Design and implementation of an UDP/IP Ethernet hardware protocol stack for FPGA based Systems

A Master's Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Gerard Guixé Orriols

In partial fulfilment

of the requirements for the degree of

MASTER IN ELECTRONIC ENGINEERING

Advisor: Francesc Moll Echeto

Barcelona, January 2019

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

telecom BCN

Title of the thesis: Design and implementation of an UDP/IP Ethernet hardware protocol stack for FPGA based Systems

Author: Gerard Guixé Orriols

Advisor: Francesc Moll Echeto

Abstract

The main objective of the thesis has been the design and implementation of a complete UDP/IP Ethernet stack that allow us the connection and use of networks by any FPGA device. The stack has been designed around Ethernet, IPv4 and UDP protocols as it was wanted a fast and scalable way of distant communication. Other protocols have been added as a complement in order to improve its operation like ARP and DHCP.

The project has focused around the implementation of this stack as a generic IP core, but it has been extended further on with the implementation of an initial data acquisition interface (DAQ) that would allow us to transmit the information of its channels to the network. At the end, the project has been successfully implemented in a real FPGA system. And all the tests have been passed with minimum packet loss, from simple operational test to more final ones like the test of a DAQ service interface.

Acknowledgements

I want to give thanks to both my directors Joan Mauricio Ferré and Francesc Moll Echeto, for his support during the project and all the advice given through the work. Also, thanks to all the people in the ICCUB for giving me the tools and help that was needed to develop this project and for the plentiful coffee breaks.

Thank you too, to all my friends Ana, Anna, Carlos, David, Didac, Guillem, Hasari, Hochi, Jordi, Judit, Nerea, Patricia, Quique, Sofia, Thais and Tom for being there and make of me a less productive person when it was really necessary.

And of course, many thanks to my parents and family for their support and understanding, and for rising me with a love for technology, and lots of 80s sci-fi.

Revision history and approval record

Revision	Date	Purpose
0	30/10/2018	Document creation
1	14/12/2018	Document revision
2	27/12/2018	Document revision

Written by:		Reviewed and approved by:	
Date	30/10/2018	Date	27/12/2018
Name	Gerard Guixé	Name	Francesc Moll
Position	Project Author	Position	Project Supervisor

Table of contents

List of Figures

List of Tables

1. Introduction

This project has been made in collaboration with the Technological Unit of the Institute of Cosmos Sciences (ICCUB). The Unit works in a high number of electronic related projects, designing ICs and PCBs, and making full systems with very specific functions. Most of the systems have an FPGA integrated.

The project initially arose from a very specific need. The Institute works with various systems which require very fast data acquisition devices (DAQ). The systems typically consist in one or more Front-End ASICs which amplify, shape and digitize signals coming from photosensors. Then, the FPGA acquires the digital data and then sent it via USB to an external computer.

The disadvantage of this design is that there is the constant need of a computer physically connected to the board in order to gather data and if it was needed some kind of configuration to the board it can only be made using that computer. The use of USB has strict limitations in distance and scalability. There was also a need to communicate the boards to a network without adding a lot of extra components to the already existent PCBs.

Then, it was proposed the design of a communication stack that would allow the PCB designs to connect, send and receive messages through various types of networks. This design would be implemented in an FPGA, as it is an integral part of the boards. The stack would allow to communicate to any computer connected to the network, and would allow at least some kind of communication via Internet. As we were working with DAQ sending data at high speed, there was a need of a fast communication, prioritizing speed over reliability. That is when it finally showed up the final idea for this Thesis, the creation of an UDP/IP Ethernet protocol stack IP core, that would be implemented in an FPGA (Cyclone 10 LP) using some hardware description language.

Some other specifications were created after that, for example it was important that the system was easy to set, and easy to move around. For this reason, some other protocols were set as main objectives like ARP or DHCP. The project focuses specifically on the stack and a few initial services, but it has been left open the possibility of creating a greater number of different applications that would increase the uses and complexity of the project.

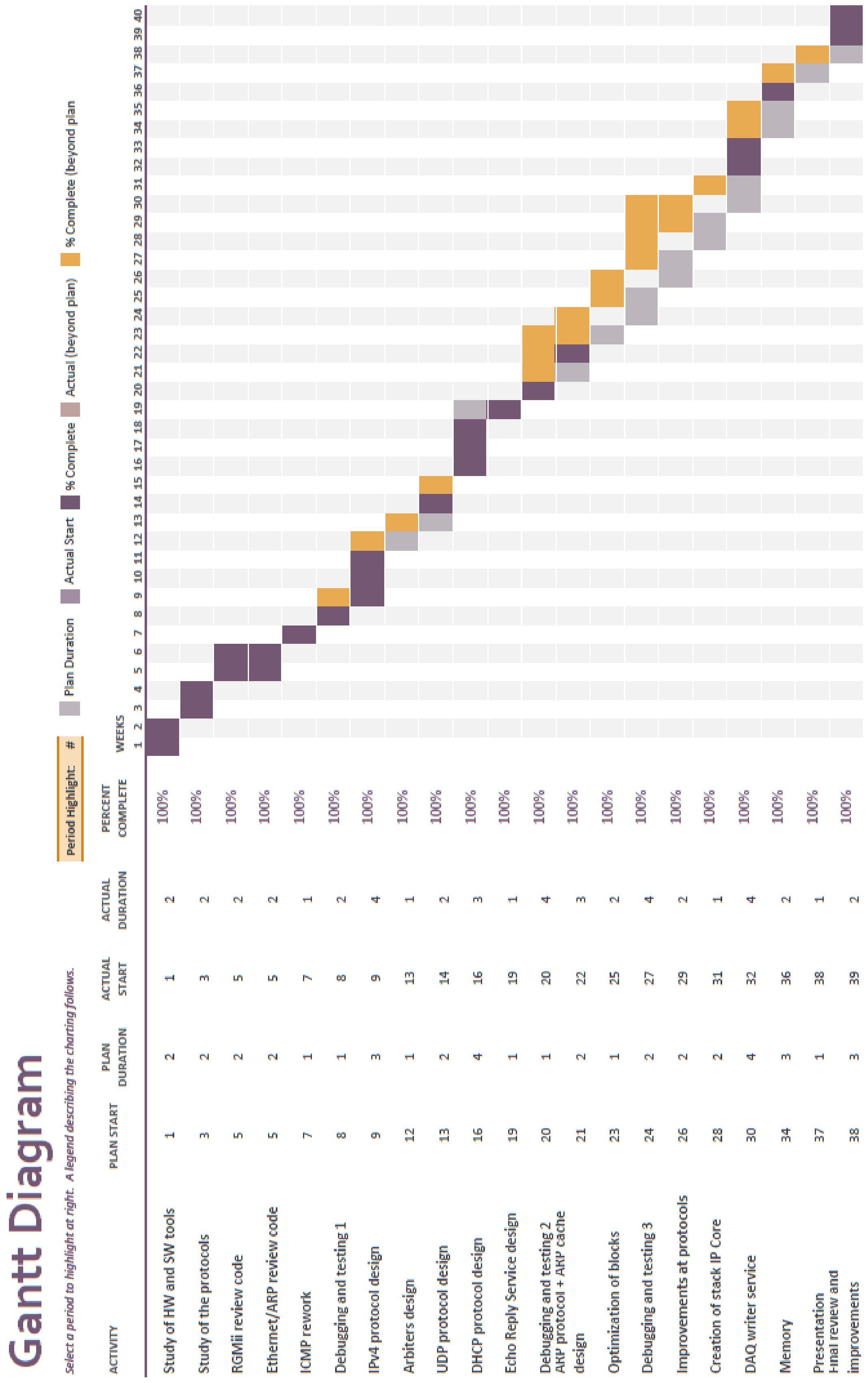
1.1. Objectives

The main objectives defined for the project are the following ones:

- Design of an UDP/IP Ethernet hardware protocol stack.
- Implementation of the design in an FPGA system with VHDL or Verilog.
- Incorporation of a dynamic host configuration protocol (DHCP) for highest adaptability in all type of networks.
- Creation of client-side applications to manage systems and data related to the FPGA via Ethernet.

1.2. Work plan

The thesis was planned as a 9-month project working between 4 and 6 hours each working day. This is the Gantt diagram of the project:



ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Study of HW and SW tools	1	2	1	2	100%
Study of the protocols	3	2	3	2	100%
RGMii review code	5	2	5	2	100%
Ethernet/ARP review code	5	2	5	2	100%
ICMP rework	7	1	7	1	100%
Debugging and testing 1	8	1	8	2	100%
IPv4 protocol design	9	3	9	4	100%
Arbiters design	12	1	13	1	100%
UDP protocol design	13	2	14	2	100%
DHCP protocol design	16	4	16	3	100%
Echo Reply Service design	19	1	19	1	100%
Debugging and testing 2	20	1	20	4	100%
ARP protocol + ARP cache design	21	2	22	3	100%
Optimization of blocks	23	1	25	2	100%
Debugging and testing 3	24	2	27	4	100%
Improvements at protocols	26	2	29	2	100%
Creation of stack IP Core	28	2	31	1	100%
DAQ writer service	30	4	32	4	100%
Memory	34	3	36	2	100%
Presentation	37	1	38	1	100%
Final review and improvements	38	3	39	2	100%

Figure 1. Project's Gantt Diagram

As we can appreciate taking a look at the Gantt, the Project was delayed from the planned dates, and the delay was kept for all the duration of the project from week 21.

The most delayed activities were the debugging and testing sections, they were planned for relatively short times like 1 or 2 weeks, but at the end they all doubled their durations. These delays appear due to a lot of unexpected problems that originate during the development of the stack, some of them were unexpected at a new challenge, so they took longer to solve.

One of the most remarkable problems was the appearance of timing violations (a negative slack in the main clock). This timing violations were a huge problem, as they induced an unpredictable behavior to the stack. In order to solve these violations, some signals were pipelined, this is further explained in Section *3.10 Timing Violations*. As the violations appeared very late on the project, the latch of signals desynchronizes most of the protocols, and it made necessary a rework of timings and counters in most of the protocols.

2. State of the art of the Internet Protocol

The project focuses on the design of a protocol stack as digital hardware in an FPGA. This means the implementation of a protocol suite that will fit our communication needs with other devices in the network.

Studying our own needs, it was decided to implement the Internet Protocol suite. This suite has a large number of protocols that can work over all kind of networks and can communicate with all kind of devices, the use of the Internet Protocol suite, makes the communication with personal computers very easy and that was one of the main objectives.

In this section we will analyze the Internet Protocol suite as a whole, and later on we will make a review of some of the individual protocols that are part of this suite. We will make a special focus on the protocols that have been implemented in our design, and in the next sections we will talk about the justifications of these choices.

2.1. The Internet protocol suite

The Internet protocol suite was created as a model that group the communication protocols used in the Internet Network and in other similar network systems [1]. The main protocols used in Internet are the Internet Protocol (IP) and Transmission Control Protocol (TCP), for this reason this suite is also sometimes called TCP/IP model.

The model has four layers of organization; they are used to classify the protocols for its functionality inside the network. The four layers are:

- **Application Layer:** Services that create user data and transmit it to other services through a network. Process to process communication.
- **Transport Layer:** Host to host communication. Reliability and flow control, multiplexing, can provide control-oriented communications. Main protocols are UDP and TCP.
- **Internet Layer:** Transport of packets across the network. Address and route of structures. Main protocol here is IP.
- **Link Layer:** Protocols that operate at the physical link level.

The model hides the network topology behind the protocols. Two host can communicate transmitting and receiving messages as if they were directly connected as seen in the network topology of Figure 2. Network topology and The host use the routers as a direct pipeline, all the complex processes are happening in the lower layers as seen in the Data Flow of the same figure. That flow is ignored by the host applications.

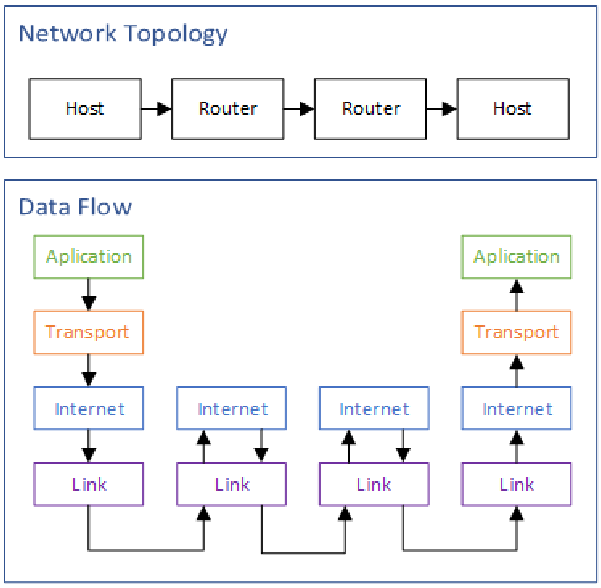


Figure 2. Network topology and data flow of the internet protocol [1]

Internet suite packets use encapsulation in each layer [1]. The data from the highest layer is encapsulated in the next one, this one at the same time is encapsulated in its next, this happens in each level of the protocol. This encapsulation process can be easily understood looking at Figure 3. Data encapsulation of the internet protocol model Data encapsulation of the internet protocol model provides abstraction to the protocols, the highest protocol sends the data down to the other layers, and it does not interact with them in any way, the data starts encapsulating until we reach the final layer and the data is transmitted. This simplify the communication and the interactions between layers.

