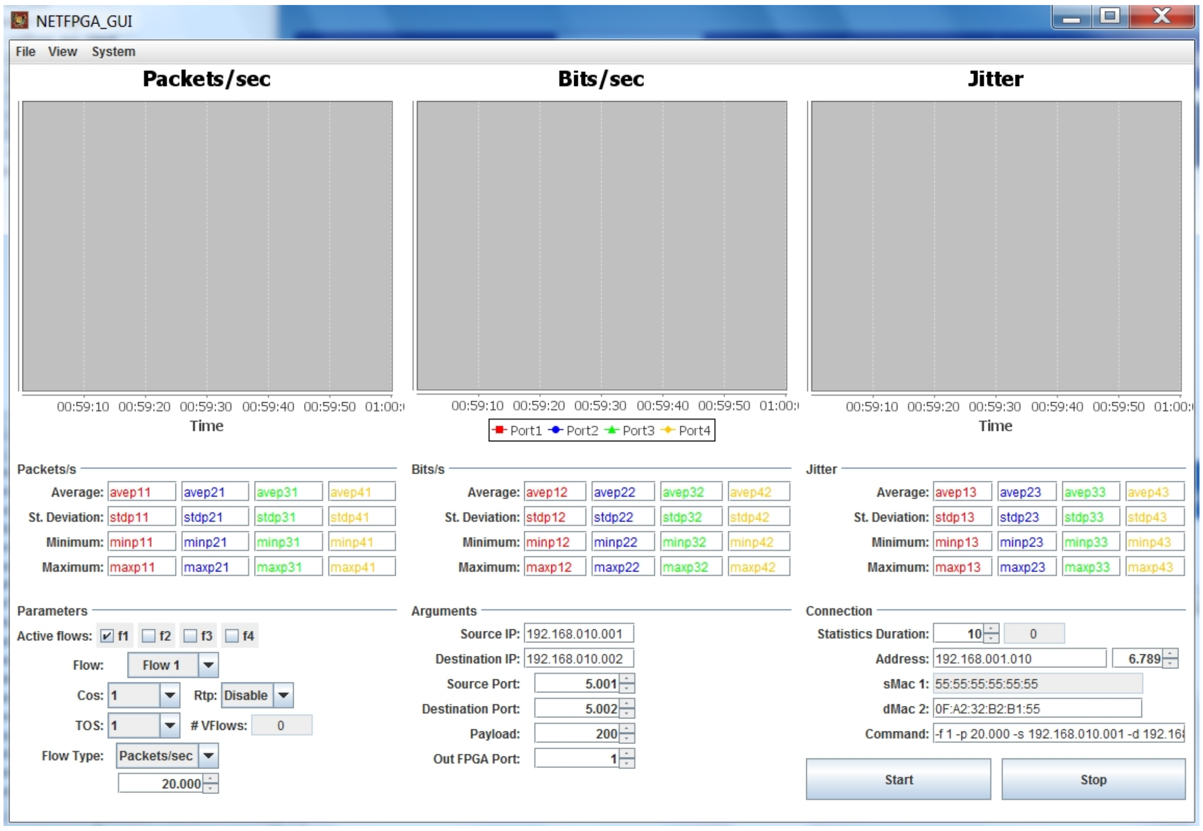
3.5.1 Interface description

The user graphical interface is divided into two parts, the representation results part at the upper part of the window and the generated flows' set part which is at the lower part of the window.

The upper part illustrates the run-time calculated results of the four NetFPGA interfaces. While the card receives the generated packets, the Packets per second, Bits per second and Jitter per second graphs are constructed. Each NetFPGA Ethernet port is represented with different color. The statistics of Average, Standard deviation, Minimum and Maximum that are placed under each graph are calculated and represented with different color, as well. From the View menu, the wanted received flows can be represented. The received statistics can be saved for further future processing through the File menu.

At the lower part of the window, the generated flows are parameterized. The flow type (UDP, RTP, etc.) with their parameters (source / destination IP, MAC and port, etc.) is chosen. The choice of Activated Flows supports multiple generated flows simultaneously. From the Load button on the System menu pre-saved parameters are loaded. The Start and the Stop buttons activate and deactivate the generated flows.

From the Initialization button on the System menu, the four NetFPGA interfaces are set with their IP. The Load button loads these values, as well.



Packets/s

Average:	avep11	avep21	avep31	avep41
St. Deviation:	stdp11	stdp21	stdp31	stdp41
Minimum:	minp11	minp21	minp31	minp41
Maximum:	maxp11	maxp21	maxp31	maxp41

Bits/s

Average:	avep12	avep22	avep32	avep42
St. Deviation:	stdp12	stdp22	stdp32	stdp42
Minimum:	minp12	minp22	minp32	minp42
Maximum:	maxp12	maxp22	maxp32	maxp42

Jitter

Average:	avep13	avep23	avep33	avep43
St. Deviation:	stdp13	stdp23	stdp33	stdp43
Minimum:	minp13	minp23	minp33	minp43
Maximum:	maxp13	maxp23	maxp33	maxp43

Parameters

Active flows: [✓ f1] f2 f3 f4
Flow: Flow 1
Cos: 1 Rtp: Disable
TOS: 1 # VFlows: 0
Flow Type: Packets/sec
20.000

Arguments

Source IP: 192.168.010.001
Destination IP: 192.168.010.002
Source Port: 5.001
Destination Port: 5.002
Payload: 200
Out FPGA Port: 1

Connection

Statistics Duration: 10 0
Address: 192.168.001.010 6.789
sMac 1: 55:55:55:55:55:55
dMac 2: 0F:A2:32:B2:B1:55
Command: -f 1 -p 20.000 -s 192.168.010.001 -d 192.16

Figure 3.11: GUI screen-shot

Chapter 4

Evaluation

The Evaluation section contains the experiments performed in order to test the NetFPGA 1G system's performance, the power consumption, the flow's production accuracy and the network devices performance such as routers, switches and wireless access points. In addition, there is a comparison of the performance and the accuracy between commercial (Ix Chariot) and open source (iPerf, TTCP) software applications/solutions.

At section 4.1 the theoretical limits presented, then the experiments are described(4.2) in detail and finally the experiment results and conclusions follow(4.3).

4.1 Theoretical limits

Maximum theoretical throughput of 802.03 and 802.11g standards are calculated in order to be used as reference values for the experiments' execution over Ethernet. The payload limit size is between 18 and 1472 bytes as the MAC layer segment defines[19].

The maximum theoretical limits of a 1 Gbps throughput connection for 802.03 are calculated with the following formula:

$$pps = \frac{Throughput}{Packet\ Size} = \frac{Throughput}{MAC\ header + IP\ header + UDP\ header + payload} \tag{4.1}$$

$$Throughput = (Frame\ size - 24) * pps \tag{4.2}$$

For the measurements the packet size used is outlined in the table 4.1:

Table 4.1: Pps and Throughput theoretical limits vs Payload Size

Ethernet Frame Size Bytes	Packet Useful data Bytes	Payload Size Bytes	Max pps	Max Throughput Bits per second
84	60	18	1.488.095	714.285.600
300	276	234	41.667	920.000.736
1000	976	934	125.000	976.000.000
1538	1514	1472	81.274	984.390.688

The bigger the payload size, the better the network's sources are utilised. However, the theoretical total throughput is not achieved because of the packet's overhead, preamble, start of frame delimiter, inter-frame gap and frame check sequence with total size 24 bytes. On the other hand, with smaller payload size, smaller throughput is achieved because of the useful information's fraction and the extra payload overhead. Fig. 4.2 shows the 802.03 link's theoretical throughput and the fig. 4.1 shows the theoretical 802.03 link's pps.

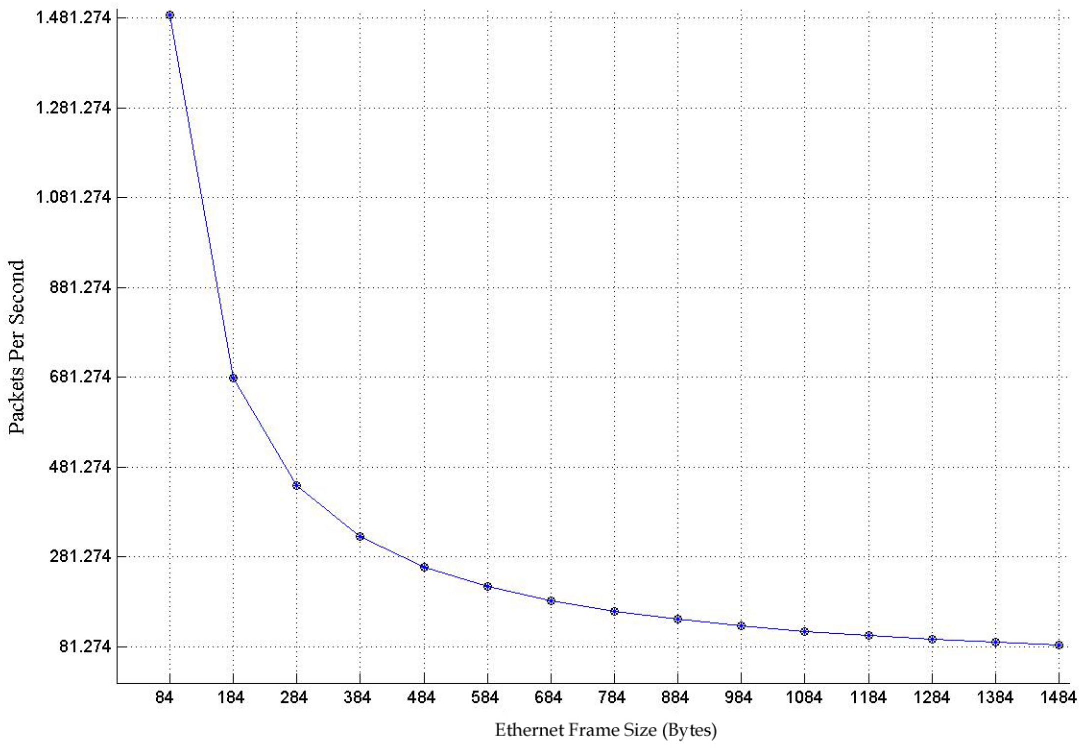


Figure 4.1: 802.03 flows' theoretical packets per second limits

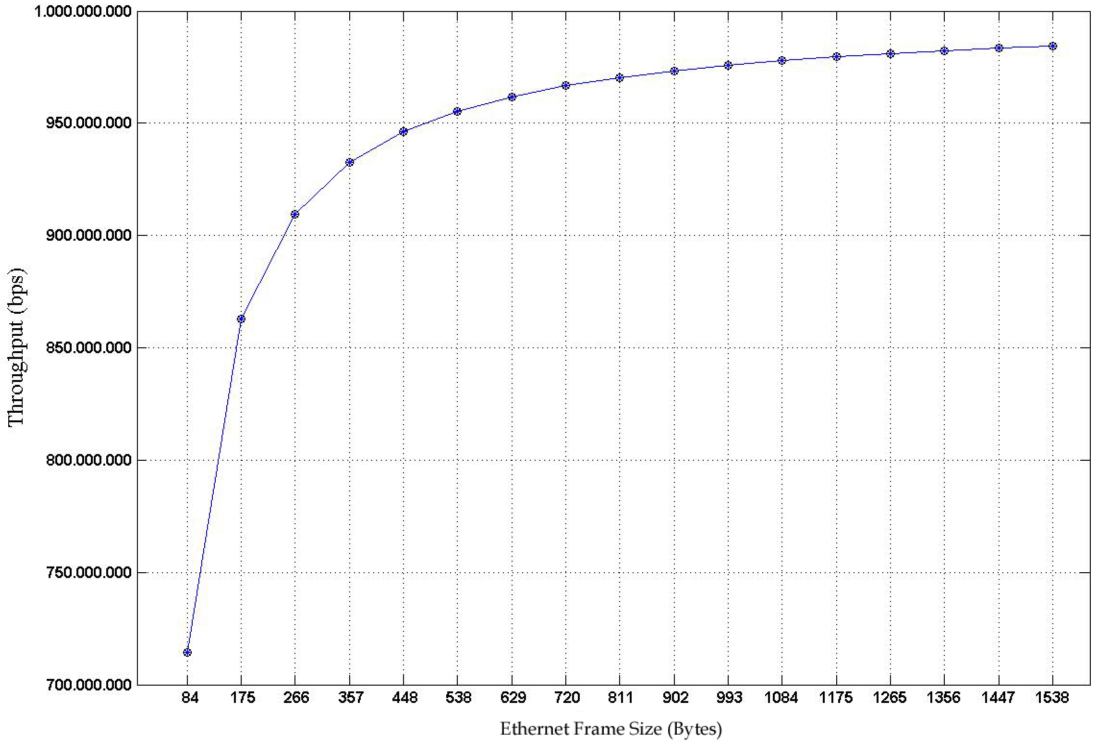


Figure 4.2: 802.03 flows' theoretical throughput limits

The IEEE 802.11g throughput limits are defined from the following formula using the time-values of the table 4.2. [20]:

$$U = \frac{t_d}{DIFS + t_{pr} + t_{tr} + SIFS + t_{pr} + t_{ack}} \tag{4.3}$$

The figure 4.3 shows the successful transmission of a single frame under 802.11g.