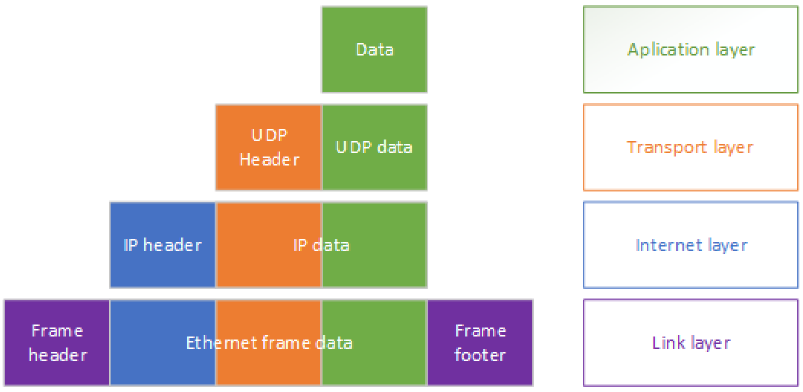


Figure 3. Data encapsulation of the internet protocol model [1]

2.1.1. The OSI model

The OSI (Open System Interconnections) model serves as way of standardization of all the telecommunication protocols without the need of knowing the technology used or the hardware structure [2].

The final goal of the model is to simplify the use and understanding of different communication systems and standard protocols. The model splits theses system and protocols in a set of specific layers. The standard model has seven layers: the physical layer, the data link layer, the network layer, the transport layer, the session layer, the presentation layer and finally the application layer. These layers describe the different steps that a bit of data has to cover in order to be correctly manage, transmit and receive.

During the project, the model has been used to classify the protocols used in the stack and to help in explanations and diagrams. The internet protocol model was created before the OSI model. Both models are constantly compared, this leads to confusion, because both models have different objectives, and assume different things.

If we compare the OSI model with the Internet protocol suite we see that both models have layers with similarities: The TCP/IP model Application layer includes the three top layers of the OSI model (Session, Presentation and Application), the transport layer is the same in both models, the Internet layer is the same as the OSI model Network layer, finally the TCP/IP Network access layer includes both the physical and data link layers of the OSI model.

With all this information we can define the following table, we have added the protocols used for the project in the correspondent layer:

Table I. OSI Model and IP model table of equivalencies [1] [2]

OSI model	**IP model**	**Layer number**	**PDU**	**Protocols**
Application layer	Application layer	L7	Data	DHCP, DAQ, Echo
Presentation layer		L6		(None)
Session layer		L5		(None)
Transport layer	Transport layer	L4	Datagram (UDP)	UDP
Network layer	Internet layer	L3	Packet	IPv4, ICMP
Data link layer	Network access layer	L2	Frame	*ARP*, MAC (Ethernet)
Physical layer		L1	Bit	IEEE 802.3u (RGMII)

The layer number is used as an abbreviation for layer names, the PDU represents the protocol data unit used in each layer. As we can see both models have equivalencies between layers.

The protocols and systems selected for the project can usually be identified to a layer but some of them are and exception as they work in more than one layer. We call them cross-layer protocols and are represented in cursive on the table, they are considered cross-layer for the following reasons:

- **ARP protocol:** It maps the IP addresses (L3) to MAC addresses (L2). The Ethernet protocol needs to know what IP address is connected to each MAC in order to being able to correctly send a message.
- **DHCP protocol:** It gives an IPv4 address (L3) to every new system and device that joins the network using UDP packets (L4). The Ethernet protocol cannot obtain an address in its own, we are in need of the DHCP.

From now on, we are going to talk about all the protocols involved in the development of our stack. The protocols have been classified following the OSI model layering. The only exception is the Application layer, due to its simplicity the three last layers of the OSI model have been represented together in a single Application layer (like in the TCP/IP model).

2.2. Physical layer

The physical layer (abbreviated as PHY) refers to the circuitry required in the system to connect a link layer device (abbreviated as MAC for medium access control) to a physical medium. The physical medium can be a copper cable, optical fiber or other types of medium like RF signals.

For the design of the UDP/IP stack, we have used a media-independent interface (MII) to connect with our physical medium, an Ethernet cable. We are going to talk about the MII interfaces and some of its variants, making a special focus in the reduced media-independent interface (RGMII) used in the Project.

2.2.1. Media-independent interface (MII)

The media-independent interface (MII) is a standard that was defined as a way to connect any Fast Ethernet MAC block to any Physical chip. This standard allows us to connect different types of physical devices to different types of media (twister pair, optical fiber, etc.). These connections can be made with a single chip, without the need of redesigning or replacing the hardware in the MAC side [3].

The original MII protocol transfers network data with 4 transmit data bits, 4 receive data bits and a clock of 25 MHz, and it achieves speeds of 100Mbit/s. Some variants have been developed over the years to be used with higher speeds and a lower number of signals. These new variants have been necessary to match with modern technologies that are faster and more efficient. Some of the current variants with a brief description of each one are:

- **Reduced media-independent interface (RMII):** Reduced the number of signals that are required to interface the physical layer to the MAC.
- **Gigabit media-independent interface (GMII):** This interface allows speed of 10, 100 and 1000 Mbit/s. It uses two clocks, one for 10/100 and the other for 1000 Mbit/s speeds, 8 transmit data bits and 8 receive data bits.
- **Reduced gigabit media-independent interface (RGMII):** A combination of the last two, it uses half the data signals than a GMII interface requires and it can achieve speeds of 10/100/1000 Mbit/s. It uses DDR to achieve this purpose [4].

The UDP/IPv4 Ethernet stack has being designed for a 10/100/1000 Base-T Ethernet port with a specific development board that contains an RGMII chip. This chip is the one in charge of interfacing with the FPGA stack protocols. Therefore, we are going to focus the explanation only on this type of interface.

2.2.2. Reduced gigabit media-independent interface (RGMII)

The RGMII is designed for 10/100/1000 Ethernet speeds with a reduced number of pins. It creates a transmit path from the media side to the physical side, and a receive path from the physical side to the media side. These are the supported speeds:

Table II. RGMII Ethernet supported speeds [4]

RGMII supported Ethernet speeds		
[Mbits/s]	**[MHz]**	**Bits/clock cycle**
10	2.5	4
100	25	4
1000	125	8

RGMII uses half the number of data pins than older standards like GMII. The reduction of data pins is achieved by using both the rising and falling edge of the clock in a process called DDR (for 1000 Mbit/s operation) and by eliminating other signals that are not essential for a working interface. The RGMII has then 12 pins:

Table III. Minimum input and output ports needed in a RGMII for a working chip [4] [5]

Signal	**I/O Type**	**Description**
TXC	Output	Transmit clock from an FPGA
TXD[3..0]	Output	Data to transmit. Bits 3:0 on the positive edge of TX_CLK and bits 7:4 on the negative edge of TX_CLK.
TX_CTL	Output	Multiplexing of transmitter enable and transmitter error signals. TXEN on the positive edge of TX_CLK and TXEN xor TXERR on the negative edge of TX_CLK.
RXC	Input	Receive reference clock from the external PHY
RXD[3..0]	Input	Data to receive. Bits 3:0 on the positive edge of RX_CLK and bits 7:4 on the negative edge of RX_CLK.
RX_CTL	Input	Multiplexing data valid and receiver error. RXDV on the positive edge of RX_CLK and RXDV xor RXERR on the negative edge of RXC.

The transmission clock signal is provided by the MAC on the TXC line. The RGMII standards say that data and clock have to be output at the same time (source-synchronous clocking) without any skew on the clock. For a correct sampling at the receiving side some delay has to be added to the clock (1.5-2ns), this can be done in the PCB design or at the receiver device as an internal delay.

Data is clocked in both clock edges for 1000Mbit/s but only in the rising edge for 10 and 100 Mbit/s. There is an exception to that: the signal RX_CTL is in charge of transmitting RXDV (data valid) on the rising edge and RXDV xor RXER (receiver error) on the falling edge. In the other hand, TX_CTL transmits TXEN (transmitter enable) on the rising edge and TXEN xor TXER (transmitter error) on the falling edge. This happens in the case of all three speeds. We will deepen into the operation of the RGMII in the design section.

2.3. Data link layer

This layer is in charge of delivering frames between different nodes on the same level of the network. The data link layer has two sublayers, the logical link control (LLC) and the media access control (MAC).

- **Logical link control (LLC):** The LLC is used to multiplex the protocols of the upper layer. It can perform more actions like flow control, error notification, acknowledgement, etc. It is defined in IEEE 802.2 [8].
- **Medium access control (MAC):** The medium access control sublayer (also known as media access control sublayer) also provides multiplexing and flow control, but in this case of the physical transmission medium. As it has already been explained, the MAC block is usually connected to the PHY via a MII block.
  When transmitting data, the MAC modify the frames into frames appropriate to the physical medium. It adds preambles for clock synchronism, padding, an FCS for error identification, etc. The MAC then sends the data when allowed. This sublayer is also responsible of retransmission of packets if congestion, collisions or jamming is found.
  When receiving data, the MAC has to check the data integrity (with the CRC). It has to remove the preamble and padding too, before allowing the data to reach higher protocols [2].

The stack has been designed around the Ethernet protocol and the project has been created for a wired Ethernet connection. IEEE 802.3 [3] defines the physical layer and the media access control for wired Ethernet. In the other hand, the use of an LLC sublayer is mandatory for all IEEE 802 networks, but Ethernet is an exception to that.

2.3.1. Ethernet protocol

Ethernet is a computer network protocol that is widely used in local area networks, metropolitan area networks and wide area networks. The protocol has grown over the years and now supports a large array of bit rates and distances. The protocol is defined in IEEE 802.3 [3].

The protocol was originally used next to coaxial cables, but nowadays it has evolved and it used twisted pairs and fiber optic too. Devices like switches, hubs, repeaters and bridges have improved its network capabilities and scalability.

The Ethernet protocol is used in the data link layer, as a part of the MAC sublayer. Ethernet as a protocol has its own frame format; it will be explained in the next subsection. The protocol does not have flow control, and it does not have set up an automatic repeat request system. If incorrect packets are detected, they can only be cancelled or not retransmitted forward. The protocol cannot retransmit packets; this job is left to higher layer protocols.

The *Ethertype* field of the Ethernet frame can be seen as an LLC identifier, as it is used for multiplexing of top layer protocols. As this field is not mandatory, some packets may not have an LLC identifier. They should then use an IEEE 802.2 LLC specific header after the Ethernet one to allow for multiplexing between higher layer protocols. There are several types of Ethernet but the most common used by the IP protocol is the Ethernet II frame or DIX frame that does not use LLC (it just uses the Ethertype field).

2.3.1.1. MAC addresses

The addresses used in the MAC sublayer are called media access control addresses or MAC addresses. They are based on an early addressing system used in an early version of Ethernet.

The MAC address is a unique identifier that is assigned to each device. It is used for communicate inside the data link layer. MAC addresses are usually assigned at the time of manufacturing the device.

The address contains 6 bytes (48 bits), they are traditionally represented as a set of 12 hexadecimal numbers separated by a colon or a dash. The first six digits represent the manufacturer, the other six digits are an identification number for the device. Nowadays we also have 64-bit MAC addresses, protocols that use them like IPv6, usually translate the 48-bit MAC addresses to 64 bits.

2.3.1.2. Ethernet frame

A typical Ethernet frame consists in two main parts: the frame and the payload [3]. The size of an Ethernet payload is of 1500 bytes; in some high-speed variants of the protocol, there is also the possibility to send jumbo frames that increases the limit up to 9000 bytes. The main parts of an Ethernet frame can be seen in Figure 4, we proceed with an explanation of each one of the fields:

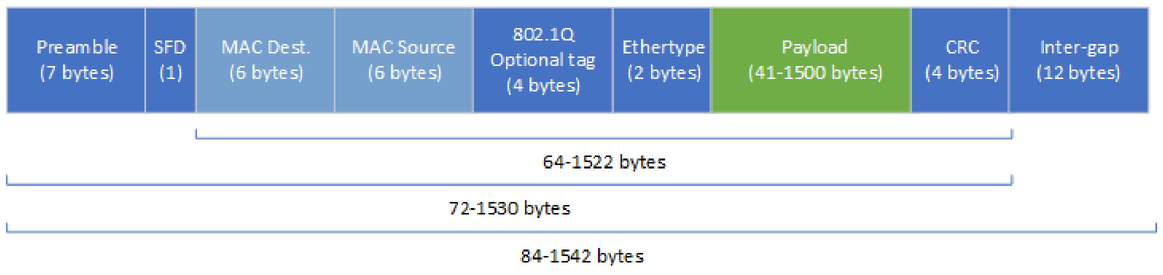


Figure 4. Ethernet frame with minimum and maximum number of bytes [3]

- **Preamble and Start frame delimiter (SFD):** An Ethernet frame always starts with seven preamble bytes. The preamble is a pattern of alternating 1 and 0 bits (starting with a 1). It is used to synchronize the receiver clock of the devices on the network. This provides bit-level synchronization.

The SFD follows the preamble and it adds byte-level synchronization to the frame. The SFD is in charge of breaking the pattern of bits of the preamble to mark the start of the frame. SFD follows the following sequence: 10101011 (0xD5). It starts following the preamble pattern, but the last bit breaks it.

The full sequence of the preamble plus the SFD for an RGMII interface is the following one (hexadecimal): 0x55 0x55 0x55 0x55 0x55 0x55 0x55 0xD5. The interface we use determines the correct way of transmitting and receiving bits. In an RGMII interface, we have an 8-bit wide output, so we can work with full bytes. In the other hand, for an MII interface of 4 bits wide the transmission will be performed by half-bytes (also called nibbles).