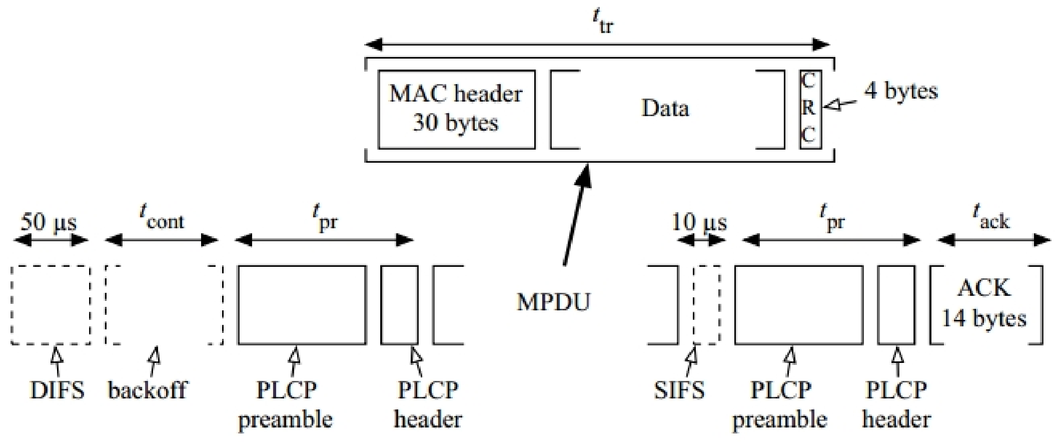


Figure 4.3: Timings of a single frame successful transmission under 802.11g

Table 4.2: Time-value of the parameters of the 802.11g

CW	0 μs
Slot	9 μs
DIFS	28 μs
SIFS	10 μs
tpr	16 μs

In figure 4.4 is represented the maximum throughput limits of IEEE 802.11g at different rates for different payload sizes, in the case of a perfect transmission without packet losses and packets retransmission.

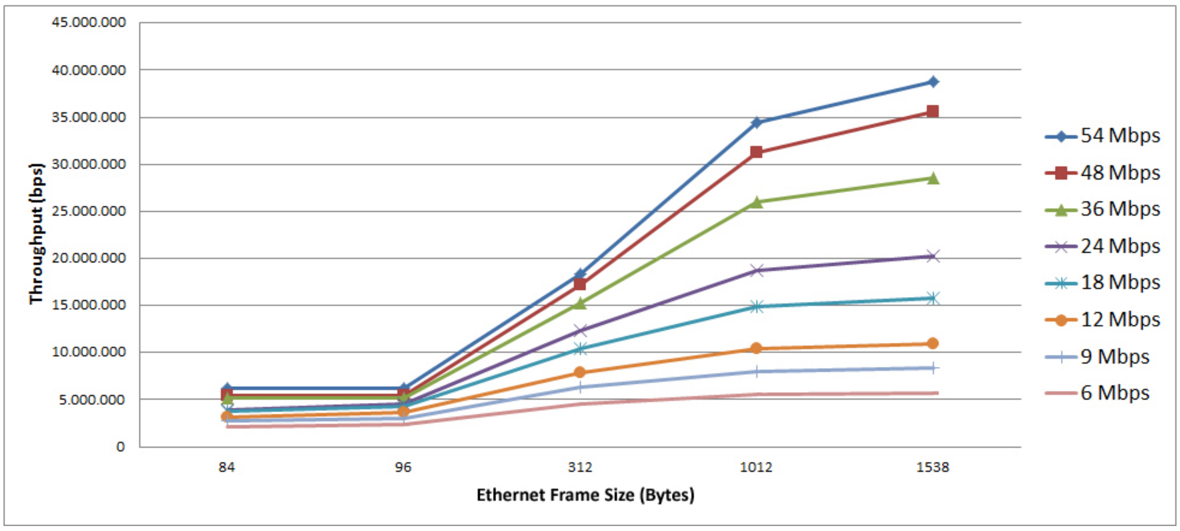


Figure 4.4: IEEE 802.11g maximum throughput limits

4.2 Experiments

4.2.1 Experiment 1: Theoretical limit of four NetFPGA flows

The experiment's goal is to check whether the card can produce 1 Gbps four independent flows, either with small payload size (maximum packets per seconds) or large payload size (fully utilization of Gigabit ethernet link) and whether it can process the flows simultaneously without packet losses or high jitter. The four interfaces of the card are connected directly ,without the use of any network device - switch or router-, in pairs of two, so each interface generates packets and receives flows for analysis, as shown in the above figure. Firstly, all the interfaces are parametrized to send the same flow (same packet/payload size) starting from the smallest packet size to the biggest packet size. The goal is to verify that the card can precisely generate from maximum packet per second to maximum throughput.

In the second version of this experiment the same goals are tested like at the first version. The difference is that the generated flows have different packet size. So each interface is set to send different flows and, as a result ,different flows are calculated for statistical analysis. Therefore, each of the four independent flows is put under test in order to verify its theoretical limits.

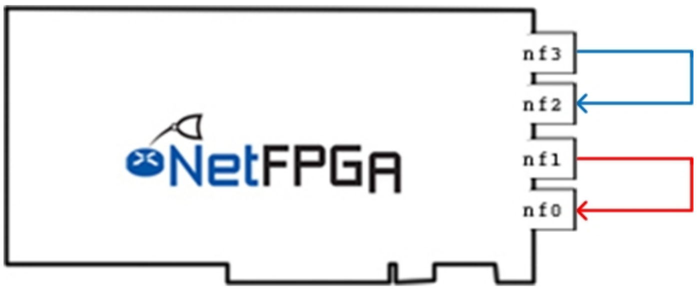


Figure 4.5: Experiment's 1 topology

4.2.2 Experiment 2: Cisco Switch SLM2008

The goal of the experiment is to test the performance of the Cisco Switch SLM2008. It is tested the capability of the switch to support multiple flows at the speed of 1Gbps. NetFPGA operates only at 1Gbps and it can not be connected with network devices of 100Mbps and 10Mbps interfaces [21]. In this case, aforementioned the switch is used as a converter from 1Gbps network connections to 100/10Mbps network connections and vice versa.

The switch's interfaces are configured in pairs in a different vlan(port 1 and 2 vlan 1, port 3 and 4 vlan 2). Then the switch receives the packets and forwards them to the next vlan depending of the destination MAC address of the packet. The flow that is sent is a normal UDP flow without extra headers. The payload of the experiment then changes from 18 bytes to 1472 bytes. The average throughput, pps and jitter are the performance values used to decide if the device is appropriate to use as a converter.

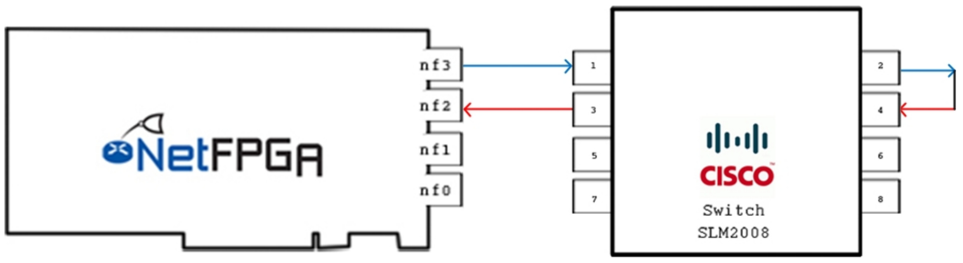


Figure 4.6: Experiment's 2 topology

4.2.3 Experiment 3: Switch vs Router mode

The experiment's aim is to check the potencial of the Linksys Wi-Fi Router WRT54GL, a low cost device, as a switch and as a router. For the first part of this experiment the device is parameterised as a switch. The four ethernet ports of the Linksys device work at 100Mbps speed so for connecting the NetFPGA with it the Cisco switch should interpolate between them. The Cisco SLM 2008 switch in this case is used as a converter from 1Gbps flows to 100Mbps flows and vice versa.

In the second part of the experiment, the Linksys WRT54GL is set to router mode. The device examines the incoming packets up to the level of the network layer(routing tables, TTL, checksum, etch) in order to route them. This process demands more calculations than the process of the packets in switch mode. In both cases the Cisco is set up according to experiment 2. As the experiment result will show at the section 4.3.2, Cisco switch can manage 1Gbps flows with perfect precision and as a result the use of the switch will not weather the final results.

Using different payload sizes will reveal if the device is able to manipulate packets with big payload size or flows of high packets per second rate. We further observe whether the extra network layer process of packet (router mode) affects the jitter and the ability of the device to achieve the theoretical limits. The figure 4.7 shows the connection between the experiment devices.

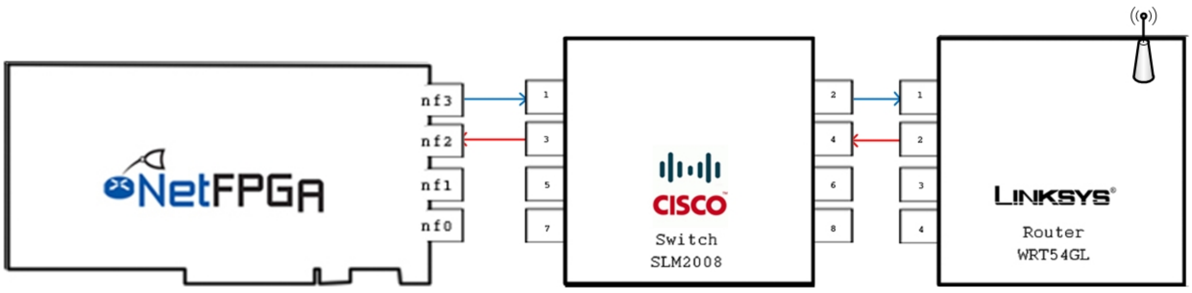


Figure 4.7: Experiment's 3 topology

4.2.4 Experiment 4: Cisco Catalyst Router 7606

Similarly to the previous experiment, the limits of the high cost router Cisco Catalyst 7606 are tested. The experiment results are compared with the results of the experiment 3 in order to find result differences between low and high cost devices. The experiment takes place in a simulated environment at FORTH Institute[22] where other flows pass simultaneously. In addition, during the runtime of the experiments, the CPU load is checked through the web interface of the router in order to identify if the generated flow needs more process power and makes the router reach its limits.

Two router interfaces are set in two different subnets. Each interface is then paired with a NetFPGA interface as the fig. 4.8 shows. The router receives the packets that the NetFPGA sends, analyzes them by reading the packet up to the level of the network layer in order to find the route path and routes the packets to the correct interface. When the packet is routed, it is received from the NetFPGA interface in order to be calculated and analyzed. The generated flow is a 1Gbps UDP flow with different payload size that changes from 18 bytes to 1472 bytes.