- **Destination MAC address:** This is the address of the device to which we are sending the packet. This address is sometimes unknown, so it may be necessary the use of address resolution protocols in the first place to discover the destination MAC.
- **Source MAC address:** The MAC address of the sender of the packet.
- **EtherType:** This field can represent two different things.
    - For values greater than 1536, it indicates the upper layer protocol (Network layer) that is encapsulated inside the payload. As an example, the IPv4 protocol it is a 0x0800 in the Ethertype field, and the ARP protocol 0x0806. Ethernet II frames use the field this way.
    - For values between 0 and 1500 it represents the length of the payload in bytes, this is used for IEEE 802.3 frames [3], in this case, an IEEE 802.2 [8] LLC header will be added next to it as we still have the need for multiplex to the upper protocol.
- **IEEE 802.1Q tag [6] or IEEE 802.1ad tag [7] (optional):** These tags are used to indicate the existence of a VLAN membership. When in use the value of EtherType is changed to 0x8100 to indicate the change in frame format. The field has 16 bits that are divided in 3 subfields pointing out the class of service and frame priority level (3 bits), flag to indicate frames eligible to be dropped if congestion (1 bit) and the VLAN identifier (12 bits).
- **Payload:** The minimum payload is of 42 bytes with 802.1Q tag and 46 without it. If the payload is shorter than that padding is required. The maximum payload is 1500 bytes and more in jumbo frames.
- **Frame check sequence (FCS):** The FCS is a 32-bit cyclic redundancy check (CRC). It allows the receiver to detect corrupt data inside the frame (even a single bit). The FCS is computed using all the frame fields with the exception of itself (the FCS field).

  The FCS is complemented by the sender to avoid false negatives that could be generated by data with trailing zeroes. The complemented FCS will then give us a specific CRC32 residue as a correct result (Ex: 0xC704DD7B).
- **End of frame:** The end of frame is a symbol or sequence that indicates the end of the data stream at the physical layer. For Gigabit Ethernet its 8b/10b encoding uses special symbols that are transmitted before and after the frame is sent.
- **Interpacket gap:** This gap represents idle time between packets. After a packet is sent, it is mandatory for the transmitter to send at least 96 bits before the next packet.

2.3.2. Address Resolution Protocol (ARP)

The Address Resolution Protocol (ARP) is a protocol used in communications to discover the link layer address of a device that is associated with an already given network layer address. This protocol is defined in IETF RFC 826 [9]

In the case of working with IPv4 over Ethernet, the ARP protocol will discover the MAC address of a specific system with the given IP associated with it. The ARP protocol is a very critical protocol in the Internet protocol suite, as IPv4 need that protocol to be able to transmit messages correctly. In IPv6 the NDP protocol replaces the ARP in this function.

The protocol uses a very simple format:

- **Hardware type:** Type of data link protocol in use Ex: Ethernet (4 bytes).
- **Protocol type:** Type of network protocol in use Ex: IPv4 (2 bytes).
- **Hardware length:** The length of a data link address (1 bytes).
- **Protocol length:** For the length of network layer addresses (1 byte).
- **Operation:** That specified the ARP operation we are performing (1 for request, 2 for reply) (2 bytes).
- **Sender hardware address:** The data link layer address of the sender.
- **Sender protocol address:** The network layer address of the sender.
- **Target hardware address:** The data link layer address of the intended receiver. (Empty in a request)
- **Target protocol address:** The network layer address of the intended receiver

The ARP has two modes of operation: Request and reply. In request, the sender is requesting the target hardware address of a specific target protocol address, during the consequent reply the target (now working as a sender) returns a message with the sender hardware address field filled with the requested answer. The targets field in this case point to the original requester. The size of an ARP message is variable and it mainly depends on the size of both types of addresses.

2.3.2.1. ARP cache

Addresses that are collected using the ARP protocol have to be stored somewhere. The ARP cache table is a repository used to store the data we gather using the protocol. In the ARP cache pairs of network layer addresses and data link layer addresses are stored and can be retrieved by the system when needed without the need of having to perform constant requests.

A system can work without an ARP cache, but adding one reduces greatly the network traffic, as the cache allows us to send a message to already matched devices without the need to request its hardware address every time.

Sometimes the IP of a device might change without notice, or even, another device can be assigned to an already cached MAC. In order to deal with this, the ARP cache has to renew its matched addresses when a certain amount of time passes (timeout), this time is variable and can be modified at the criterion of the designer. In addition, if a cache entry is not used for a specific time, it has to be removed from the cache, thus leaving space to entries of other devices that are being used at the moment.

An ARP cache can have static and dynamic entries. Static entries are manually added to the table and are permanent. Dynamic entries are added by the protocol normal operation and this are the ones that are only kept in the cache for a specific period.

2.4. Network layer

This layer groups the set of protocols in charge of transport network packets from a specific source to a destination host. This process can happen though one single network or multiple ones. Some of the functions of this layer are:

- **Connectionless communication:** There is no need of ACKs; connection-oriented communication is left to higher layers.
- **Host addressing:** There should be a system of identifiers in the network (ex: IP addresses).
- **Message forwarding:** To forward packets between different networks.

The main network protocol of our stack is the IP protocol, specifically it is used the IPv4 protocol. It is not necessary the use of IPv6, as the stack is designed for use in small or medium LAN networks that do not require that amount of addresses, and that are usually still using IPv4. The ICMP protocol is encapsulated inside IPv4, but it is not considered a transport protocol as it works as a complement to IP messages. Both protocols are explained below.

2.4.1. Internet protocol version 4 (IPv4)

The internet protocol version 4 (IPv4) is the fourth version of the well-known internet protocol (IP). The internet protocol is a network layer protocol, made for CL-mode communication on packet-switched networks, each data unit is addressed individually, and it is routed based in the information found in itself, not in a prearranged information. The protocol works on a best-effort delivery; it does not provide any kind of guarantee that the data is delivered or that the delivery meets the required quality. The users will receive a variable bit rate, latency and packet loss depending on the traffic load. Some of the aspects mentioned here can be changed or improved using the correct transport layer protocol, for example the Transmission Control Protocol (TCP) is a connection-oriented transport protocol that can be used with IPv4, in the other hand if we want to keep the connection connectionless, we have other transport protocols like the User Datagram Protocol (UDP). The protocol is fully defined in IETF RFC 791 [10].

2.4.1.1. Protocol header

An IP packet have a header section and a data one in Figure 5 we can see the structure of an IPv4 packet:

IPv4 Header Format																																
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 (if IHL > 5)																															
24	192																																
28	224																																
32	256																																

Figure 5. Format of an IPv4 header with byte references [10] [11]

We are going to explain now all the parts that forms an IPv4 packet:

- **Version:** The IP packet version, for IPv4 this is always 4, for IPv6 [11] it will be 6.
- **Internet header length (IHL):** Size of the IP header. The length is calculated multiplying the value at IHL by 32 bits, the maximum value inside the IHL is 15 so: 15x32=480 bits.
- **Differentiated Services Code Point (DSCP):** This field specified differentiated services (DiffServ) by RFC 2474, 3168 and 3260. Differentiated services are mainly used to provide quality of service, for example low-latency to critical networks (streaming), best effort to non-critical ones.
- **Explicit Congestion Notification (ECN):** This field allows end-to-end notification of network congestion without dropping of packets, this is an optional feature that can only be used if both ends of the communication are willing to do it. It is defined in RFC3268.
- **Total Length:** The entire packet size in bytes, it includes the header and all the data. The minimum size is 20 bytes (no data), and the maximum size is 65535 bytes. All host must be able to manage packets of at least 576 bytes.
- **Identification:** This field is used to identify the group of fragments of a single IP datagram.
- **Flags:** These three bits are used to control and identify fragments. The bit 0 is reserved, the bit 1 (Don't Fragment DF) forbids the fragmentation of the packet if it is set at '1' (if packet is too big it will be dropped). The bit 2 (More Fragments MF), for unfragmented packets the MF is '0', for fragmented packets all packets have that bit at '1' with the exception of the last one.
- **Fragment Offset:** This field is measured in units of eight-byte blocks. It specifies the offset that a particular fragment has regarding the original unfragmented packet. The first fragment has an offset of 0, and the maximum offset is set by the maximum data that can be sent with an IPv4 packet (65528 bytes of data).
- **Time To Live (TTL):** This field sets the datagram lifetime once sent. Its value is decreased by one every time the packet hops to the next network device (it is also called hop limit). When the field reaches a 0 the router discards the packet, and it usually sends an ICMP Time Exceeded message to the sender.
- **Protocol:** Here we defined the protocol used in the upper layer (transport layer), for example UDP (17), TCP (6) or ICMP (1).
- **Header Checksum:** The checksum is the 16-bit one's complement of the one's complement sum of all 16-bit words of the header. When the checksum field is

considered zero during the calculation of it. It is used for error checking, when the packet arrives a router or device the checksum is calculated, if the checksum calculated does not match the one in this field the packet is discarded.

- **Source Address:** The IPv4 address of the sender of the packet.
- **Destination Address:** The IPv4 address of the receiver of the packet.
- **Options:** Not usually used.

2.4.1.2. Fragmentation

The protocol supports fragmentation, if we want to pass through a link a packet that is too big, the packet needs to be broken into smaller pieces and then reassembled by the host at the end. The maximum transmission unit (MTU) is the maximum size allowed in a link, if this size is lower than the packet length it has to be fragmented. Fragmentation of packets for IP is defined in RFC 791 [10] and the reassembly algorithm in RFC 815 [12]. As we have already seen the IPv4 header uses some of its fields for fragmentation.

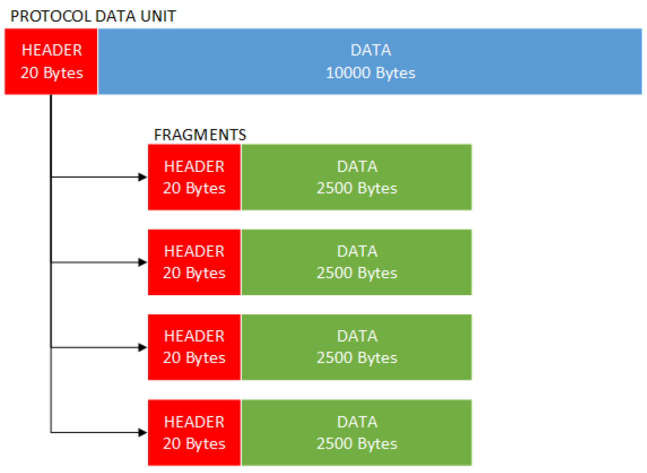


Figure 6. Example of fragmentation of a Datagram [10]

When fragmenting each fragment of the original datagram will get its own fragment header, this new header contains the fragment offset corresponding to each fragment. All fragments with the exception of the last one will have the More Fragments flag set to '1'. The length field of the IP header will have the fragment length, and the checksum of each fragment is calculated for separated. The rest of the header is like the original one. The receiver end is in charge of the reassembly of the fragments in the correct order using the offset field, the reassembly can be also made by an intermediary.

2.4.1.3. Protocol addresses

IPv4 uses 32-bit addresses, the total of possible addresses in a network is then (2^{32} = 4294967296). Addresses are usually represented as four decimal octets ranging from 0 to 255 and separated by dots. The addresses are divided in two main parts, first the network portion that is usually located in the first bits, secondly the host identifier that occupies the rest of the bits.

In the original design, the network number was the first (or highest) octet, but this method only allowed 256 networks, so it was quickly outdated [13]. New methods were created to increase the number of networks as the IP address exhaustion has being a critical problem since the almost the beginning, as the original design did not have enough capacity. Things like mobile devices and the rapid growth of the internet all over the world aggravated the problem.

In 1981 classful networking was created to try to extend this limit [10]. This system defined five classes from A to E. Classes A, B and C had different bit lengths for network identification, the rest of the address was used as previously to identify the host. Class D was defined for multicast and Class E was reserved for the future.

Later on, Classless Inter-Domain Routing (CIDR) replaced the system of classes. CIDR is a standard that eases the routing by having blocks of addresses grouped into routing table entries [14].

CIDR uses variable-length subnet masking (VLSM) to assign IP addresses to subnets depending on the need of each one of them. The division between the network and the host can happen in any of the 32 bits that an IP address has. Subnet masking uses a combination of bits to delimitate a network, its main function is to tell to devices which part of the IP address is the network and which one the host.

The address groups are called CIDR blocks and share a sequence of bits in their IP addresses representations. The blocks are identified with a similar format than IP addresses with four decimal octets separated by a dot and a slash at the end followed by a number from 0 to 32.

The first numbers are interpreted like an IP address, the slashed number is the length of the prefix, counting from the left it represents the number of common bits in all addresses. To read the CIDR address correctly, we need to represent the address in binary.

2.4.2. Internet Control Message Protocol (ICMP)

The Internet Control Message Protocol or ICMP is a support protocol of the Internet Protocol. The ICMP complements the IP Protocol with a range of options, it is used by network systems and devices to send error messages and other types of information like when a service is not available, or a device cannot be reached, etc. The ICMP protocol formal definition can be found in RFC 792 [15].

ICMP messages are normally used for control or diagnostic. It is also used as a response to errors in IP operations as specified in RFC 1122 [1]. Some of the possible applications that the ICMP messages have are for example: pings, time exceeded messages, traceroute commands, destination unreachable messages, router solicitation and advertisement, etc.

ICMP uses the IP like an upper layer protocol, but ICMP actually behaves as an integral part of the IP protocol. ICMP is a network layer protocol, so it does not have a port number associated.

ICMP Header Format																																	
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**	Type								Code								Checksum															
4	**32**	Rest of Header																															

Figure 7. Format of ICMP header [15]

The ICMP works using the IPv4 protocol, in the IP header it is identified as the protocol number ‘1’. All the ICMP packets are made by an 8-byte header and a data section of variable size. The first byte of the header is reserved for the type of ICMP message, the second byte called code specifies an ICMP subtype. The third and fourth bytes are a checksum. The next bytes are of variable size and content, depending on the type and code we want to send. It is important to notice ICMP error messages maximum length is 576 bytes.

ICMP control messages are specified by the type and code fields. The code field is used to give a context to the message. Since the introduction of the protocol some control messages have been deprecated and are no longer available to use. Here we have an example of the most used ones:

Table IV. Example of some ICMP control messages [15]

Message name	**Type**	**Code**	**Description**
Echo Reply	0	0	Echo reply (used to answer ping)
Destination Unreachable	3	0	Destination network unreachable
		1	Destination host unreachable
		2	Destination protocol unreachable
		3-15	Others…
Echo Request	8	0	Echo request (used to ask ping)
Time Exceeded	11	0	TTL expired in transit
		1	Fragment reassembly time exceeded

2.5. Transport layer

The transport layer is set to provide host-to-host communication. This layer can implement many functions, like connection-oriented communications, flow control or multiplexing of higher layer protocols (applications or services). It is the layer with a highest focus on reliability. Its main functions are:

- **Connection-oriented communication:** TCP is known for having it. UDP is an alternative transport protocol that is actually still connection-less.
- **Same order delivery:** The network layer does not guarantee that this is solved here.
- **Reliability:** Error detection techniques, acknowledgements, automatic repeat request schemes, etc. In order to avoid the loss of packets during network congestions or other issues.

- **Flow control:** Control between two nodes to prevent a sender that is too fast for the receiving buffer.
- **Congestion control:** To avoid the collapse of the network. Things like reducing the rate of sending packets, and slow-start, etc.
- **Multiplexing:** The option to multiplex between the higher layer nodes is available thanks to ports.

The most used protocol at the transport layer in the Internet suite is the TCP, despite that, the stack has been designed putting speed in front of reliability, and this make it preferable to use the UDP protocol. Both protocols are going to be explained next, but the focus will be in the UDP protocol, as it is where we have the real interest.

2.5.1. User Datagram Protocol (UDP)

The User Datagram Protocol or UDP is one of the two main transport layer protocols used in the internet protocol suite. It is formally defined in RFC 768 [16]. It is a transport protocol that is usually used as an alternative to TCP. UDP uses a connectionless communication model, it has checksums for data integrity and it uses port numbers as a way for addressing messages in the device. It does not have handshakes so it is exposed to unreliability, there is not guarantee of delivery, ordering or duplicate protection.

If we are looking for error checking, error-correction and reliability using TCP is recommended. UDP is usually used for time-sensitive applications and real time systems as it is more important the speed of the transmission (not waiting times, not retransmission), and the drop of packets can be allowed.

2.5.1.1. Protocol Header

UDP 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**	Source port																Destination port															
4	**32**	Length																Checksum															

Figure 8. Standard format of UDP header [16]

- **Source port:** The sender's port number, assumed to be the port where we should reply. If the field is not used, it should be 0.
- **Destination port:** The receiver's port. This port is mandatory, in our system each service has a port assigned and is in charge of tell the stack when a message is directed to it.
- **Length:** The length in bytes of the UDP header and the UDP data. The minimum length is 8 bytes (only header), the maximum length is 65507 bytes (65535 – 8 UDP header – 20 IP header).
- **Checksum:** The UDP checksum for error-checking, the field is optional for IPv4 (set to 0 if not used). The UDP protocol is calculated with the UDP header, the data and an IPv4 pseudo header.

2.5.1.2. Ports

UDP ports are 16-bit integer numbers (between 0 and 65635). Ports are used to multiplex the messages between all the services and applications linked to a specific IP address (assigned to a specific device). Port 0 is reserved, but it is allowed as a source port value if the sender is not expecting a reply.

2.5.1.3. Checksum computation

The UDP checksum is calculated by the 16-bit one's complement of the one's complement sum of the UDP header, the data and a IP pseudo-header, all padded with "zero" bytes at the end (if necessary) to make it a multiple of two bytes.

If the network protocol is IPv4 (as in our stack), the pseudo header computed has to be formed by the Source IPv4 Address, the destination IPv4 Address, a byte of zeroes, the protocol field, and the UDP length (notice that this length is asked again in the UDP header).

2.5.2. Transmission Control Protocol (TCP)

The TCP protocol is the other main protocol of the Internet model (also called TCP/IP model). The protocol is defined in RFC 793 [17]. It is the completely opposite than UDP in most of its features.

TCP It is characterized for being a very reliable protocol, it provides error detection and retransmission of wrong packets, flow and congestion control, acknowledgments, handshakes and the most important feature; it is connection-oriented, TCP requires three packets to create a connection. It can rearrange the packets in the correct order. TCP is slower than UDP but more reliable, we are guaranteed to receive the packets.

2.6. Application layer

In this last section, we have what we called application layer, here we have joined the session, presentation and application layers of the OSI model. This last layer groups the protocols in charge of process-to-process communication.

Here we will describe all the services that will use our designed stack to transmit and receive data. In the design, services of the application layer are created as separated IP cores that will be then connect to the UDP/IP stack for its use. The only exception to that is the DHCP protocol.

The DHCP protocol is necessary for a good performance of the stack inside network, this protocol is an integral part of the stack IP core. The stack has been designed for an easy use inside a non-controlled network, it should be "plug and play". This makes DHCP a mandatory protocol, as we do not want the need to configure the addresses of the devices we are connecting.

2.6.1. Dynamic Host Configuration Protocol (DHCP)

The Dynamic Host Configuration Protocol (DHCP) is a protocol of the application layer that is used to manage an UDP/IP network, it is derived from the older bootstrap protocol (BOOTP). They are both defined in RFC 2131 for DHCP [18] and RFC 951 for BOOTP [19].

The DHCP protocol is used to assign IP addresses and other network parameters to each network device, and to each new device that joins the network. This allows the devices to communicate with themselves and other networks. A DHCP Server is the one in charge of assigning these parameters and addresses to all the DHCP clients, usually the DHCP Server is located at the router.

A typical DHCP session consist in four steps as seen in Figure 9: First the client (the device) sends a Discover broadcast to the network, the discover is searching for DHCP servers on the network, the servers that receive the discover message will answer with an IP offer, during the discovery phase the client can ask for more information about the servers like; server IP address, subnet mask, domain name, etc. all this info will be provided by the server's offer.

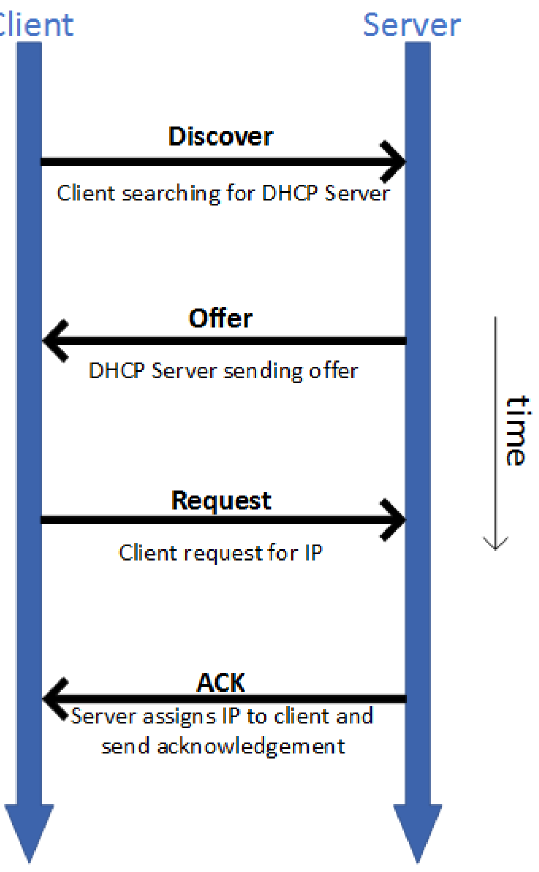

Client
Server
Discover
Client searching for DHCP Server
Offer
DHCP Server sending offer
Request
Client request for IP
ACK
Server assigns IP to client and send acknowledgement
time

Figure 9: DHCP protocol session [18].

Once the offer has been received the client will take the offered IP and additional information and it will broadcast a Request of that IP address. If the server is ok with that request it will answer with an ACK message and the address will be assigned to that specific client, if the request is rejected the server will send a NACK answer and the client will have to request for a different IP.

The IP address will stay assigned to the client for a time specified in the ACK message. The IP lease time will be chosen by the server, typically the DHCP lease time default setting in most servers is 24 hours. Servers should try to avoid setting the lease time too low as this can cause interruptions to service.

2.6.1.1. DHCP header and options

The DHCP header is the same in all the messages, changing just the values of some field and the last field (Options) we are able to send all kind of DHCP messages (discovery, request, offer, ack, nack, etc…), these are the fields of a DHCP packet:

- **Opcode (1 byte):** Distinguish between a client request (0x01) message and a server reply (0x02)
- **Hardware type (1 byte):** Specifies the used network architecture, for example if set to 0x01 it will represent an Ethernet network.
- **Hardware address length (1 byte):** Represents the length of the hardware address. For Ethernet the length of a MAC address (48 bits).
- **Hop count (1 byte):** Number of relay agents that the message has passed.
- **Transaction ID (4 bytes):** ID number that identifies the client request and it is used to match request with replies.
- **Number of seconds (2 bytes):** Time that has passed since the client has started the DHCP process.
- **Flags (2 bytes):** If set to '1' it will mean that all replies have to be broadcast to the client.
- **Client IP address (4 bytes):** The client's IP address, once it has been confirmed, if the client still doesn't have an IP this field is set to all zeroes.
- **Your IP address (4 bytes):** The IP offered by the server to the client.
- **Server IP address (4 bytes):** IP of the next server the client has to contact to follow the configuration process.
- **Gateway IP address (4 bytes):** Relay agent IP address, filled by the relay agent if some exists.
- **Client hardware address (16 bytes):** Client hardware address (MAC address). It may require padding with zeroes to fill the 16 bytes.
- **Server name (64 bytes):** Optional server host name.
- **File (128 bytes):** Optional boot file name.
- **Magic cookie (4 bytes):** This field identifies the mode in with the next set of data will be interpreted. For DHCP the magic cookie must be 63.82.53.63 in hexadecimal.
- **DHCP options (Variable):** That field contains the data of the DHCP packet, a client must be prepared to receive options of at least 312 bytes (total message of 576 bytes). DHCP options are a set of custom configuration parameters and control information that can be added in the last field of a DHCP packet [20].

The Options field is the most critical of the packet, in there we will make our data requests to the DHCP server, and we will retrieve most of the information of the replies. All options begin with a tag byte that identifies the option, then for options with variable length it follows a length field, this length field doesn't include the tag and length fields itself, only the option data that will follow next. We can find some of the most important options in the following Table V. Some of the most common DHCP options, most of them used in the design of the project [20].. The option field ends when the option with code 255 is found, this option marks the end of the field and packet.

Table V. Some of the most common DHCP options, most of them used in the design of the project [20].

Option Name	Code	Length	Description
Pad Option	0	Fixed	It can be used to align other fields
End Option	255	Fixed	Marks the end of the options, it can be followed by padding (pad option).
DHCP Message type	53	Variable	Identifies the type of DHCP message. 1 Discovery, 2 Offer, 3 Request, 4 Decline, 5 ACK, 6 NACK, 7 Release, 8 Inform.
Client Identifier	61	Variable	Identifies a DHCP client, it usually has the hardware type and address.
Parameter Request List	55	Variable	Used by the client to request a list of configuration parameters. (Subnet Mask, Domain Name, Router, etc.)
IP Address Lease Time	51	Variable	In discovery and request it allows to request a lease time, in offer it shows its offered value. (time in seconds)
Domain Name	15	Variable	Domain name for DNS
Subnet Mask	1	Variable	Specified the client's subnet mask
Requested IP Address	50	Variable	Used in Discovery or Request to request a particular IP.

2.6.2. Other services

We can add any number of services to the stack, the stack was mainly designed to read data from a DAQ block and sent the data to another host of the network. The DAQ is an example of a service block. Another service that have been suggested is an SPI interface for the configuration of ICs or other electronics. Some testing services have been created for the design, for example an echo block that returns the same message that we sent to it. This service only has the purpose of testing the design and testing that the connection between host and processes is correct (with a ping we are only testing the network layer).

2.7. <u>Existing implementations</u>

Some already existing implementations can be found with similar applications. Some companies have released IP cores with similar characteristics (or even more) but they were directly discarded for its high prices (more than 4000$ for the IP core without extra protocols). One great example of that is the UDP/IP Ethernet IP core of Enclustra [21]:

- It has 1Gb/s, 100Mb/s and 10 Mb/s operations
- UDP, IPv4 and Ethernet protocols

- ARP protocol
- It works with MII, RMII, GMII and RGMII interfaces
- Destination UDP port, IP address and MAC address filtering
- UDP checksum generation and check, Ethernet CRC too
- Multiple UDP ports with dedicated Rx and Tx interfaces for each one.

In the territory of the open code we can find one remarkable core called 1G eth UDP/IP Stack and provided by Peter Fall and the FIXQRL project [22], it has the following remarkable characteristics:

- UDP, IPv4 and Ethernet protocols
- ARPv2 protocol with cache of multiple slots
- Separate clock domains for Tx and Rx
- UDP ports user filtering
- AXI interface to connect MAC core.
- It uses the Xilinx's MAC interface.

This solution is good but it is not very customizable, it does not have implemented a good system to increase its number of protocols in the different layers and a few protocols are missing like DHCP or ICMP. It is also designed specifically to be used with an AXI interface at its input.

At the end, it was decided for the Technological Unit of the ICCUB that it would be better to design its own core specifically designed to fit the needs of the Unit's projects and collaborations. Creating our own design we have the complete control over the core, from its design decisions to its final performance and possible bugs or problems. It allows for further improvements following a specific route map that fits the needs of the unit. And it does not require to rely in 3rd party developers with unknown skills. This project is big enough to be considered a challenge, and an a refined and improved final product has real economic value and can be useful in a huge range of applications and future projects.

3. Design of the stack

The stack has very clear objectives. The initial idea has been to create an FPGA stack to work in a PCB, which main function will be to read data from a DAQ system. The stack should be able to send and receive data in any network to any host connected to the network, and it should also be able to transmit and receive via Internet (using port forwarding). The last condition is simplicity, the stack should be easy to use, and with the minimum number of configurations once the system is embedded inside an FPGA. These conditions fixed the protocols we have used for the stack:

- The RGMII was selected, as it was the chip used in the implementation boards.
- It was decided to use Ethernet, IPv4 and ARP, as they are very common and robust protocols that allow easy and reliable communications inside any network, including the Internet.
- The UDP protocol was selected, as it was preferred simplicity and speed than reliability to the stack.
- The DHCP was necessary to easily connect the system to different networks without the need of additional configurations.
- The ICMP protocol was added as a way of testing the connection between hosts (ping).