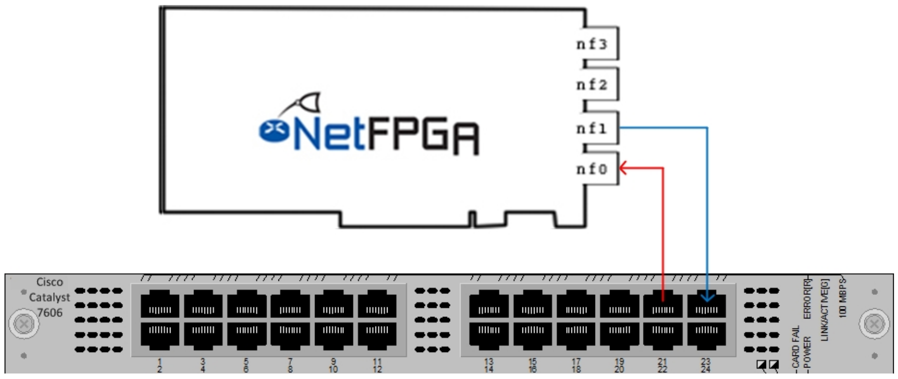


Figure 4.8: Experiment's 4 topology

4.2.5 Experiment 5: Wireless Access points

Similarly to experiment 3 - Switch vs Router mode, the same Linksys Wi-Fi Router WRT54GL is used. The experiment's aim is to check the wireless connection limits of two devices. Two Linksys devices are used, one as wireless access point and the other as wireless card. One NetFPGA port (send Ethernet port) is connected through the Cisco switch to one Ethernet interface of the first Linksys device (access point) and a second port of the NetFPGA (receive Ethernet port) is connected through Cisco switch to one Ethernet interface of the other Linksys device (wireless card) as the following figure shows. As the two devices are wirelessly associated, the packets of the first NetFPGA port go back to the other port through the wireless connection.

The devices can work at the stand of IEEE 802.11g ,which permits speeds of 54 Mbps, 48 Mbps, 36 Mbps, 24 Mbps, 9 Mbps and 6 Mbps. In order to achieve the maximum throughput, the flows of the NetFPGA is set as high priority flows which means that wait time of the queues is reduced. In addition, instead of using antennas we employ a proper coax cable with attenuators to directly connect the antenna outputs, in order to avoid the possible interference which produces packet losses, retransmissions and as a result lower bandwidth.

The experiments uses different payload sizes at all the available IEEE 802.11g speeds.

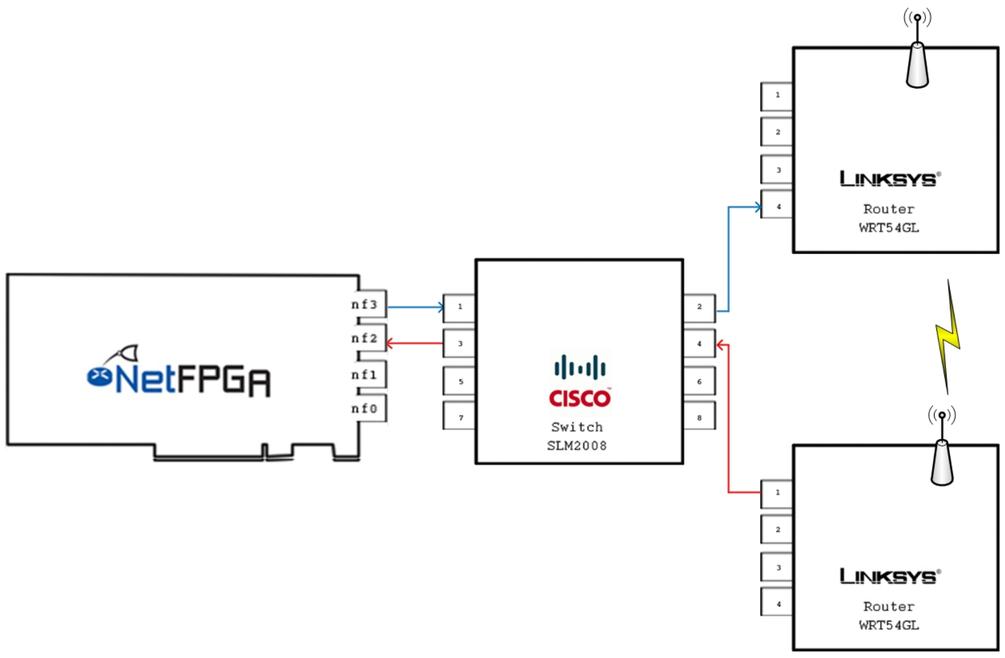


Figure 4.9: Experiment's 5 topology

4.2.6 Experiment 6: IPERF vs IxChariot

The experiment's goal is to compare the performance of a free open source software packet generator versus a commercial software packet generator. In this case, the performances of iPerf v.2.0.4 pthreads and the IxChariot 7.30 EA Build Number: 7.30.43.35 are tested.

IxChariote is a commercial close source packet generator that is used to measure network performance. IxChariot provides the ability to confidently assess the performance characteristics of any application running on wired and wireless networks [2]. On the other hand iPerf is a commonly open source network testing tool that can create TCP and UDP data streams and measure the throughput of a network that is carrying them. The network set and the experiment variables are the same as in experiment 5: Wireless Access points, but with a computer network interface instead of the NetFPGA. Because all the wired devices' interfaces work at 100Mbps there are no issues of incompatibility so the CISCO switch is not used as shown in the following figure. The computer used is an Intel Core i3-3225 @ 3.30 GHz and a Xeon E5-1620 (Quad Core, 3.60GHz Turbo, 10MB Cache). The Intel is used as a server and the Xeon ,which is a more powerful machine, is used as a client for both softwares.

Apart from the throughput statistics that the two different programs achieve, we also observe the CPU usage. All the unrelated kernel services are switched off and only the client and the server CPU usage of the programs is observed.

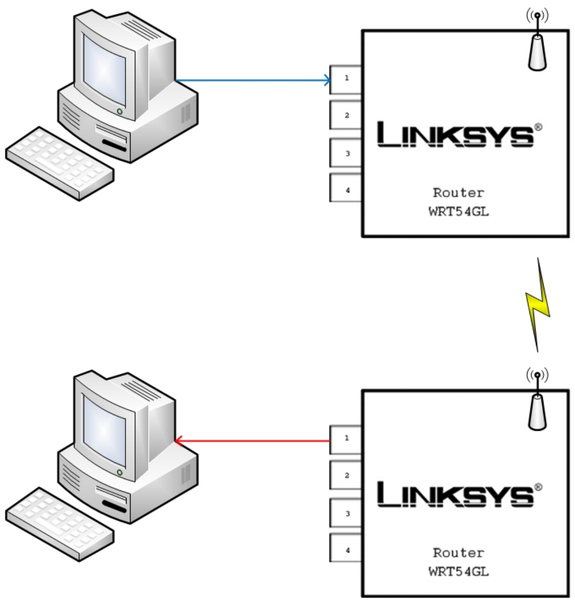


Figure 4.10: Experiment's 6 topology

4.2.7 Experiment 7: Power consumption

The experiment 's aim is to calculate the power consumption and the CPU load of the the computers that send and receive packets when network experiments are executed with NetFPGA and with software programs. In this case, the network structure doesn't matter as the power that the computer demands to generate the flow and to receive and calculate the received flows depends only on the implementation of the software. In addition, the power consumption of the CPU load is dependent only on the software implementation and not on the structure of the network experiment. To simplify the network topology the two computers are directly connected.

In this final experiment the NetFPGA and two free open source software programs, the iPerf and the TTCP, are used. In both cases, UDP flows are generated with the same packet size at maximum throughput. As regards the software programs ,the flows are set at 100Mbps because of the 100Mbps network cards of the computers. As regards the NetFPGA flows, they are set at 1 Gbps.

The power is calculated in both computers, both when generating the packets(client), and receiving and analysing the packets (server).During the measurements, all other processes of the computer are closed so that hardware resources would not be occupied and results would not be corrupted either. The reference power when the computer is idle is 80 Watts.

4.3 Results

4.3.1 Experiment 1: Theoretical limit of four NetFPGA flows

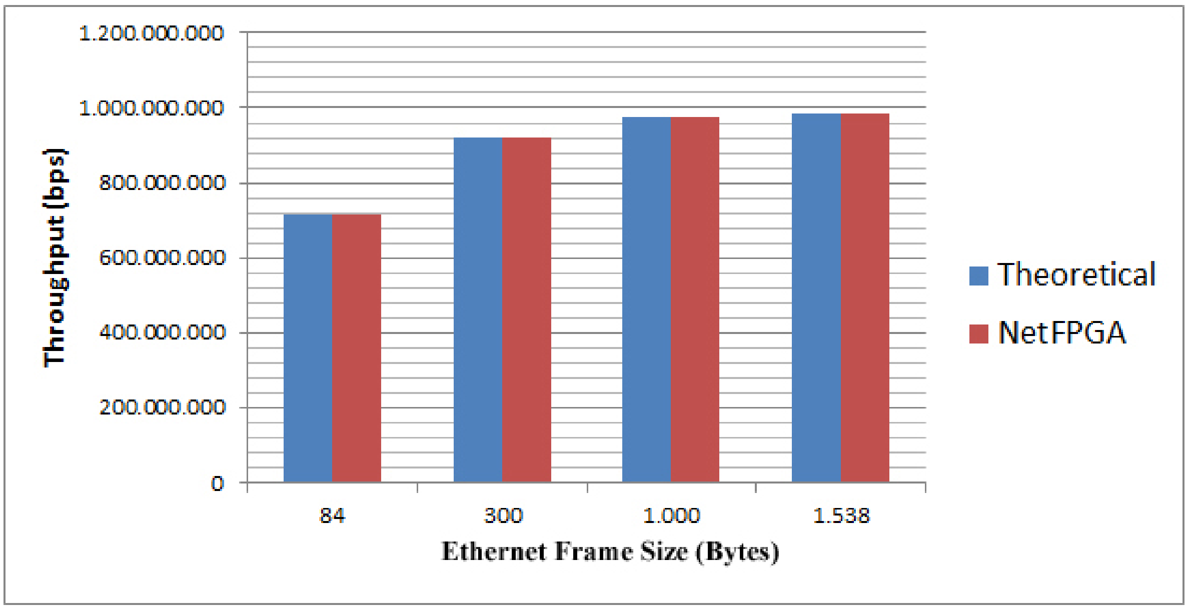


Figure 4.11: Experiment's 1 results. Throughput vs Ethernet Frame size

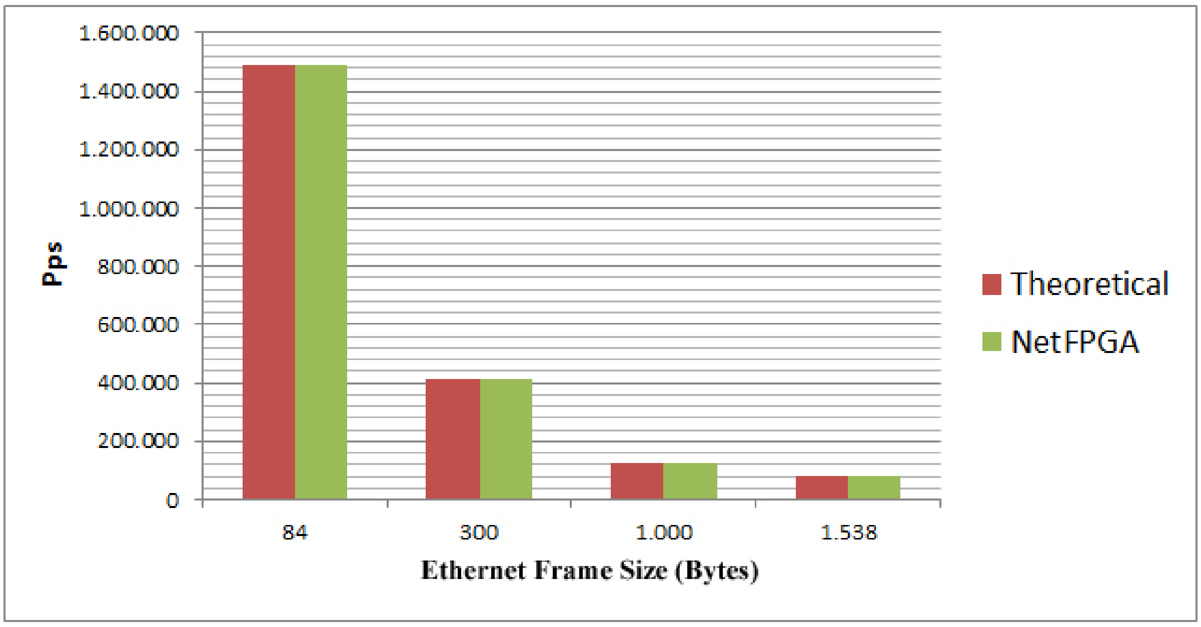


Figure 4.12: Experiment's 1 results. Pps vs Ethernet Frame size

This experiment is the most important as it proves the capability of the system to generate packets with high precision at zero jitter. In the first version of the experiment, the figures 4.11 and 4.12 show that the card can generate and calculate the data of four flows simultaneously at 1Gbps with high precision. The results of the packets per seconds match the theoretical calculation, and the jitter is always zero. This proves that the payload size affects neither the generation rate of the packets nor the process of the received packets.The results are collected either from the statistics of each receiving port or from the total data that is received from the module implemented after the Input Arbiter module.