This set the initial conditions for the design, each protocol and section has been studied and designed with its own peculiarities, and they are explained in their corresponding sections.

The stack was firstly designed in a theoretical level and then was translated to VHDL code. In Appendix A is possible to check the true hierarchy of the VHDL implementation. At the end, the final design has been fully implemented in VHDL and it is fully implementable to the Cyclone 10 LP family of FPGA systems. The system may work in other Intel FPGA families, but some IP components and macros may be incompatible. If it is wanted to use the core with an incompatible family or an FPGA system of a different manufacturer the macros that have been used (FIFO, PLL, RAM, etc.) should be translated to the specific device.

3.1. UDP/IPv4 stack core structure

The stack designed in VHDL has been created with a very specific structure that can be checked in Figure 10.

The reception and the transmission side of the system have been separated in two independent sets of blocks for the main communication protocols. The RGMII interface, the Ethernet frame, the IPv4 protocol and the UDP protocol are all divided in a reception set of blocks and a transmission set. Both sets work with complete independence of each other, being possible to receive messages from a specific system or device, while sending a message to a different one.

Auxiliary protocols like ARP, ICMP and DHCP are built as a single block. The three protocols have to receive final datagrams from the stack, and they all work as the final step in a transmission, with these characteristics, it has been more comfortable to design these systems as a single entity.

A system of filters, demultiplexers and arbiters has been created to manage the entire set of protocols request and data moving all over the stack. The full system will be explained with more detail in the following sections *3.2 Reception side* and *3.3 Transmission side.*

The stack is highly customizable; the number of services added at the top layer can be extended indefinitely for reception and transmission (independently of each other), things like the ARP cache or the ARP timers are also configurable. There is also the possibility of disable the MAC and IP filters. This is the list of all configurable properties of the IP core:

- **SERVICES_TX:** Number of services that need to work with the transmission side of the stack. Services that will require transmitting a message.
- **SERVICE_RX:** Number of services that need to work with the reception side of the stack. Services that will require receiving a message.
- **IP_DST_FILTER:** Enable/disable the IP address filter, this filter only let pass packets where the destination IP address of a packet is our own or a broadcast., it also checks the subnet too.
- **MAC_DST_FILTER:** Enable/disable the MAC address filter, this filter only let pass packets where the destination MAC of a packet is our own or a broadcast.
- **CLK_FRQ_HZ:** Clock frequency in Hertz (Hz), this is used for the timers to calculate real time.
- **DHCP_XID:** DHCP transaction identifier. A 32-bit identifier generated by the client allows to match up the request messages with its replies from DHCP servers.
- **IP_ADDR_SET:** IPv4 address of the FPGA, if set to 0.0.0.0 (default) the IP will be requested to the DHCP client, otherwise it will be set as specified.
- **ARP_SIZE:** Size of the ARP cache table. The default is 2048 bits, and the system is designed to allow sizes from 0 to 65536 bits.
- **ARP_TIMER_SIZE:** Size of the timers that manage the timeouts at the ARP table. This generic determines the size of the subtraction vector, being $2^{ARP\_TIMER\_SIZE}$.
- **SILICON_DEBUG:** This Boolean is used for testing, when true probes and sources are activated all over the stack. It should be set to false by default.

There are some generic parameters that should not be modified. These constants are set as generic as they are part of the declaration of inputs and outputs of the stack. But they should be always set to the default values, because a change in their values can only be made with the correspondent modification of the internal stack design.

- **DEFAULT_SERVICES:** The default number of services, there is only the DHCP as a default service.
- **AUX_BUS_SERVICES:** The number of additional bits that we are getting through the top layer arbiter. This number should be always the same (83) as it sets the length of a lot of vectors in the design.

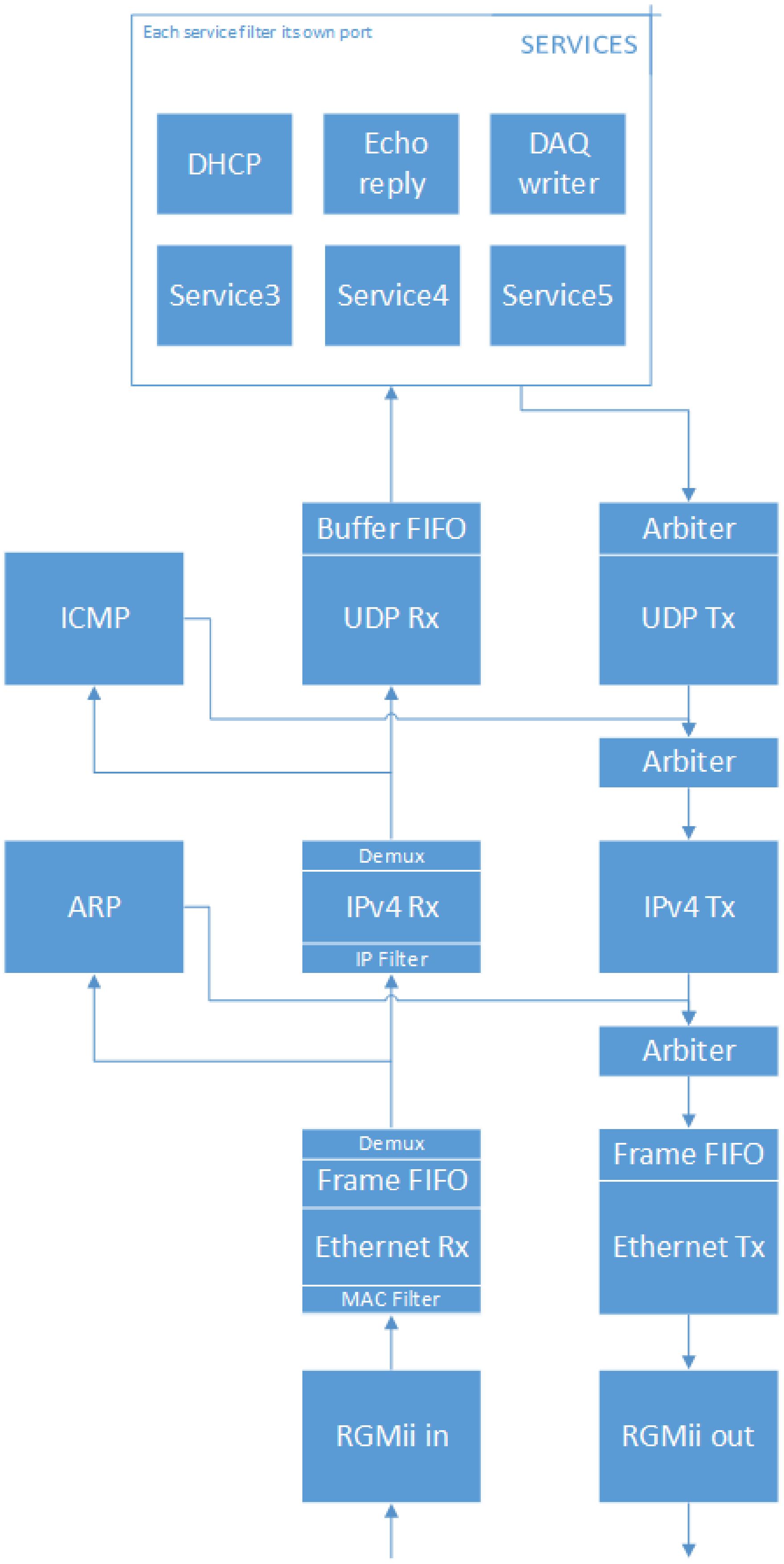


Figure 10. Block diagram of the full UDP/IPv4 stack core without timers

3.2. RGMII interface

The design has two blocks that work as an interface for the RGMII to the rest of the system. These blocks (one reception and one transmission) use the standard RGMII signals with an additional reset. Both blocks use a 125 MHz clock and are connected to an external physical RGMII chip. We will call RGMII MAC the interface set up inside the FPGA and RGMII PHY the physical chip in the board.

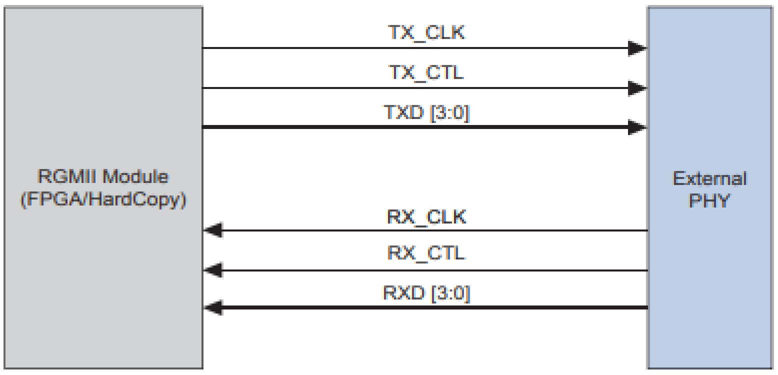


Figure 11. Signal connections between the RGMII MAC and RGMII PHY [5]

RGMII data works in DDR mode, so it is transmitted or received in both clock cycles. Usually the clock and data signals coming from the RGMII PHY are generated at the same time, so both signals are aligned. This has to be avoided, so clocks have to be delayed by some methods. This delay can be added in the PCB design, but this can needlessly complicate the design, another option is to use an RGMII PHY internal delay, but this is not always available in older chips.

The design guidelines of both RGMII MAC blocks have been extracted from an Intel manual [5]. We will try to summarize the ideas and design concepts of this designing guide,

3.2.1. RGMII Physical chip

The board used in the implementation (Section 4.1.1. FPGA Board) uses the PEF 7071 as our physical RGMII chip. The PEF 7071 is a single port gigabit Ethernet physical interface for speeds of 10, 100 and 1000 Mbit/s [23].

This IC is highly configurable, for example, it can be set as RMII, RGMII, RTBI or SGMII. Configurations are hard-wired in the PCB. So, our specific IC is always set as an RGMII interface and this can't be modified. The type of interface is configured by soft pin-strapping, a resistor and a capacitor are added to a pin, the combination of the resistor and capacitance values give us a vector of bits, this vector is translated into a specific configuration for the chip.

For the RGMII interface we can also configure the internal delays for the clocks, we have two independent delays of 1.5ns for TX_CLK and RX_CLK. The PCB board [24] sets the TX_CLK delay as '0' (disabled, no delay), and the RX_CLK delay as '1' (enabled). The designed FPGA MAC has had to have these configurations taken into account for its design.

3.3. Reception side (Rx)

The reception of a message occurs when a packet is received thought the physical Ethernet cable of the board. The reception is done by steps, first we receive a packet through the physical layer, this packet is going to be read by the Ethernet block and then, its payload is going to be stored into a FIFO memory [25]. Once the full Ethernet frame is received, upper protocols will read from that FIFO, the data will be interpreted from lower to upper protocols until it arrives to the service FIFO, this last FIFO where we will start the UDP data, from this last FIFO all services will take its data. If an auxiliary protocol is required, the process will end in lower layers.

When a message is received, the system has a set of filters that allows us to sort of the messages. If a message does not match the device's MAC, IP or the service's port is then discarded at different steps of the reception. Wrong CRC or Checksums result in discarded packets too.

The system also includes multiple demultiplexers that read the received packets and interpret the destination protocols, sending the information to them.

- MAC filter and first protocol demux: After a packet is received and interfaced by the RGMii_in block, it is sent to the Rx Ethernet block (eth_frame_rx). There, using the eth_frame_validator we check that the MAC fits our own or a broadcast one. We check the CRC too, a fail in one of the two, discards the packet. If the MAC and CRC are correct, the system reads the Ethertype and activates a flag to the correct protocol (ARP or IPv4).
- IP Filter and Second protocol demux: Once the Ethernet protocol has activated the IPv4 Rx block (ip_rx), the block starts to read the received message, here the IP will be checked. If the IP matches our own or a broadcast one, we message will not be discarded. A valid message will be read and an upper protocol will be demultiplexed from its header and called with a flag.
- Port Filter: Once the message arrives the Rx UDP block (udp_rx) the data will be read until the end. The destination port it is going to be sent to the application layer and the service that occupies that port will be activated, this service will use the top layer FIFO to gather application layer data.

3.3.1. RGMII Rx interface

Here, we are going to focus in the design of the receiving interface block (RGMII Rx). In Figure 12 we can see the proposed design of the RGMII RX block. As we can see the RGMII PHY is multiplexing signals to generate DDR outputs to our FPGA (RGMII MAC). These signals have to be captured at both edges of the clock.

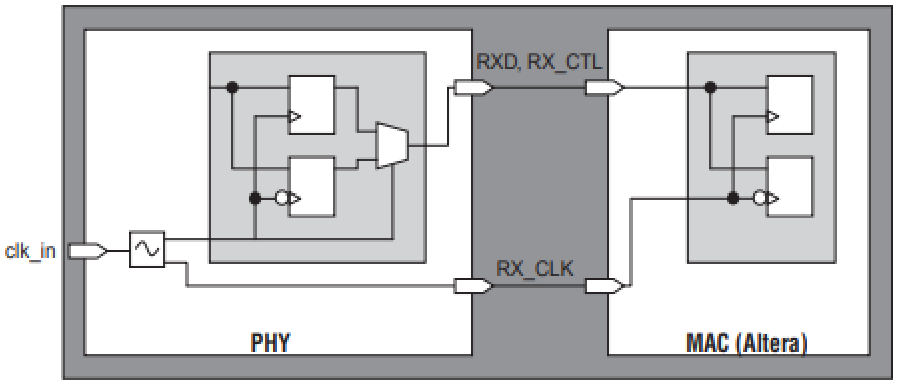


Figure 12. Block diagram of the receive interface [5]

The easiest way to transform this DDR signals in two differentiate output is to use Intel's double data/output (DDIO) I/O register [26]. We are going to use the so called ALTDDIO_IN, as we want to take a single DDR input and then output two signals from it. This block gives us a single clock and a single input, but outputs two different signals one for the rising edge of the input and another for the falling edge. Two ALTDDIO_IN cores have been necessaries, one for the RXD signal, and the other one for the RX_CTL signal. RX_CTL is not a DDR signal but it has been decided to use the same circuitry as the data, in order to have the same delays and therefore to avoid losing synchronism between both signals.