The second part of the experiment shows that each flow can achieve its theoretical limits with jitter zero. These observations are identified in each statistic module of each corresponding receiving port.

In conclusion, the Packet Generator is capable to generate and receive four flows of 1 Gbps with high precision and zero jitter.

4.3.2 Experiment 2: Cisco Switch SLM2008

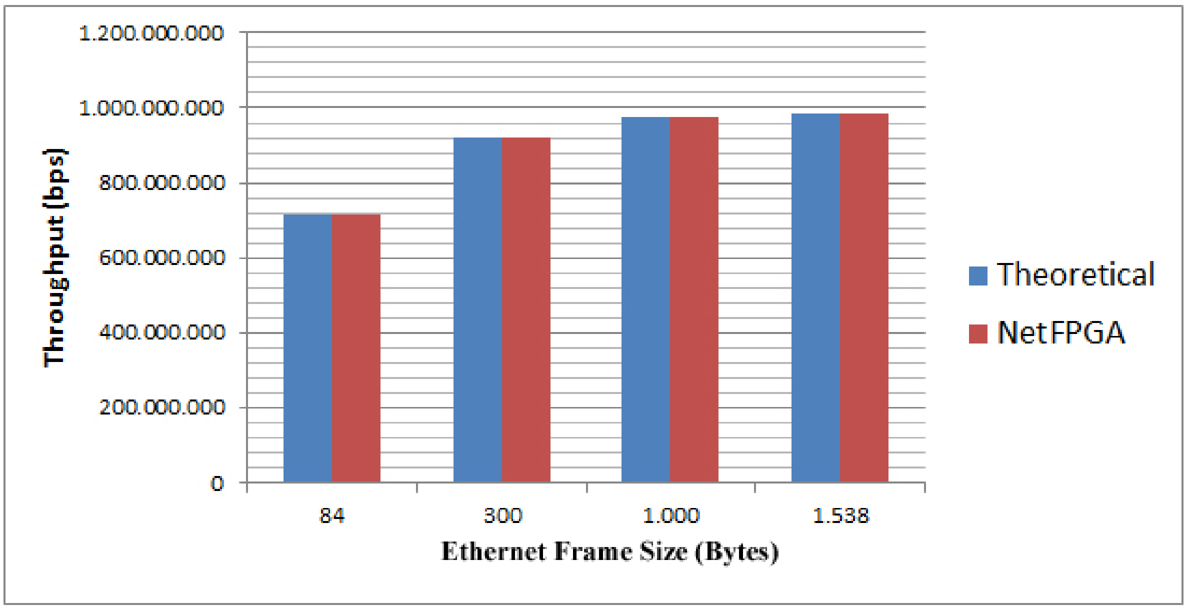


Figure 4.13: Experiment's 2 results. Throughput vs Ethernet Frame size

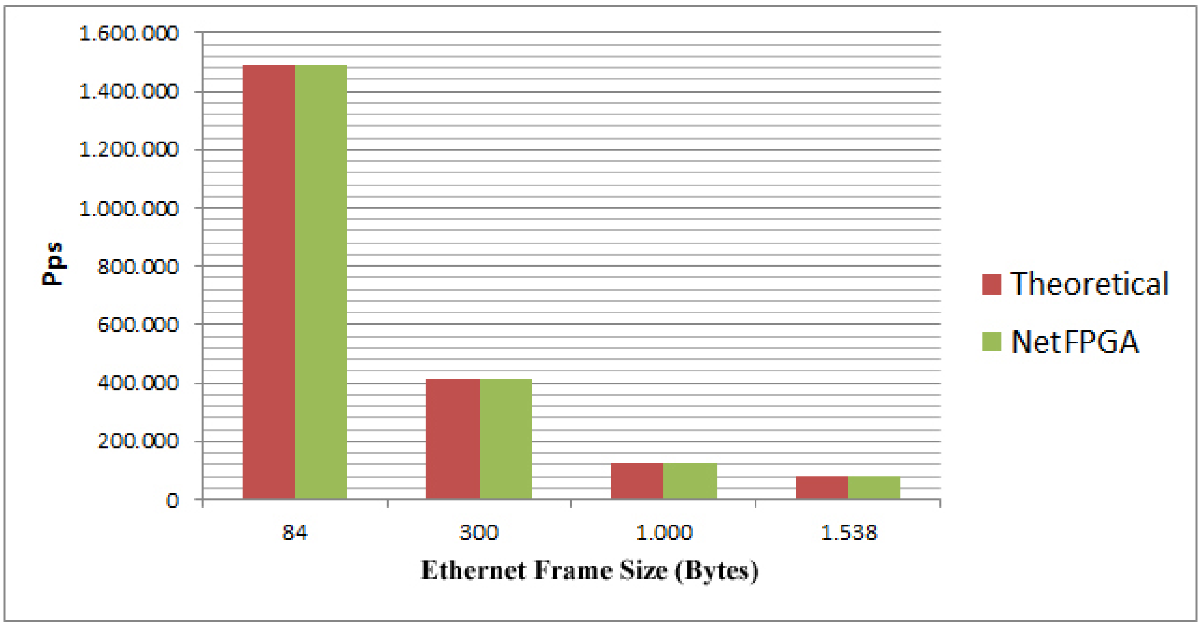


Figure 4.14: Experiment's 2 results. Pps vs Ethernet Frame size

The figure's results 4.13 and 4.14 show that the Cisco Switch SLM 2008 is capable of managing 1 Gbps flows. All the pps and throughput theoretical limits are achieved. The only anomaly observed is when the packet size is the smallest due to the 8-cycle NetFPGA jitter(8 * 8ns = 64 ns). This may be due to the large packet rate (1.488.095)and the packet buffering. However, jitter time is so small that is negligible . The experiment shows that this switch can be used as a network converter from 100Mbps flows to 1Gbps flows.

4.3.3 Experiment 3: Switch vs Router mode

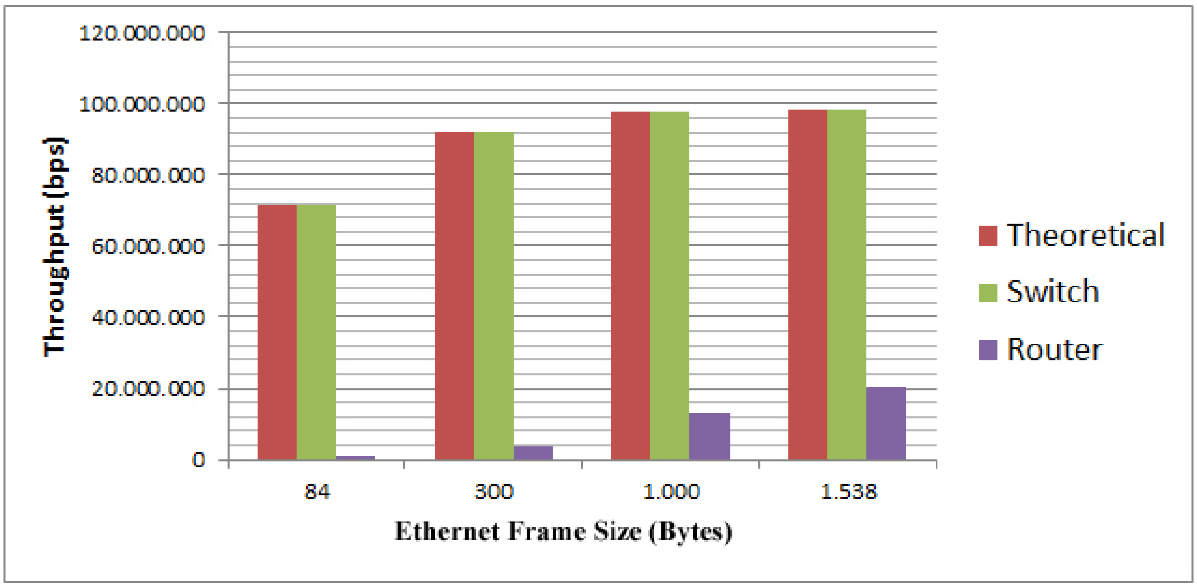


Figure 4.15: Experiment's 3 results. Throughput vs Ethernet Frame size

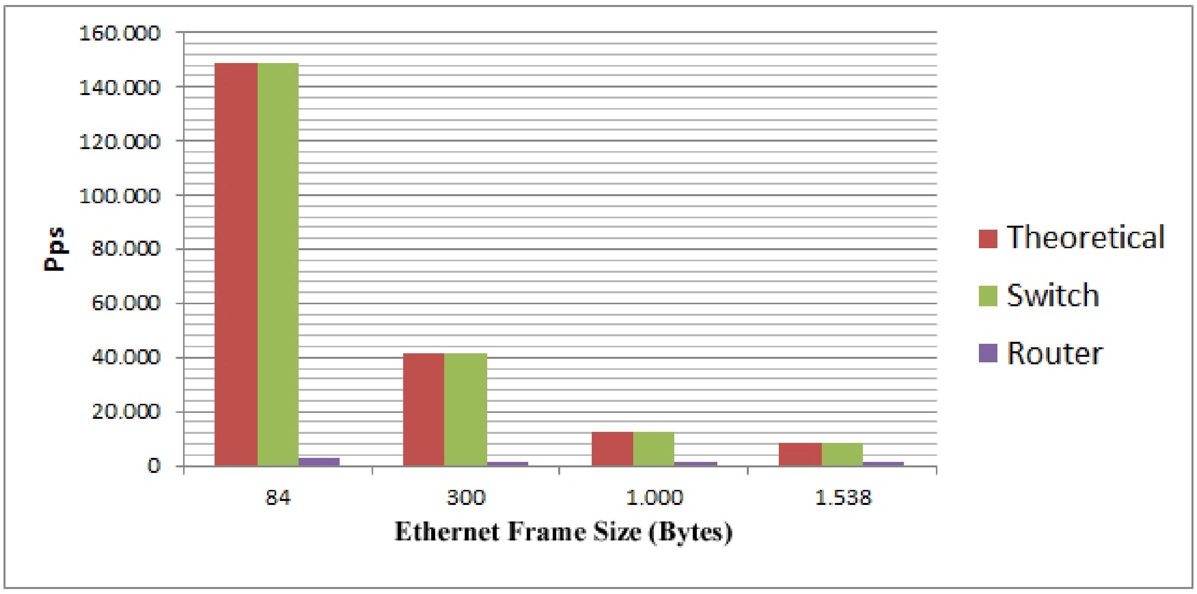


Figure 4.16: Experiment's 3 results. Pps vs Ethernet Frame size

Table 4.3: Experiment's 3 results. Jitter vs Ethernet Frame size

Ethernet Frame size (Bytes)	Switch (ns)	Router (ns)
84	72.00	4299.35
300	80.00	8112.00
1000	48.00	8498.35
1538	24.00	8988.42

This experiment shows the different performances of the Router WRT54GL when is set as router ,and as switch. In the case of the switch, the experiments revealed that the device can achieve the theoretical limits of throughput and pps (figures 4.15 and 4.16). The table 4.3 demonstrates that the jitter is low, between 24ns and 80ns. When comparing the results of the device as a router to the theoretical limits, we see that the throughput and the pps are pretty lower (from -79% to -98%)(figures 4.15 and 4.16). This is because the device checks more fields than when in switch mode. The TTL, the route table search, the checksum, the queuing and the forwarding of the packet demand a lot of process time. Because of these reasons the jitter rises to ms compared with the switch mode where it stays in the ns scale as the table 4.3 illustrates.