As we already know the RX clock has to be delayed in regard to the data signals. This can be achieved with an internal RGMII PHY delay, as we already explained this delay has been implemented in our RGMII PHY by the PCB. With the internal delay enabled we are receiving the clock signal center-aligned to the data, as we can see in Figure 13. Then, it is not necessary to apply any more changes to the signals at the FPGA side, as the received signal is already correctly aligned.

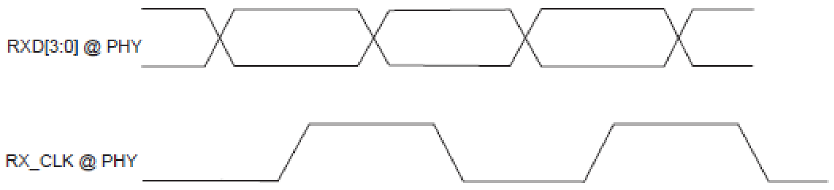


Figure 13. RGMII physical chip generated RX_CLK signal and RXD (data), with internal delay enabled. With the internal delay both signals are center-aligned [5].

The final design obtained can be seen in Figure 14. With the internal delay activated in the RGMII PHY side, the digital circuit at the FPGA is simplified a lot. Some timing constrains were added to the RGMII (both Rx and Tx) and are explained in section 3.10.1.

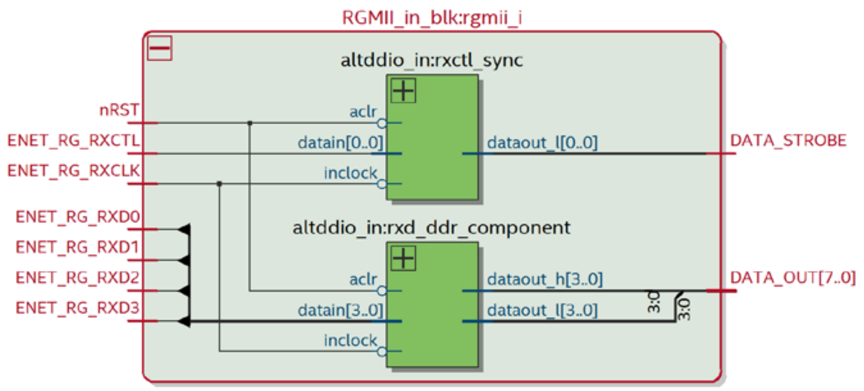


Figure 14. Final implemented design of RGMI RX core [5].

3.3.2. Ethernet Rx core

The receiving Ethernet core is in charge of decoding the packet frame as it arrives, checking that the CRC is correct and storing packets inside the Rx FIFO. It can demultiplex packets to upper layers, and filter wrong packets (wrong destination MAC, CRC, protocol, etc.). The operation is managed by an FSM and with counting of clock cycles. The block has been designed as four children blocks that interact between them, each one with a clear function:

- eth_frame_rx_fsm: Contains the FSM and manages the process.
- eth_frame_validator: Validates the CRC, Destination MAC address and Ethertype.
- CRC: Calculates the CRC of the frame.
- dcfifo: The RX FIFO.

This is a summary of the core's normal operation:

- The system starts in the IDLE state, it will start its process when receiving a high signal from ENET_DREADY. This signal is connected to the RGMII strobe also called RX_CTL.
- Once the FSM is activated, we will then go PREAMBLE state, here we will check if the received data is the correct preamble (0x55h bytes), if it is not, we will discard the packet going to the state WAIT_EOF, where we will wait until ENET_DREADY is 0 again, meaning the end of the frame. If the preamble is correct, we will reach the start-of-frame (SOF) state, where we will check if the SOF field is correct too (a single 0xD5h byte). If it's not we will again discard the packet, if it is correct, we will reach the MAC_RX state.
- In MAC_RX we will start to calculate the CRC of the received data. Here we will receive the MAC addresses and shift them inside a register. If the MAC. We will repeat the same with both Ethertype states (ETYPE_MSB and ETYPE_LSB).
- We will then reach the state PAYLOAD_AND_FCS, here we will start receiving the payload data, and storing them into the FIFO with some exceptions. We have designed a pipeline (pipeline_4), to remove the FCS trailing from the payload, data will be delayed 4 cycles. When the packet stops receiving data (ENET_DREADY = 0) we will mark the end of the packet in the RX FIFO (PKT_FIFO_EOP = 1) and reach the EOF state.

- In the end-of-frame (EOF) state, we will set the flag PKT_FIFO_NEW_PKT, this flag will tell the frame validator block that we have finalized the storage of a packet inside the FIFO. With this information the block will check if the Ethertype matches a known protocol, if the destination MAC address matches the local host one (or it is a broadcast) and if the received CRC matches the calculated one. If all the criteria are met, we will set PKT_SKIP to 0, and we will send an activation signal to the upper protocols PROTOCOL_SEL <= ethertype_match. If some of the criteria does not match, we will set PKT_SKP to 1, and PROTOCOL_SEL will be 0, meaning the upper protocols won't be disturbed. With PKT_SKP being 1, the FIFO will be cleared, dropping all the frame at once.
- After dropping or notifying the upper protocols about the received frame we will reach the IDLE state again and wait for the next reception.

3.3.3. IPv4 Rx core

The IPv4 Rx core is in charge of decoding the IP header of the packet. Their functions are: filtering the packet by checking if the destination address matches with the local host, or at least it's a broadcast. Checking who is the upper protocol, and multiplexing the packet to it, if the stack doesn't have that protocol the packet should be discarded. It reads and stores other important data like the packet ID, the offset or the fragmentation flags.

The core is managed by an FSM, the passage of clock cycles is supervised by a digital counter. This is the operation of the core:

- The FSM starts at the IDLE start, it will start its operation with the trigger of NEW_PACKET, this signal tells us that a packet has been multiplexed from the MAC layer to us.
- The next states are very straightforward, we will stay in them for a number of cycles, and we will read the data that is coming to us and classifying it as Length, ID, offset, and the source and destination addresses. Each state covers a field (or two) of the header: VER_IHL, DSCP_ECN, TOTLEN, ID, FLAGS_OFFSET, TTL, HCKSUM, SCR and DST.
- Once we reach the last state End-of-header (EOH) we will stop reading from the FIFO, here we will check if the upper protocol matches one of the known ones, if this happens and a combinational set of conditions are true (subnet is correct, local host IP or broadcast IP) we will multiplex the packet to a higher layer protocol. Then the FSM will wait until the FIFO is empty or we receive a new packet SOP (Start of packet) being 1.

The core doesn't check the value of the checksum, the checksum of the IPv4 is redundant (IPv6 has eliminated the checksum) and it doesn't cover the data, it is only a checksum of the IP header. The checksum will be checked in the UDP Rx core, as this last checksum covers the data, the UDP header and part of the IP header (as a pseudo header) and we also have the Ethernet CRC that covers the full packet at the beginning.

3.3.4. UDP Rx core

The UDP Rx core has the function of decoding the UDP header of a received packet. It resolves both ports of the header; the length and it calculates and checks the checksum. It has a FIFO memory integrated where it stores the payload data. Services will then acquire this data using this FIFO. The core is managed by an FSM, its operation follows these steps:

- The FSM starts at the IDLE state, it will be activated when receiving a NEW_PACKET signal, this signal comes from the lower layer (IPv4) and it means that the packet has been multiplexed to this upper protocol.
- The next state CHECKSUM_CALC starts to calculate checksum of the packet, the UDP packet includes a pseudo header that is calculated now. This previous calculation includes the IP source address, IP destination address, the protocol (UDP) and a byte of zeroes (the checksum is a 16's complement), this data has been obtained at the IPv4 protocol. Next state is SCR_PORT_L.
- In the next states we will acquire all the fields of the UDP protocol, as we receive them, we will keep calculating the checksum, in the checksum field we will input the UDP length twice, as the IP pseudo header includes this field again. Then we will reach the DATA state.
- At the DATA state we will output the destination port to the services, if we have a match with a service port, we will start storing the data inside the UDP FIFO, if not, we will discard the data. We will keep calculating the checksum as the UDP checksum includes the data. When the Rx FIFO is empty or we have received an EOP (end of the packet) signal we will go to the EOP and FLUSH states.
- In the EOP state we will wait a cycle and go to FLUSH, the FLUSH state will then check the checksum result, first we will need to input a byte of zeroes to the checksum if the number of inputs is not a multiple of two (we have used a counter to check that). Once the checksum is correctly calculated we will check if our calculation matches with the header's checksum. If it does not match, we will discard the packet, if it matches or the header's checksum is all zeroes (checksum is not mandatory in UDP), then we will send a trigger to the service that will allow the service to start processing (reading) the data stored in the UDP FIFO. After all that, FSM will then wait until the FIFO is empty or we receive signals of EOP or SOP meaning we have reached the end of the packet or received a new one.

3.4. Transmission side (Tx)

The transmission of a message can be started by a service on the top layer or an auxiliary protocol (ARP, ICMP and DHCP). The following explanations will be focused around the transmission of a packet using the full UDP/IPv4 stack, that means that the transmission request will be made by protocols at the application layer.

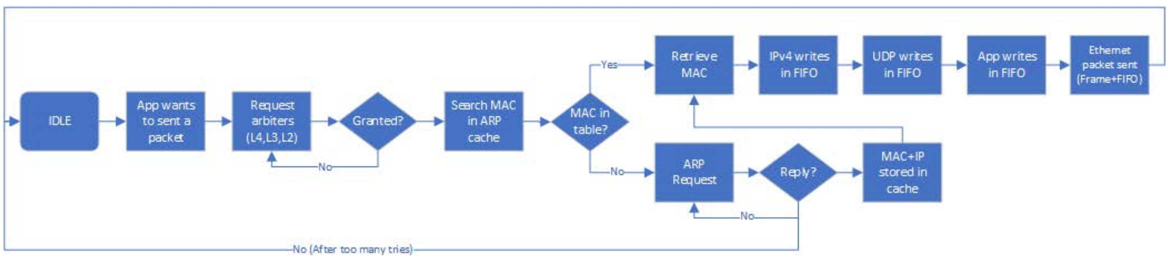


Figure 15. Flow diagram of the transmission of a datagram using the stack

In a typical full transmission cycle, an application layer service requires from a packet to be sent, this service starts sending a signal to all the arbiters, closing the way to the physical layer. The service will wait until all the arbiters are granted.

Once all arbiters are granted the system will take a look at the ARP cache, trying to find the destination IP, if the destination IP is found the related MAC will be picked, if it is not found the second layer arbiter will be changed to let way to the ARP protocol and an ARP request will be issued. Once the request has been replied with the MAC, this data will be saved inside the ARP cache, and it will be used to finish the service's message.

Having all the needed data the stack will start building the IPv4 part of the packet inside an output FIFO, then the UDP data will follow and finally the services data will be joined appended. Once we have the entire payload inside the FIFO, the Ethernet Tx block will start to send the message the physical layer through the RGMII_out interface. The Ethernet block will send the first part of the frame, then it will output all the FIFO content and at the end, it will finish the Ethernet frame. At this point, the message has been correctly sent.

3.4.1. Bus Arbiters

The designed system is formed by three arbiters that manage the access of top layer protocols to the next lower layer. Due to its simplicity, easy implementation and reliability it has been decided to use arbiters with a modified round-robin architecture.

Upper layer protocols will be given access to the bottom resource on arrival order, if multiple protocols request at the same time they will be served access in a circular order without any kind of priority, we are allowed multiple request, but only a single grant.

Arbiters that have granted access will be locked to the requester until they are released by the *free_bus* signal. The arbiters are lock during all the transmission process. When the arbiters are freed, the next queued service will be granted access.

The arbiter has a main input and output ports to transmit packet data (8 bits) and two auxiliary ports (input and output), these ports are configurable and they work on parallel to the main port, they are used to arbitrate extra signals between layers.

The arbiters are completely customizable in size. Using the variable CNT, we can decide the number of top layer devices (services) that we are going to connect at the inputs, all ports will adjust its size to this value. With the variable CNT_AUX we can decide the number of additional bits that we want in the auxiliary ports.

Table VI. List of the system arbiters and the protocols they have at the input and output.

Arbiter Layers	Input protocols	Output protocol	CNT	CNT_AUX
(L5-L4) Application-Transport	N Services, DHCP	UDP	N+1	83
(L4-L3) Transport-Network	UDP, ICMP	IPv4	2	3
(L3-L2) Network-Data Link	ARP, IPv4	Ethernet	2	3

3.4.2. RGMii Tx interface

This Figure 16 shows the hardware diagram for the transmit interface between the FPGA (MAC) and the physical chip:

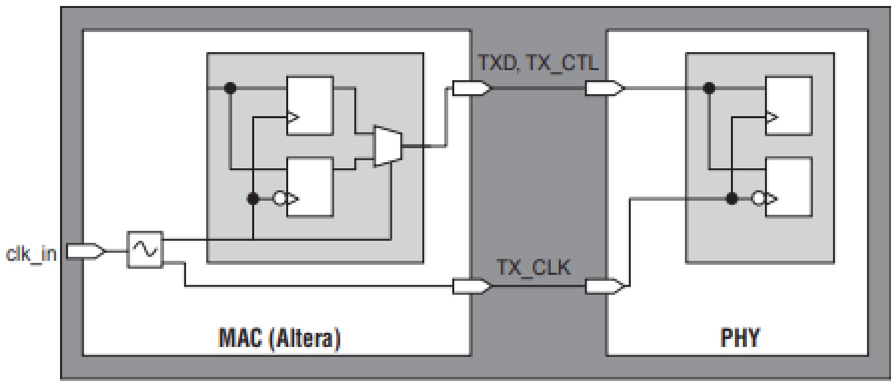


Figure 16. Block diagram of the transmit interface [5].

As we can see in Figure 16 the design is formed by a multiplexor with two inputs. This multiplexor will give pass to one set of signals in the rising edges of the clock, and to the other set in the falling edges, generating the DDR.

As we did in the reception side in order to generate this DDR signal, we will use the Intel's double data/output (DDIO) I/O register [26]. In this case we are going to use the ALTDDIO_OUT core, as we want to output a DDR signal from two inputs. These blocks give us a single clock and two inputs, one for the rising edge and another for the falling edge, and a DDR output. Two ALTDDIO_OUT cores have been necessaries, one for the TXD signal, and the other one for the TX_CTL signal.

As we have already explained in section 3.3.1 it is necessary a delay in the RGMII PHY clock signal regarding the RGMII MAC clock. Contrary to the RX case, this time the TX internal delay of the RGMII physical chip is deactivated by the board used in the implementation.

With the internal delay disabled, we have to be sure that the data and clock waveforms are center-aligned as seen in Figure 17.

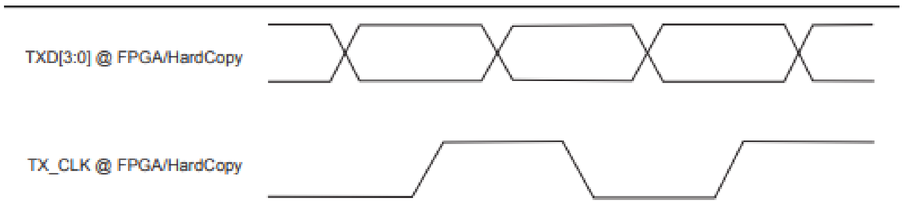


Figure 17. FPGA MAC generated signal TX_CLK with RGMII PHY internal delay disabled [5].

In order to center-align the clock signal to the data signal, we used a Phase-Locked Loop (PLL) [27]. We adjusted the PLL phases to generate two clock signals one with 0º phase and another one shifted 90º. The clock with 0º phase has been used in the ALTDDIO_OUT blocks that manage the data signals (TXD and TX_CTL). In the other hand, the 90º phase clock signal has been used as the output TX_CLK clock output to the RGMII PHY. To generate the PLL we have used another Intel core called ALTPLL. The final design obtained can be seen in Figure 18.

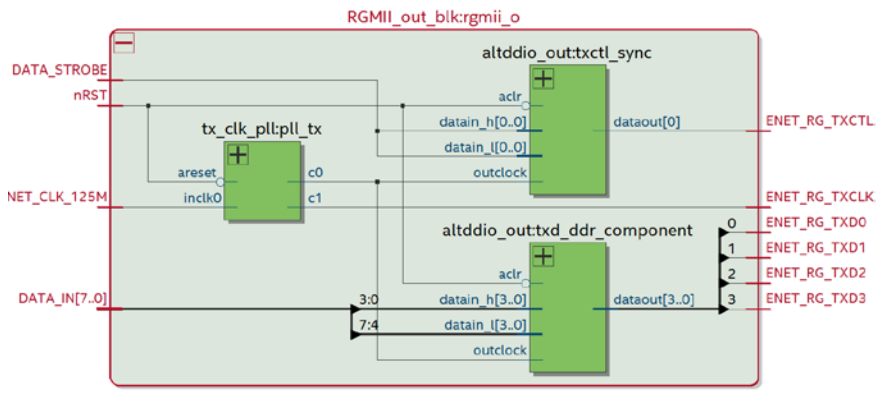


Figure 18. Final design of the RGMII TX block.

3.4.3. Ethernet Tx core

The Ethernet Tx frame block is in charge of generating the transmission sequence that is sent to the RGMII and then using wired Ethernet to the full network. The core contains an FSM control block, the CRC generator for the FCS and a FIFO to store the packet while it is being generated.

We are going to describe systematically the operation process followed by the core:

- The FSM starts in an IDLE state. The FSM will be activated upon receiving the flag END_TX, which is sent by the application layer when the service has ended the introduction of data inside the TX FIFO. At this point, the FIFO should have the entire payload inside, so we are ready to start the final step for transmission.
- After receiving the flag, we go to the state START. We will remain in this state until pkt_counter is higher than 0, this counter is increased by 1 each time a payload of data is ready to be send. It will be reduced by 1, each time we sent a packet.
- Then, we reach the PREAMBLE state, and 0x55h is sent for 7 cycles. The transmission is controlled by a counter that allow us time the states to a certain number of cycles (or bytes) as each cycle we sent a byte. From this state, until we reach the FCS state the CRC will start to use the data, we are transmitting to calculate its algorithm.
- In SOF we will send the start of the frame delimiter (0xD5, 1 byte/cycle). Then we will go to the next state.
- In the state MAC_TX we will send the destination MAC and the source MAC addresses. The macs are transmitted using a shift register with parallel load. After 12 cycles we will reach the next state.
- In ETYPE_MSB and ETYPE_LSB we will send both bytes of the Ethertype field. The Ethertype is determined by the combination of the arbiter granted signals (each combination is used by a different protocol). After that, we will reach the next state.
- Here we reach the PAYLOAD state. In this state we will start to transmit the content of the TX FIFO. We will transmit until the FIFO is empty or we receive an end of packet signal from the FIFO. If the payload is smaller than 46 bytes zero padding will be necessary (we use a counter to determine that), we will then go to the ZEROPAD state, if the payload is greater than 46, we will not need zero padding and we will go to the FCS state.

- In the ZEROPAD state we will send zeroes until we reach the correct payload size, then we will go to FCS state.
- In the FCS state we will output and transmit the calculated CRC value. Then we will go to the INTERPACKET state.
- This is the last state of the process, here we will free the arbiter buses, usually all of them, but only the layer 2 arbiter in the ARP case (as the ARP sometimes interrupts a service). We will stay in the interpacket state for the mandatory cycles and after that, we will return to the IDLE state, ready to transmit another packet.

3.4.4. IPv4 Tx core

The IPv4 Tx block is one of the main blocks of the stack. It is in charge of generating the IPv4 header, calculating the Tx Checksum and bypassing the required data from upper protocols to the FIFO as is the core directly connected to the Tx FIFO. It is also the first step in the transmission and therefore, the block is in charge of sending request to the ARP table and ARP protocol in general.

We are now going to describe the process that is followed during the IPv4 operation:

- A service makes a request for the needed arbiters, the arbiters grant the request. When the request is granted the service sets RUN_PKT to 1, this signal activates the IPv4 Tx FSM that changes state from IDLE to INPUT_DATA.
- In this state all, the important outside data is set inside the registers, in this state a distinction is made, if the IP is 0 (as it means we are still working with the DHCP) we won’t check the ARP tables, if we already have an IP the tables will be checked.
- In a normal case (no DHCP) the next step will be the state CHECK_RAM. Here a request is made to the ARP to check if we already have the MAC corresponding the IP where we want to send. If the MAC is available, we will go to the next state, if not we will send another request to the ARP protocol. This time we will ask for an ARP request message to obtain this MAC. After obtaining the MAC the system will proceed to the next state.
- In this new state CHECKSUM_IP we will pre-calculate part of the checksum, we will have to pre-calculate all the fields that are placed after the checksum, for the calculation we will use the component checksum16, this component is designed to calculate a 16-bit one complement checksum as is required by the protocol. After 7 cycles we will go to the next state.
- In the state VER_IHL the core will begin to form the IPv4 header, we will start writing to the Tx FIFO the header fields in order, meanwhile the checksum is being calculated in the background, and it will be delivered to when reached its field. The next FSM states work in the same way and are self-explanatory as they all have names of header fields. From VER_IHL we will go through the following states DSCP_ECN, TOTLEN, ID, FLAGS_OFFSET, TTL, PROTO, HCKSUM, SRC, DST and EOH.
- In EOH a signal flagging the end of the IP header will be set (MID_TX). This flag will activate the next protocol queued (for example UDP or ICMP). After that we will reach the protocol UPPER_L.

- In UPPER_L we wait for the other layers to finish its operation, once they finish a signal will be received PKT_FIFO_TX_EOP_IN, this signal will return the protocol to the beginning at IDLE.

3.4.5. UDP Tx core

The UDP Tx block is the one in charge of creating the UDP header for transmission. The operation of this block is as follows:

- When a service is granted all the arbiters, the FSM of UDP Tx goes from IDLE to BOTTOM_L.
- In the state BOTTOM_L, the block waits the lower layer protocol to finish its operation (IPv4 Tx). When IPv4 finish its operation, a flag called MID_TX_UDP is activated, and this will lead the FSM to the SCR_PORT_H state.
- From this state, SCR_PORT_H, the header starts to be formed and stored inside the FIFO; signals to the FIFO will go through the IPv4 block. Each state will last one cycle and deliver a specific byte to the FIFO. The FSM will go through SCR_PORT_H, SCR_PORT_L, DST_PORT_H, DST_PORT_L, TOT_LENGTH_H, TOT_LENGTH_L, HCKSUM_H, HCKSUM_L and TX_REST_MSG.
- In this last state (TX_REST_MSG) we will send a flag to the service with CALL_SERVICE, this will indicate the service that the header is completed, and the data is ready to be added to the FIFO. During this state, the service signal will go through the UDP block to reach the FIFO memory. We will stay in this state until FIFO_IN(8) is 1 (this signal means, “end of data”), then we will return to IDLE.

3.5. <u>Central timer</u>

The system requires the computation of real time in some operations like the timeout of stored ARP cache addresses or the DHCP lease timeout. In order to optimize the design a central timer has been added.

The central timer sets a flag every 125000 clock ticks (equivalent to a second). This signal is common in all the system, auxiliary timers can be then added and they can use the central timer flag to calculate its own passage of time.

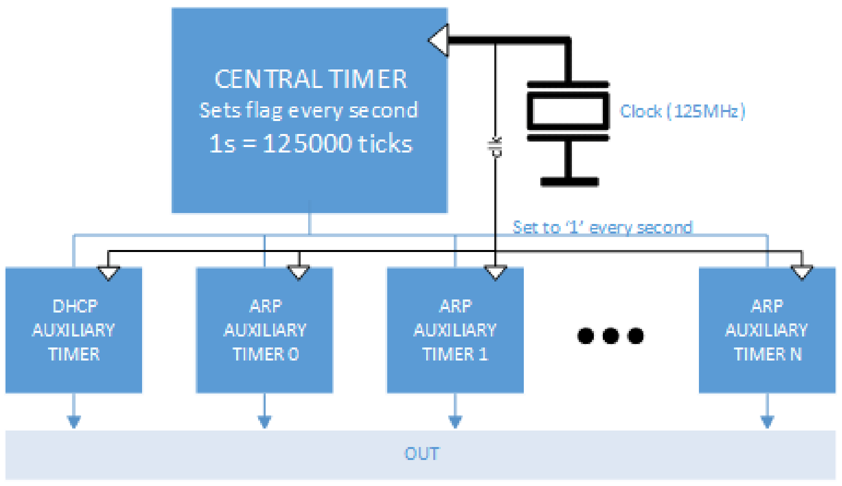


Figure 19. Block diagram of the designed system of timers

This method uses way less logic space than having each auxiliary timer calculate its own seconds, as this requires a very big counter for the 125000 ticks (17 bits). The final goal of this method is to reduce the logic resources required by adding extra timers to the system, as some blocks like the ARP cache have the possibility of increase its size. An increase with a large impact in space will limit a lot the system, and at the same time it will waste resources for more vital or critic components.

3.6. ARP protocol design

The ARP core has been designed to be able to do three main things: answer to an ARP request, send an ARP request and manage the ARP cache.

ARP request and ARP replies are managed using an FSM. The FSM (ARP_fsm) can interpret the operation codes of the messages we are receiving and it can send replies and request when needed. Setting flags, we can activate the block of RAM_check to search for a MAC at the ARP cache. The ARP cache has been designed as a Content-addressable Memory (CAM) [28] of customizable size.

Every time we receive and ARP request, we will answer with our own MAC address. We will send our own request when required by the stack. The request and cache system will work as specified in Figure 20.

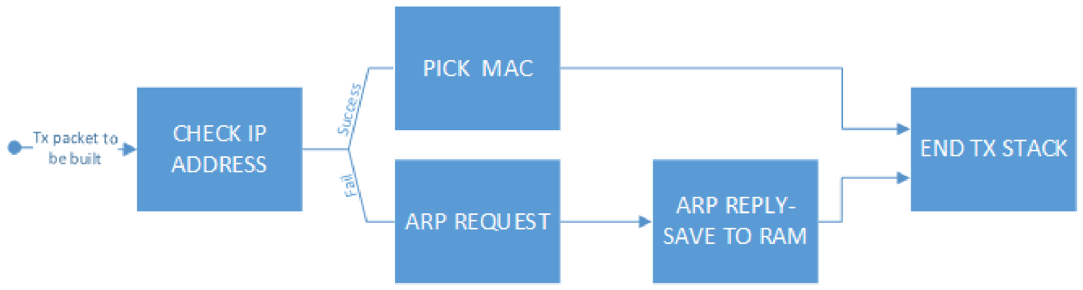


Figure 20. Flow diagram of the ARP checking cycle when a packet has to be sent

When a Service wants to send a packet to a certain IP, we will send a signal to the RAM_check block, there, we will try to find this specific IP inside the ARP cache RAM. If we find the IP, we will look at its related MAC and the stack will finish the service's message with the delivered MAC.