In conclusion, switches from experiments 2 and 3 work perfectly and can achieve the theoreticals limits because of the low process that the packets demand. On the other hand, the router that demands more process power has worst results. This means that average cost routers are not capable of achieving rates that more expensive and powerful routers can.

4.3.4 Experiment 4: Cisco Catalyst Router 7606

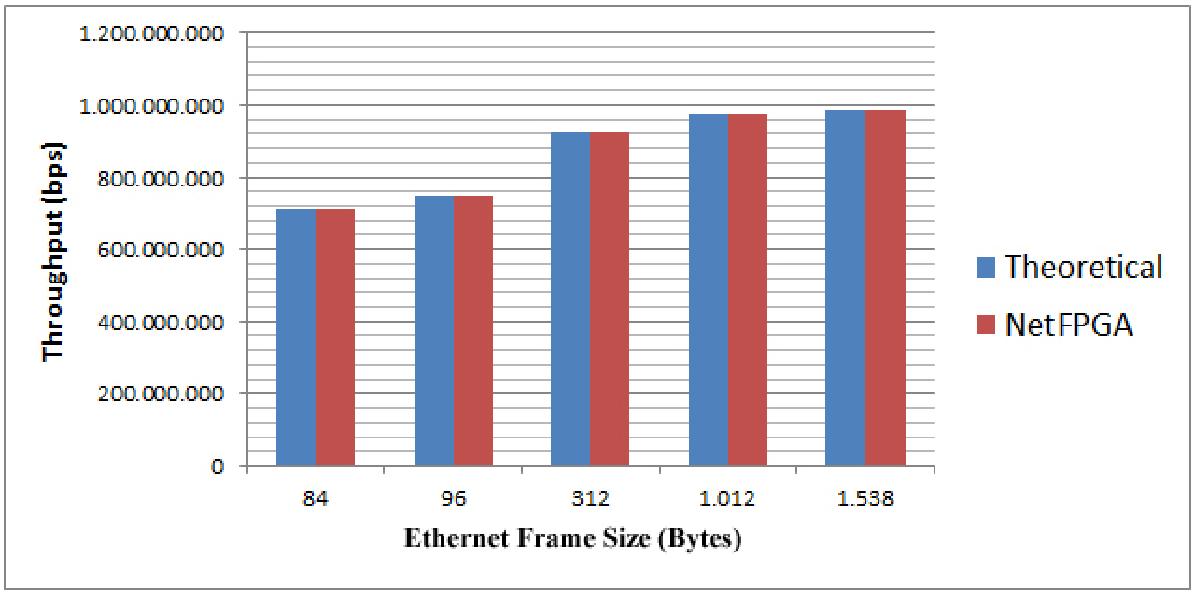


Figure 4.17: Experiment's 4 results. Throughput vs Ethernet Frame size

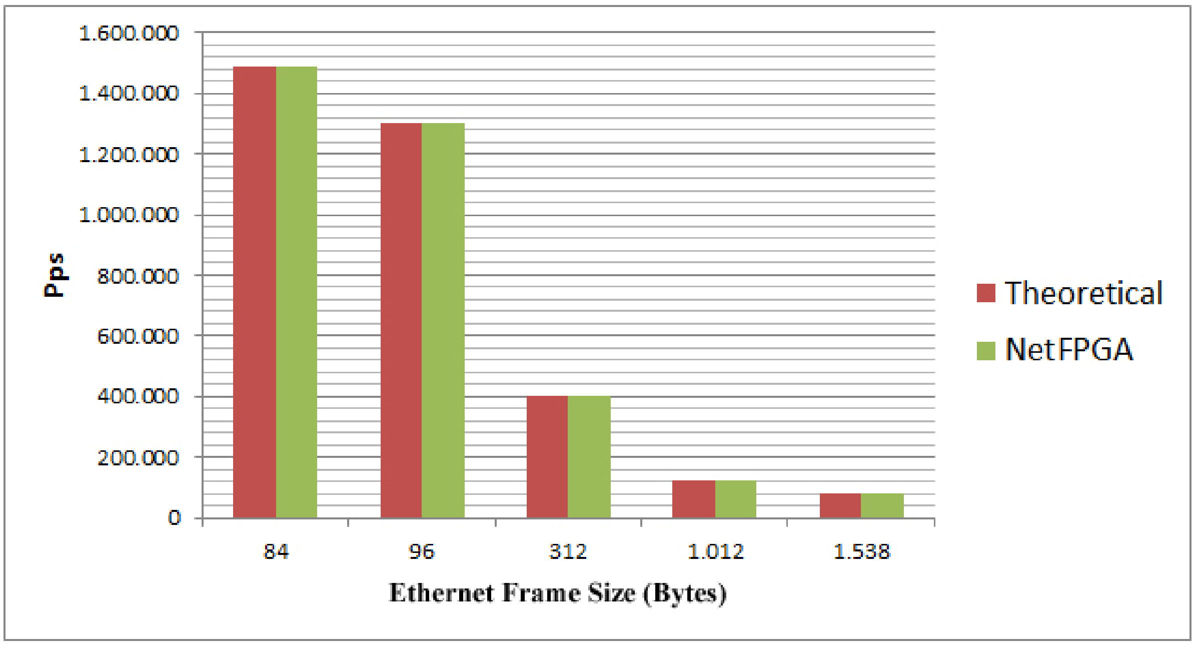


Figure 4.18: Experiment's 4 results. Pps vs Ethernet Frame size

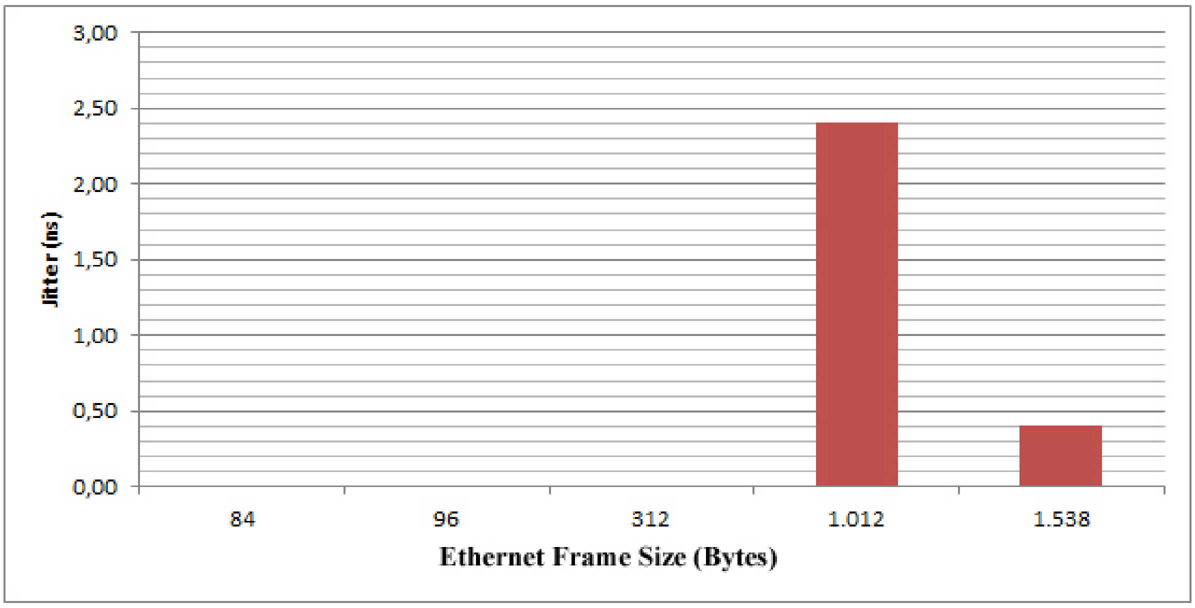


Figure 4.19: Experiment's 4 results. Jitter vs Ethernet Frame size

The Cisco Catalyst 7606 can manipulate flows of 1 Gbps without a problem. The results figures 4.17 and 4.18 show that the pps and the throughput of the router is really close to the theoretical calculations (-0.00243%). Also it is observed that the CPU load of the router didn't increase during the experiment. This means that a flow of 1Gbps is not enough to slow down the process of the packets or lose the packets altogether . The figure 4.19shows that the jitter for small packets is 0ns and for big packets is 2,4ns, which is minimum. Finally, the router is capable of routing packets at 1Gbps with high performance. Compared to the previous experiment, we may conclude that the more expensive the router is, the more capable and better results it yields.

4.3.5 Experiment 5: Wireless Access points

This experiment shows the performance of the Wireless Router WRT54GL at the available rates of the 802.11g. The results at the figures 4.20, 4.21, 4.22, 4.23, 4.24, 4.25, 4.26 and 4.27 with columns blue and red illustrate that at all rates the difference between the Theoretical throughput and the measurement throughput with the NetFPGA fluctuates. For small packet sizes the difference is higher than that of bigger packets. In table 4.4 is represented the percentile difference in the theoretical maximum throughput (TMT) and the NetFPGA throughput at the rate of 54Mbps.

Table 4.4: Percentile difference of TMT and NetFPGA throughput at 54Mbps

Ethernet Frame size (Bytes)	Percentile difference
84	-55.42%
96	-43.82%
312	-12.02%
1012	-9.47%
1538	-1.52%

The rest of the rates have similar patterns, wide difference of throughput for smaller packet size and narrow difference of throughput for bigger packet size. It is also observed that for smaller rates the difference in throughput of small packet size is smaller too. In table 4.5 is represented the percentile difference of TMT and NetFPGA throughput for packets of 84 byte size.

Table 4.5: Percentile difference of TMT and NetFPGA throughput of 84 byte packet.

Rate (Mbps)	Percentile difference
54	-55.42%
48	-48.51%
36	-46.63%
24	-28.53%
18	-27.74%
12	-22.21%
9	-22.91%
6	-23.00%

The figure 4.28 shows that the jitter for all the rates with small packet sizes is smaller (0,02ms) than that of bigger packet sizes (maximum 0,06ms). Another interesting point is that the jitter seems to be independent of the rate as at the 6Mbps rate we observed higher jitter than that of 54 Mbps and,similarly,at the 48Mbps we observed smaller jitter than that of 18Mbps packet sizes .

Lastly, the NetFPGA is able to measure the performance of the wireless connection with high precision. This wireless link can not achieve the TMT even though for big packet sizes the theoretical and the measured throughput are really close. This jitter is estimated at the scale of ms (0,02 - 0,06 ms). The same wireless link is measured with the iPerf and IxChariot software in experiment 6 in order to show the difference in precision of the measurements that the software packet generators offer with the NetFPGA hardware implementations.

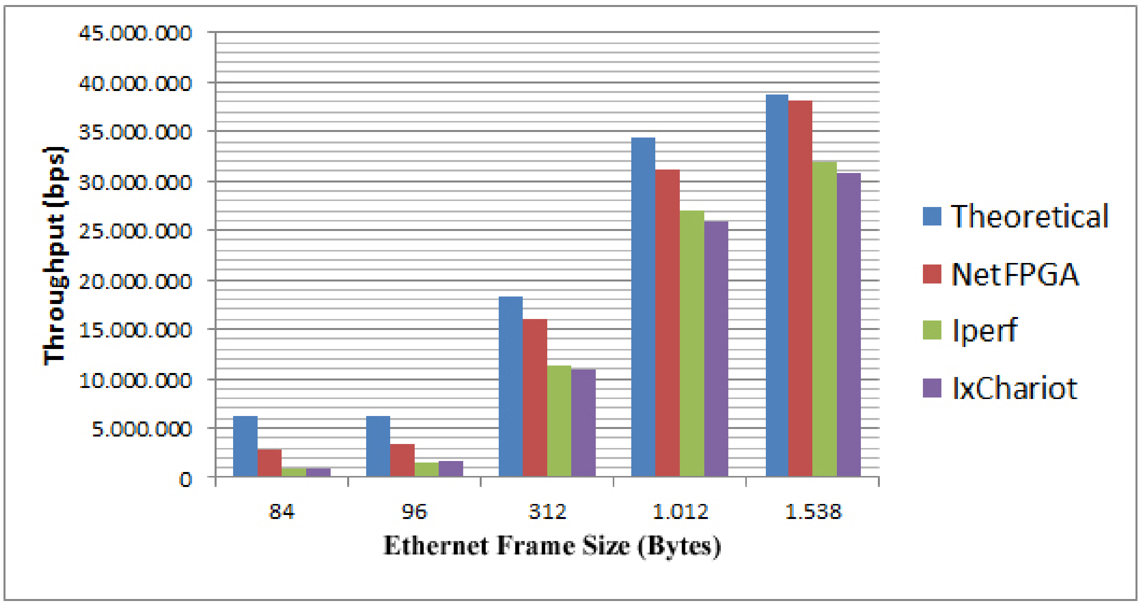


Figure 4.20: Experiments' 5 and 6 results: Throughput vs Ethernet Frame size at 54Mbps set

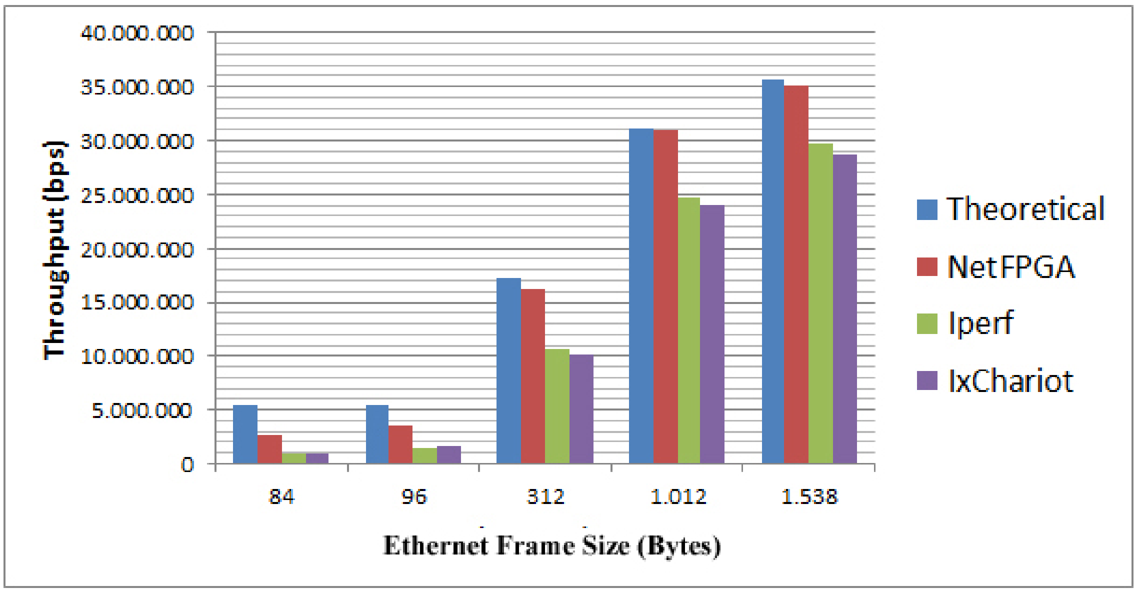


Figure 4.21: Experiments' 5 and 6 results: Throughput vs Ethernet Frame size at 48Mbps set

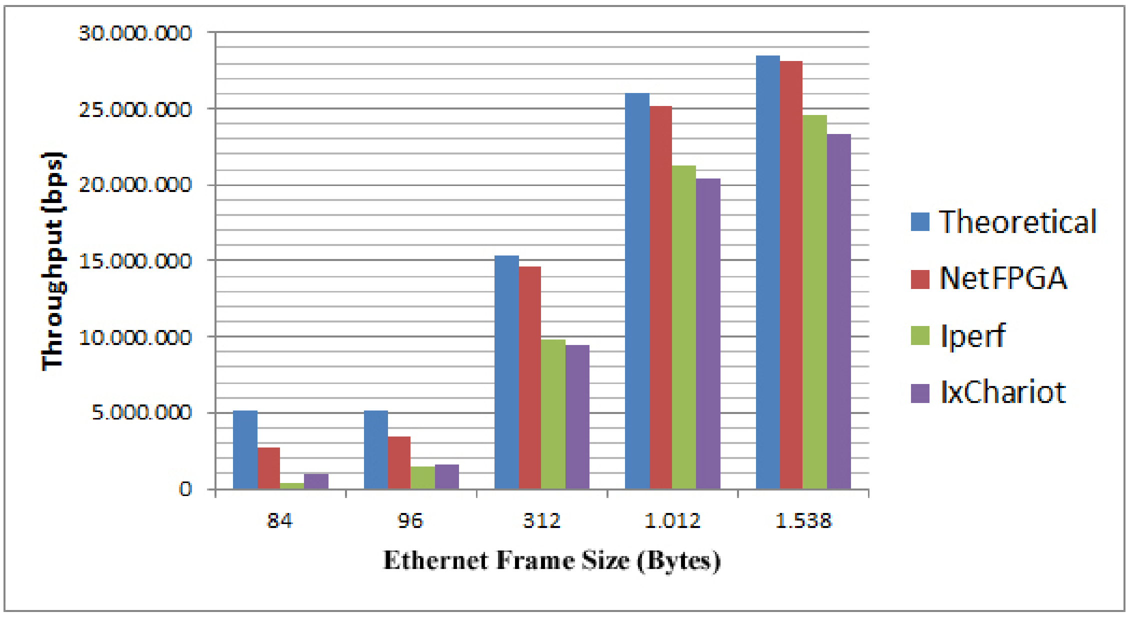


Figure 4.22: Experiments' 5 and 6 results: Throughput vs Ethernet Frame size at 36Mbps set

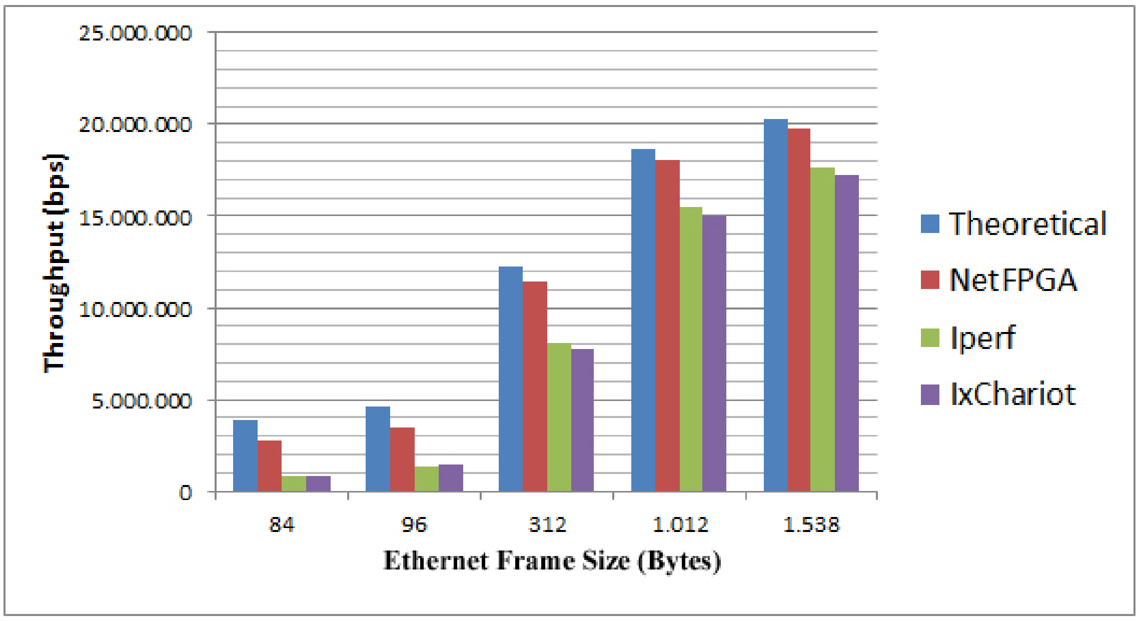


Figure 4.23: Experiments' 5 and 6 results: Throughput vs Ethernet Frame size at 24Mbps set

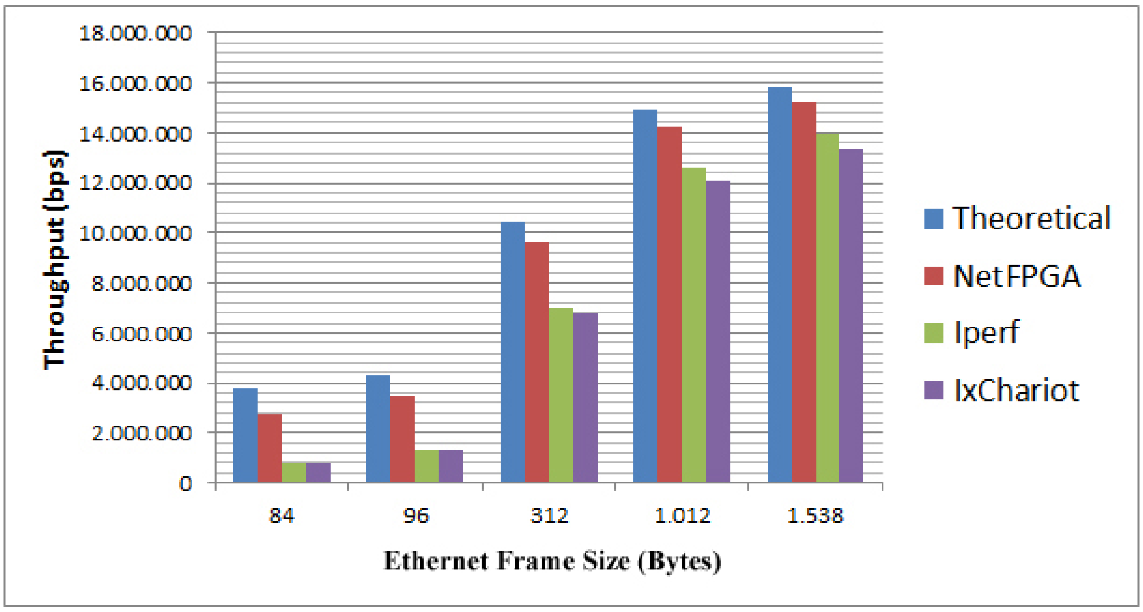


Figure 4.24: Experiments' 5 and 6 results: Throughput vs Ethernet Frame size at 18Mbps set

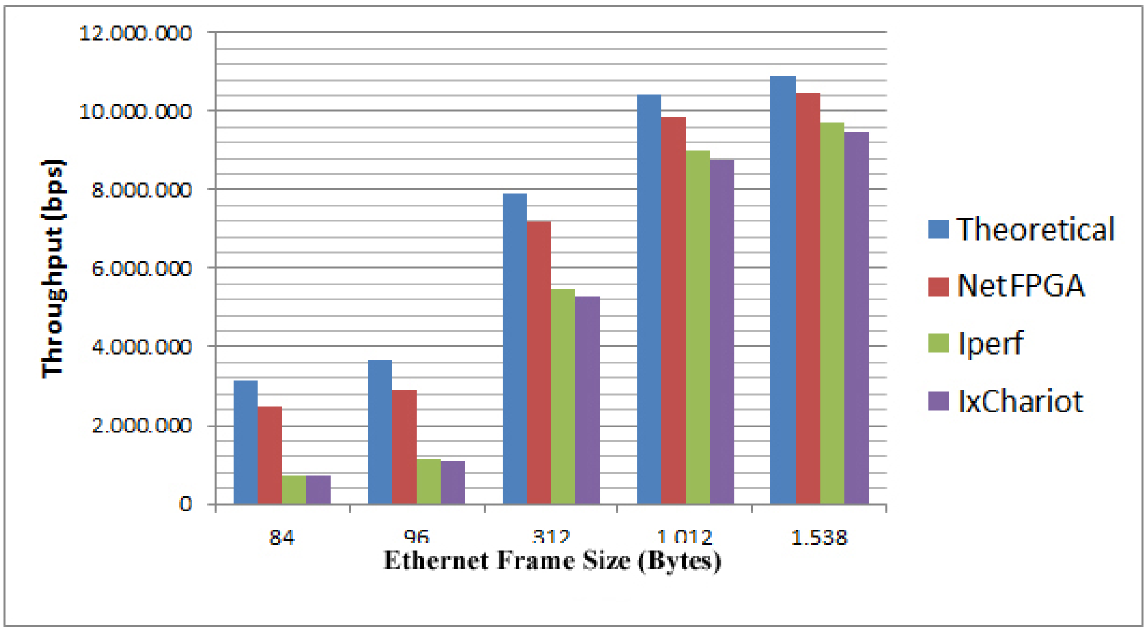


Figure 4.25: Experiments' 5 and 6 results: Throughput vs Ethernet Frame size at 12Mbps set

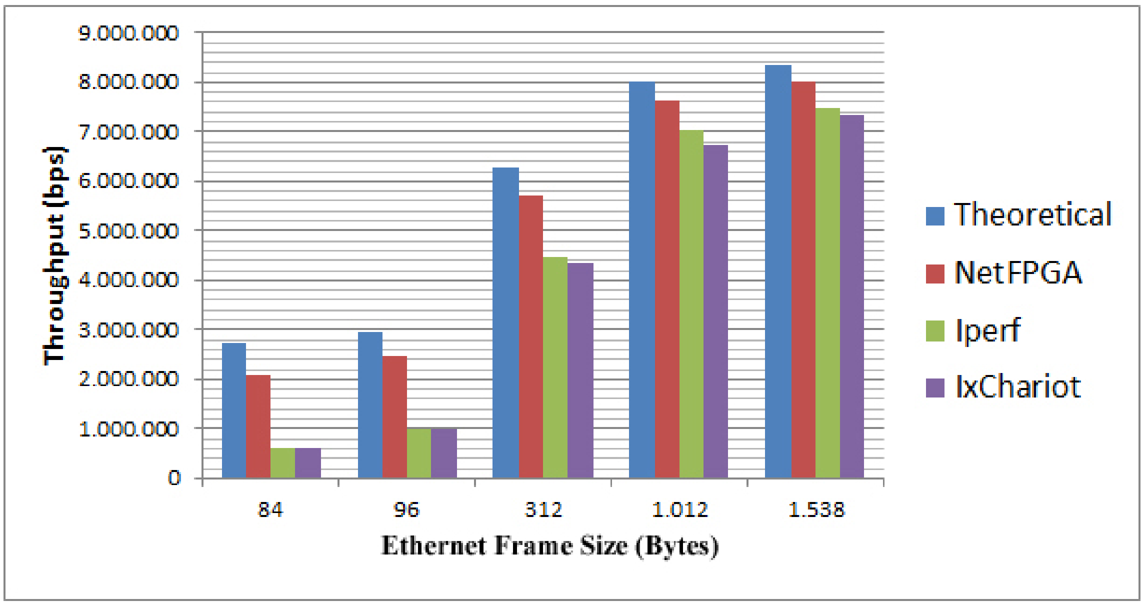


Figure 4.26: Experiments' 5 and 6 results: Throughput vs Ethernet Frame size at 9Mbps set

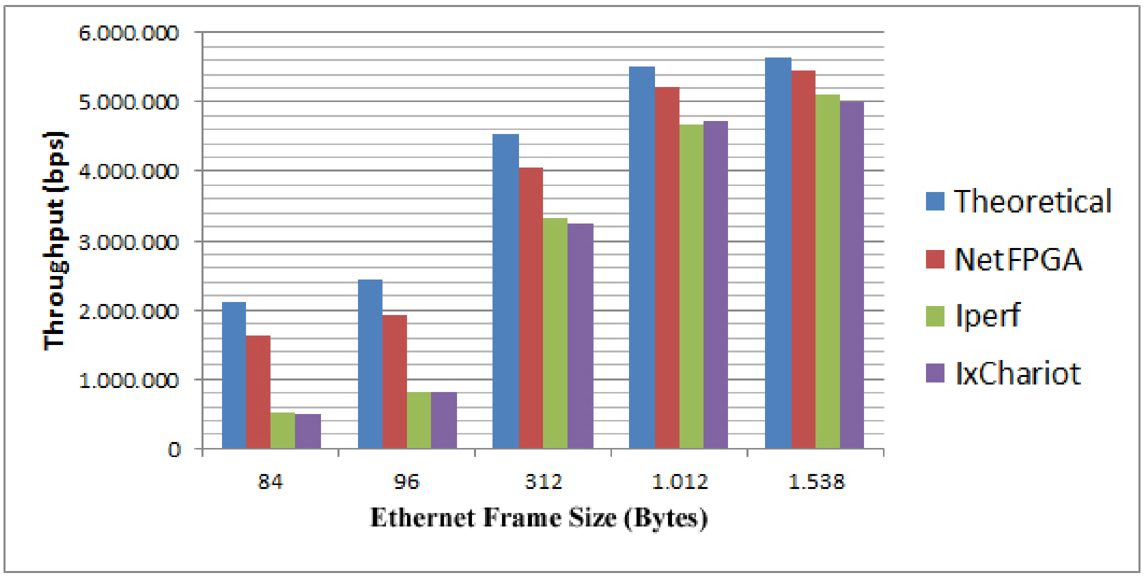


Figure 4.27: Experiments' 5 and 6 results: Throughput vs Ethernet Frame size at 6Mbps set

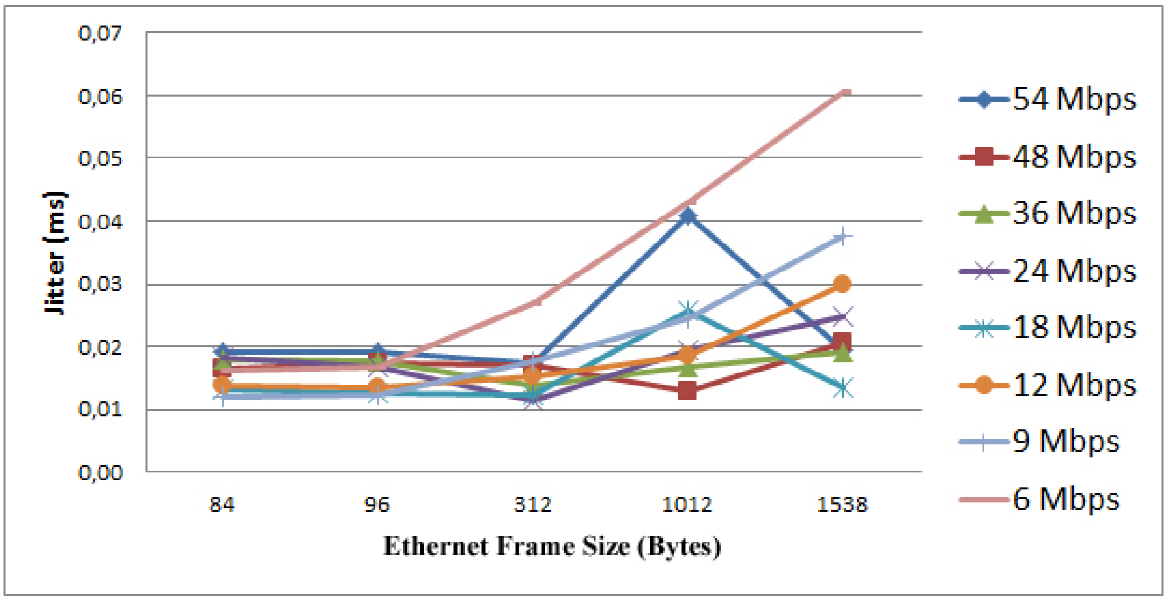


Figure 4.28: Experiment's 5 results: NetFPGA jitter vs Ethernet Frame size at the speeds of 802.11g

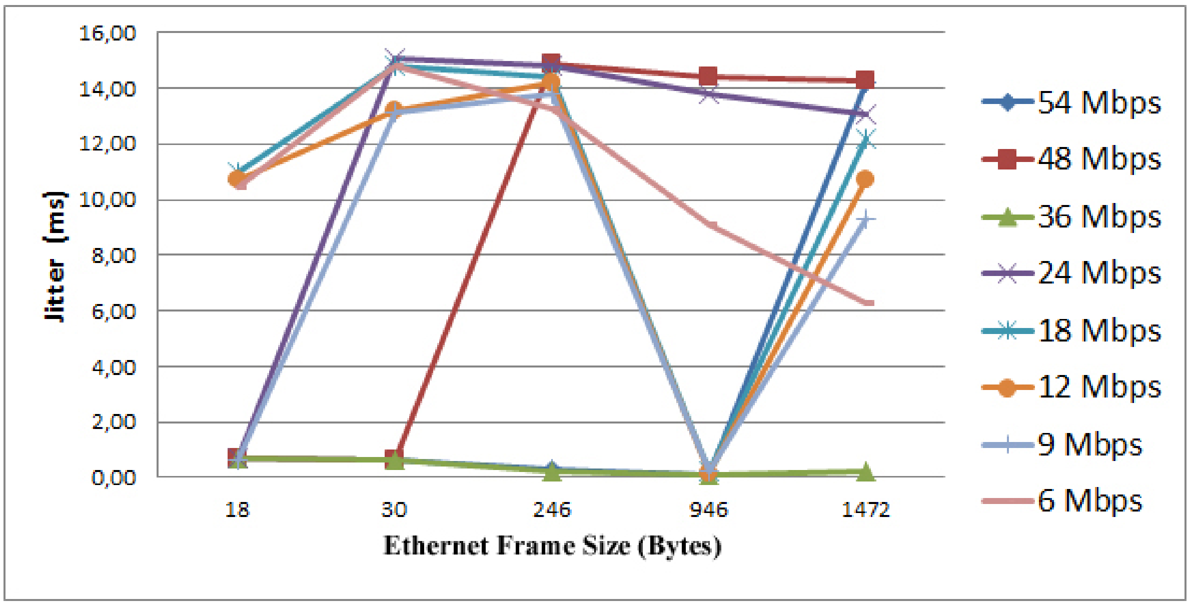


Figure 4.29: Experiment's 6 results: Iperf jitter vs Ethernet Frame size at the speeds of 802.11g

4.3.6 Experiment 6: IPERF vs IxChariot

The figures 4.20, 4.21, 4.22, 4.23, 4.24, 4.25, 4.26 and 4.27 with columns green and purple outlining the 802.11g throughput show that the two software have completely different performances. For Ethernet Frame sizes greater than 312 bytes the throughput of iPerf is greater (3.5%). For Ethernet Frame sizes less than 312 bytes (84 and 96 bytes) the throughput of IxChariot is sometimes greater and other times smaller than iPerf. It is observed that there are no fixed behavioral patterns. The software performance seems to be completely erratic. The results of experiment 6 are completely different than the results of experiment 5. For small payloads iPerf has an -84.64% difference of throughput than the NetFPGA. For bigger Ethernet Frame size the difference is smaller (-16.45% at 54Mbps).

The chart that depicts the CPU load of the client and server of the two softwares for 54Mbps reveals that CPU load is different for the two softwares. Iperf consumes less CPU than the IxChariot server for all the packet size. (-73%). For both iPerf and IxChariot the CPU load is independent and standard of the payload size.

On the other hand the CPU load of the client changes with the size of the packet. The smaller packet, the higher CPU load is needed. For small packet sizes IxChariot needs more CPU load (-8%) ,but for bigger packet sizes iPerf consumes more CPU load (from 186.27% to 82%) than IxChariot.

The figure 4.29 of the jitter of Iperf is at the scale of ms. The jitter doesn't follow a pattern of packet size or connection rate. The jitter varies from 1ms to 15 ms which is much higher than the NetFPGA jitter(ns scale). Because IxChariot doesn't give the information of the jitter at his measurements, it can not be commented.

Concluding, each software yields different results. Some tools yield better results for smaller packet sizes while others at bigger packet sizes, but both of them yield smaller results than the NetFPGA experiments. The calculations are not very precise as the jitter of the iPerf is high and behaves randomly. The CPU load depends on the implementation of the CPU and varies according to the packet size. The server CPU load is always lower than the client CPU load and this is because generation of the packets costs more than receiving and analysing them.

4.3.7 Experiment 7: Power consumption

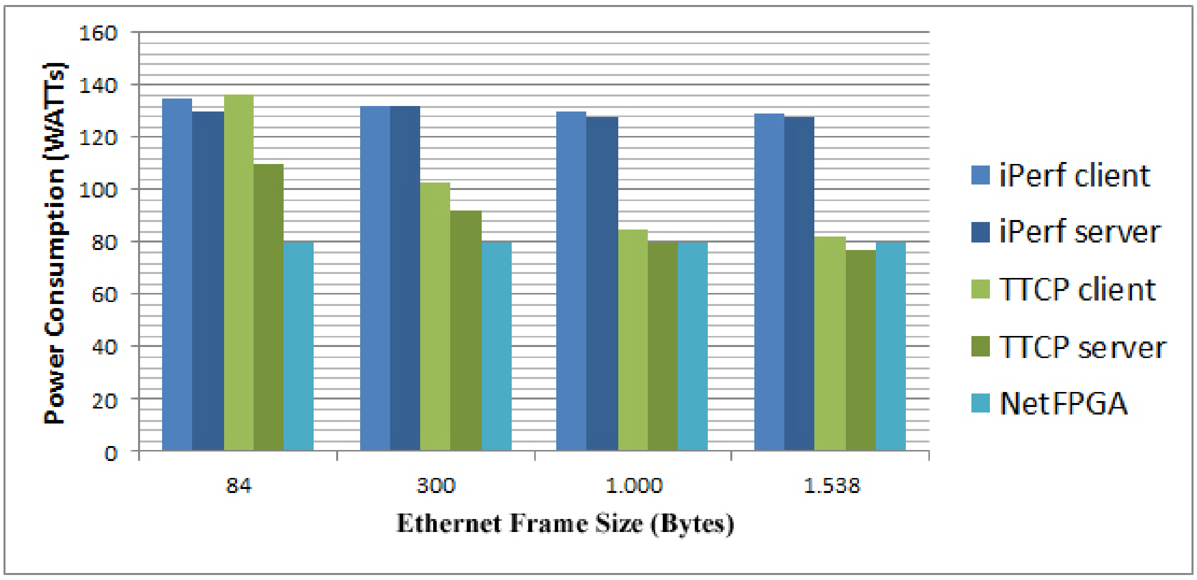


Figure 4.30: Experiment's 7 results. Power consumption vs Ethernet Frame size

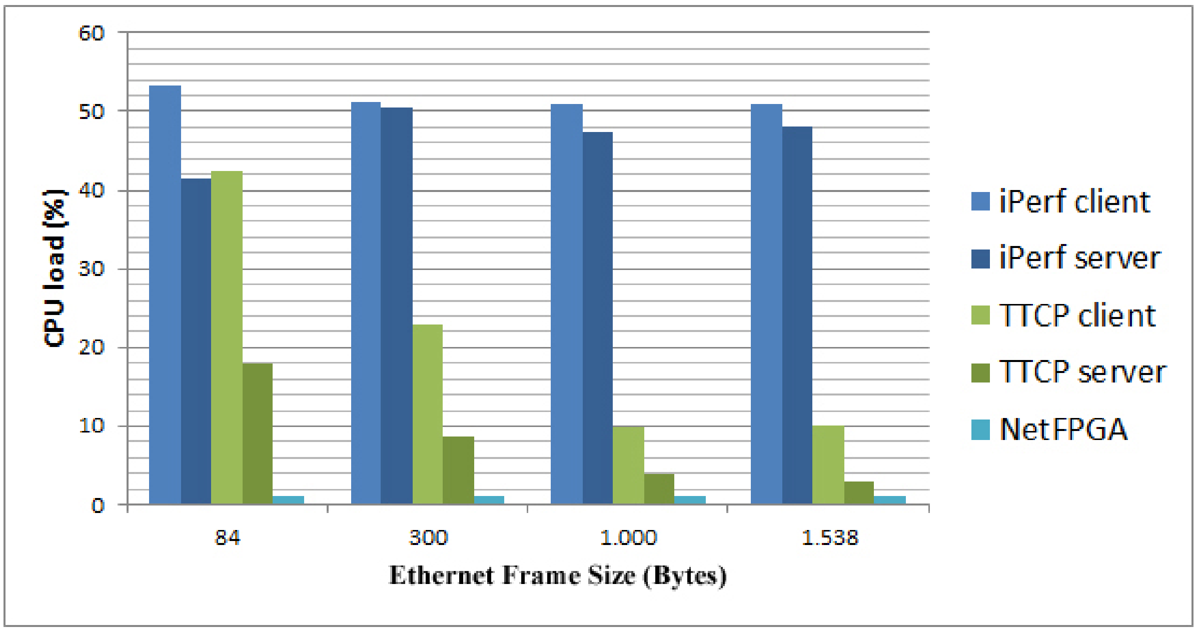


Figure 4.31: Experiment's 7 results. CPU vs Ethernet Frame size

For both software implementations,iPerf and TTCP, the computer that generates the flow (client) demands more processing and consequently more CPU load and more power consumption than when the computer receives and calculates the statistics (server) as the figures 4.31 and 4.30. Comparing the client CPU load of the two software implementations proves that the iPerf has dramatically more CPU load that the TTCP(the biggest difference is 40.84% for 1538 Ethernet Frame size). While the packet size increases, the client and the server CPU load reduces in the case of the TTCP, but not in the case of the iPerf where the CPU load remains the same. On the other hand, the NetFPGA CPU load is always 1% while the only process that is executed at the host computer is the driver that writes and reads the values from the card. This process does not require CPU load and ,as a result, does not require a powerful computer in order to generate precise statistical results.

The graph of the power consumption has the same shape as the graph of the CPU load. The more CPU is loaded, the more energy is consumed. The iPerf power consumption is between 129 and 136 Watts and for TTCP between 82 and 136 Watts. The power consumption of the NetFPGA is 80 Watts, which means that the program of the NetFPGA doesn't consume extra power on execution whereas when the computer is at idle state it consumes 80 Watts.

The power consumption of the iPerf is higher than the TTCP in all cases. The worst case of iPerf power consumption is noted at 51 Watts ;more than the ,TTCP which is double of the TTCP power consumption.

Concluding, the software implementations seem to depend a lot on the hardware that is executed. The high use of the CPU reveals that powerful computers should be used for more precise measurements. In addition, the software implementations consume exponentially more energy that the NetFPGA system.

Chapter 5

Conclusion

Packet injection into a network at high rate demands a lot a process power and high precision. Packet injection technique is used for network measurement and network devices under test. Traffic generators are implemented over both hardware and software platforms. Software platforms exist both as open-source and free-ware, and as closed source commercial products. Software platforms are more widely used because of their flexibility (easy deployability of multiple nodes, ability to rapidly modify and extend, possibility to perform more realistic experiments). Unfortunately, the experiments with different software on the same networks yields different results. This means that the software implementations are highly unreliable as they are dependent on the commercial off-the-shelf (COTS) hardware used, the operating system adopted, and the status of the host used for traffic generation. Moreover, hardware platforms are typically more precise and reach better performance as they are completely independent of the host computer features and they sometimes run without a computer. The hardware solution that incorporates FPGAs combines high computational capabilities with low energy consumption.

This thesis has focused on the implementation of a multi-gigabit high performance network measuring system based on FPGA. A UDP packet generator on NetFPGA 1G was successfully implemented and achieved four independence UDP flows with zero jitter at 1 Gbps. In addition, a high precision receptor is implemented for the statistics calculations. The ability our system to generate packets without the pre-capture of any traffic flows, makes it more flexible than other similar commercial implementations. The friendly user graphical interface, that is successfully implemented, gives the the same usability as other software products.

As the experiments showed, our implementation has higher network measurement results, lower generated packets jitter, the lowest power consumption and lowest CPU load than all other equivalent software programs that have been previously examined. Even in low measurement rates the software implementations' results are varied that makes them unreliable. As a result, in cases that high precision network measurements is needed our system is the best affordable solution.

Bibliography

[1] "Iperf," http://code.google.com/p/iperf/, 2014.

[2] "Ixchariot," http://www.ixiacom.com/products/ixchariot/, 2014.

[3] C. Dovrolis, P. Ramanathan, and D. Moore, "What do packet dispersion techniques measure?" in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, 2001, pp. 905–914 vol.2.

[4] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis, "An architecture for large-scale internet measurement," *IEEE Communications*, vol. 36, pp. 48–54, 1998.

[5] S. Savage, "Sting: A tcp-based network measurement tool," in *Proceedings of the 2Nd Conference on USENIX Symposium on Internet Technologies and Systems - Volume 2*, ser. USITS'99. Berkeley, CA, USA: USENIX Association, 1999, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251480.1251487

[6] C. Demichelis and P. Chimento, "IP Packet Delay Variation Metric for IP Performance Metrics (IPPM)," RFC 3393 (Proposed Standard), Internet Engineering Task Force, Nov. 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3393.txt

[7] "Netfpga," http://www.netfpga.org/, 2014.

[8] G. A. Covington, G. Gibb, J. W. Lockwood, and N. McKeown, "A packet generator on the netfpga platform." in *FCCM*, K. L. Pocek and D. A. Buell, Eds. IEEE Computer Society, 2009, pp. 235–238. [Online]. Available: http://dblp.uni-trier.de/db/conf/fccm/fccm2009.html#CovingtonGLM09

[9] "Packet generator netfpga," https://github.com/NetFPGA/netfpga/wiki/PacketGenerator, 2013, wiki.

[10] "Reference router walkthrough," https://github.com/NetFPGA/netfpga/wiki/ReferenceRouterW
2013, wiki.

[11] D. Plummer, "Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet

Hardware," RFC 826 (INTERNET STANDARD), Internet Engineering Task Force, Nov. 1982, updated by RFCs 5227, 5494. [Online]. Available: http://www.ietf.org/rfc/rfc826.txt

[12] J. Morriss, "Gratuitous arp," 2009. [Online]. Available: http://wiki.wireshark.org/Gratuitous_ARP

[13] E. A. Hall, *Internet core protocols - the definitive guide: an owner's manual for the internet.* O'Reilly, 2000.

[14] J. Postel, "Internet Protocol," RFC 791 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1349, 2474, 6864. [Online]. Available: http://www.ietf.org/rfc/rfc791.txt

[15] R. Braden, "Requirements for Internet Hosts - Communication Layers," RFC 1122 (INTERNET STANDARD), Internet Engineering Task Force, Oct. 1989, updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633, 6864. [Online]. Available: http://www.ietf.org/rfc/rfc1122.txt

[16] C. Perkins, in *RTP: Audio and Video for the Internet*, vol. 2, 2012, p. 414.

[17] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," RFC 3550 (INTERNET STANDARD), Internet Engineering Task Force, Jul. 2003, updated by RFCs 5506, 5761, 6051, 6222, 7022, 7164. [Online]. Available: http://www.ietf.org/rfc/rfc3550.txt

[18] "Ieee standard 802.1q-1998, "ieee standards for local and metropolitan area networks: virtual bridged local area networks" (institute of electrical and electronics engineers, new york, 1999)." [Online]. Available: http://www.ieee.org

[19] "Ieee standard for ethernet - section 1," *IEEE Std 802.3-2012 (Revision to IEEE Std 802.3-2008)*, pp. 1–0, Dec 2012.

[20] Y. Xiao and J. Rosdahl, "Throughput and delay limits of ieee 802.11." *IEEE Communications Letters*, vol. 6, no. 8, pp. 355–357, 2002. [Online]. Available: http://dblp.uni-trier.de/db/journals/icl/icl6.html#XiaoR02

[21] R. McIlroy, "Technologies of ethernet speed 100mbps / 1gbps."

[22] "Forth," 2014. [Online]. Available: http://www.forth.gr