In the other hand, if are not able to find the IP inside the cache, an ARP Request is sent to that specific IP. When the device that holds that IP answers with its MAC (ARP reply), we are going to gather that data and save it inside the ARP cache. With the same data, we are going to finish and send the pending service's message.

An already written address will be marked as “full”. A “full” RAM address cannot be overwritten, and it will force the system to write an empty address. In an exceptional case, if all RAM addresses are full the system will be allowed to overwrite them.

This system does not have a way to read and interpret gratuitous ARP requests. If an IP is changed, the device has to tell the stack using alternative methods, the top layer services will be the ones in charge to manage that.

3.6.1. Auxiliary timers

An auxiliary timer is generated for each cache address; the size of the cache is customizable using the generics *RAM_size* and *RAM_width*. The division between them give us the number of RAM addresses. If we decide to modify the number of addresses, the number of auxiliary timers will change consistently. Timer's counter size can be set using the *timer_size* generic.

The ARP cache auxiliary timers are activated once a MAC+IP address have been written inside the RAM. The timer automatically starts to count until it reaches zero. When the timer finishes, it will set to '1' a timeout signal, and thus indicating the expiration of this specific cache address.

The ARP protocol will delete expired addresses when the protocol finishes an already started process. Deleted addresses cannot be recovered and the ARP protocol will ask for them again if needed. Deleted addresses are no longer marked as "full".

3.6.2. ARP cache design

The ARP cache has been implemented as CAM, this memory is also known as assistive memory. In a standard RAM memory, we give an address to the memory and it returns a set of data. In the other hand, a CAM memory compares an input tag (in our case an IP address) with a table of matching tags. If there is a match, the memory will output the matching data (the MAC address for us).

When a search input data is given to the CAM, it will search its entire memory looking for a match. If found the CAM will return either the address where you can find the data or the data itself. Our CAM memory has been designed over a RAM, the VHDL code added on top of the RAM make it behavior as a CAM that takes IP addresses as search items, and when found inside itself will return the linked MAC addresses.

3.6.3. ARP VHDL design

Now we are going to take a close look at the design of the ARP protocol operation. We will follow the steps of its FSM operations like in previous cases, the ARP protocol have two main FSM, one for receiving and transmitting data in (ARP_fsm) and another one for searching data and retrieving data of the cache, or writing to it in (RAM_check). This is the normal operation of the designed protocol:

- The protocol starts at the IDLE state, from here we have three ways, if we receive an ARP packet (ARP_REQUEST) we will go to the READ_MSG state to interpret this message, if we receive a MAC_REQUEST from the transmission side we will go to CHECK_RAM_IP to check the cache. Finally, if we are coming from an already failed cache fail (FAIL=1) we will go the TX_MSG state and make an arbiter request (we want to send an ARP Request to the unknown IP).
- From the state READ_MSG we will read the received ARP message and interpret it. If the message is an ARP Request (opcode is 1) we will go to TX_MSG and reply with our own MAC and IP. If we receive an unexpected reply (opcode 2 and FAIL=0) we will discard that packet, but if we were expecting the reply from a request (FAIL=1) we will set EN_ARP_REPLY, activating a process to save this info to the cache.

- In CHECK_RAM_IP we will activate EN_CHECK_IP to 1. This will enable a check for a MAC in the RAM_CHECK block. If the check is successful, we will return to IDLE, if not we will activate the FAIL flag, and proceed to TX_MSG to send an ARP request.
- TX_MSG is the start of the transmission of an ARP message. The ARP protocol may interrupt other protocols, so previous to perform an ARP Request we have to free the layer 2 arbiter (FREE_L2=1). For an ARP reply we will make a normal arbiter request and wait for the arbiter to be free. The following states will create the TX packet that will be stored in the TX FIFO (being a L2 protocol it is the only one that will be stored inside the TX FIFO). The following states create the packet fields: HTYPE_H, HTYPE_L, PTYPE_H, PTYPE_L, HLEN, PLEN, OP_CODE_H, OP_CODE_L, SRC_DST_MAC_IP and EOP. The signal op_code_reply will be the one in charge of distinguish if we are performing a Request and a Reply during this transmission process, and the packet will change depending on its value. If ARP_REQUEST is 1, op_code_reply will be 1 and we will perform a request. If MAC_REQUEST is 1 or FAIL is 1, the signal will be 0, and we will transmit a reply.
- When we finish a transmission of a packet we will return to IDLE if we have made a reply or to WAIT_REPLY if we have made a request.
- In WAIT_REPLY we will wait for a request. If the reply takes too long to arrive, we will raise a timeout signal and the request message will be sent again.

Until now we have only seen the operation process of the ARP Protocol FSM, all the management of the cache has been overlooked in favour of greater clarity. Now we are going to focus on the operation of the RAM_check block, which is the one in charge of managing the cache, looking for requested addresses, storing the new ones, and equally important, managing the timers of each address and annotate which addresses are full or empty:

- The RAM_check FSM starts in an IDLE state. In this state we will wait for a EN_CHECK_IP signal, which tells us that we are required to search for a specific IP in the cache. From IDLE we can also go to the AUX_TIMER state. We will reach this state when a timeout is detected in one of the timers, meaning that one of the addresses has expired and this address has to be emptied.
- In READ_RAM_IP we will proceed to read all the cache comparing its output to the wanted IP (INPUT_IP). We will use a counter (n_RAM_counter) to move over all the cache read addresses. If we have a match, we will set REQUEST_SUCCESS to ‘1’, if not we will set REQUEST FAILED instead. With a success we will output the result to external blocks with OUTPUT_MAC = q(79 downto 32) and go to the END_CHECK state. With a fail we will go to WAIT_REPLY instead.
- In END_CHECK we go directly to IDLE. This state has been created to let a cycle pass.
- When a check has been failed, we will reach WAIT_REPLY. Here we will wait for the ARP_fsm block to send a request and receive the reply, once we have gone the reply, we will set REQUEST_SUCCESS to 1, as we now have the correct IP and MAC. Once we have the requested IP and MAC addresses, we will enable the timer that check its expiration date, we will also write both addresses inside the cache

and mark the written address as “full”. We will then reach the FIND_EMPTY_ADDR state.

- In FIND_EMPTY_ADDR we find a new empty address for the next incoming addresses. full_addr signal has been created with the unique purpose of keeping track of that. If all the addresses are full, we will go to the AUX_TIMER state, if not we return to IDLE.
- The AUX_TIMER state can be reach with a timeout of an address timer in IDLE or by having all the addresses full in FIND_EMPTY_ADDRR. This state empty (write all zeroes) to all the timeout cache addresses, and it will reset this timeouts flags. If all the addresses are full, we allow the system to overwrite any of them until this situation is solved.

3.7. ICMP protocol design

The designed ICMP core is very simple. It only performs one of the huge amount of options that the protocol offers. As a diagnosis tool, the block will receive Ping request and answer them, this is called echo reply and it is the ICMP type 0 and code 0. The block does not perform more control messages and it does not send any kind of error message neither. Despite that, the protocol is easily modifiable and can be improved to interpret and use a large set of messages.

As in the previous cases the protocol is managed by an FSM. The block will receive incoming ICMP messages, it will then decode its type and code fields and if they match the ones of an Echo Request it will process to answer with an Echo Reply message. The core also calculated the ICMP checksum and discard packets with errors. The operation of the block is as follows:

- When a packet is multiplexed from the IP protocol (DATA_VALID_RX signal) the FSM will start its operation. We will move from the IDLE state to DATA_VALID.
- In DATA_VALID we will read the ICMP type, if it matches an Echo Request, we will continue, if not we will discard the packet. The checksum will start calculating itself.
- If the type is correct, the code state can be skipped, as an Echo Request only have 1 type of code. We will go through the CODE and CHECKSUM states/field and then reach REST_OF_THE_MSG.
- In REST_OF_THE_MSG we are dealing with the data field of the received ping, this data will be stored in a FIFO, as it has to be used later for the Echo Reply. We will use all the data to calculate the checksum, when we have stopped to receive data (DATA_VALID_RX = 0) we will then check if the checksum is correct. If it’s correct we will start a request of transmission to the arbiters and the Tx side, as we want to answer the ping (START_TX_ICMP = 1). If the checksum does not match, we will end here the operation of the FSM (not answer required) and the FIFO will be emptied.
- Once we are granted by the arbiters, we will start a normal flow of data transmission (BOTTOM_L state). The lower layer will generate and store its header, and then it will flag us (MID_TX_ICMP = 1) to store our data inside the Tx FIFO. Once flagged we will go to the TX_TYPE state.
- In TX_TYPE we will write the Echo_Reply type (0) inside the FIFO, next in TX_CODE the only reply code (also 0).

- After that we will write the checksum in two states TX_CHECKSUM_H and TX_CHECKSUM_L, the checksum has been calculated in a more efficient way, we will take the checksum of the Echo Request and subtract the type field while adding our new type. This can be done because the type is the only changed field in the ICMP data between these two messages.
- Finally, we will reach the TX_REST_MSG state. Here we will read the ICMP FIFO and write the ping data from the request as the data of our message. When the ICMP FIFO is emptied (FIFO_EMPTY = 1) we will signal the end of our packet with PKT_FIFO_EOP = 1 and the full process will end as we return to the IDLE state.

3.8. DHCP protocol design

The designed core follows the previously described operation. When the FPGA is turned on, the DHCP protocol will be the first to act. The DHCP protocol will activate if the system does not have an IP assigned, or if the IP lease has expired. Then, the core will send a discover message, if it is unanswered, the message will be repeated every certain time until a response is received, after receiving an offer, the same will happen with the IP request. Finally, after receiving the acknowledgement, the core will finish its job and it will do nothing until the IP lease expires (there is a timer set to calculate that). A NACK answer will force the design to request a new offer.

The system has been designed as two main FSMs: the first and main FSM (FSM0), and a second FSM (FSM1).

The first FSM (FSM0) controls the DHCP operation cycle, it will go from the IDLE state to DISCOVERY, OFFER, REQUEST and ACK states when reaching specific goals. We will leave the IDLE state if we don't have an assigned IP or if the assigned IP has been expired. The Discovery and Request states will end when the packet is formed, the Offer and ACK ones when a correct packet is received. If we don't receive an offer packet in while we will return to the Discovery state, if the same happen in the ACK state we will return to the Request state. If we receive a NACK packet the FSM will return to the Discovery state.

In parallel with this FSM we have set a second FSM (FSM1) that manages the data we receive and sends packets to the DHCP server. In the Discovery and Request states of FSM0, FSM1 will sent an arbiter request and will follow a typical transmission process, adding the DHCP data inside the Tx FIFO when required. At the end we will send a signal to FSM0 to go to the next state.

In the Offer and ACK states we will wait to receive a message to our UDP Port, when a message is received it will be decoded, and we will register the wanted data, like the XID code, Magic cookie, Source MAC address and the DHCP options like IP lease time, the subnet, the DHCP server and the most important one the offered IP. If the XID code does not match our own or the Magic cookie does not match the DHCP one, the packet will be discarded.

The block is designed to dynamically interpret the options of the DHCP messages. DHCP options are found at the end of the data packets, these options are formed by a Tag (1 byte), the data length, and the option data. A third and fourth FSM (Rx and TX FSM) have been created that are able to interpret the tags and length and read (or write) the needed data. The option tagged as hexadecimal FF sets the end of the BOOTP message and allows FSM1 to reach the next state.

The DHCP originally used a lot of resources due to the long string of bytes that it had to send that were stored in registers. In order to solve that problem a RAM was added to the DHCP. This RAM stores the options field array of bytes that has to be sent by the protocol during “Discovery” and “Request”. This array is pre-set in the RAM using a MapInfo Interchange Format file (.mif) [29]. The file writes the initial configuration of the RAM. As we can see this RAM is not read in any moment, so it could be exchanged by a ROM, a RAM was preferred in this case for future improvement of the design, in the future the modification of the DHCP options could be implemented, so in this case a RAM is the solution to go.

3.9. Design of Services

The stack has been designed to allow the attachment of an undetermined number of services to it. A service is a high layer application that does one specific function; it interacts with the stack by receiving its messages and/or sending messages to the network.

The receiving part can send data to any number of Rx services. Each service has a port number assigned to it and every time we receive a valid datagram, the UDP_RX block sends its destination port to all the active services. If the port number matches with the service port the message data will be diverted to the service.

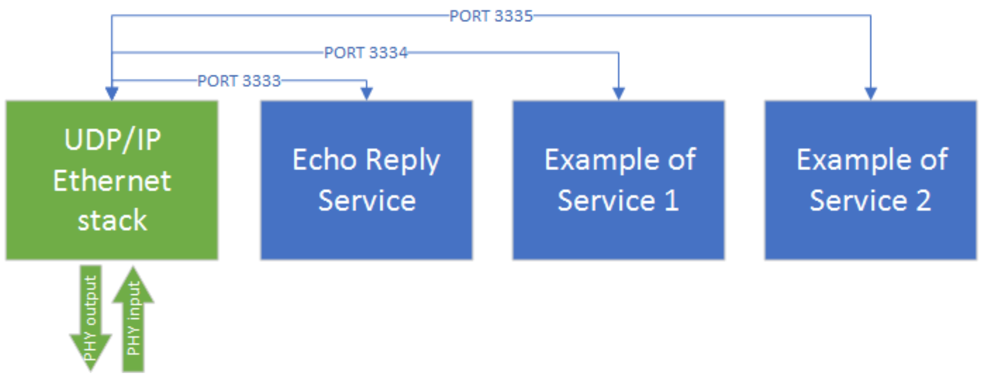


Figure 21. Block diagram of UDP/IP stack with implementation of three different services assigned to three different ports.

For the other side of the stack, the transmission part has an arbiter at its input, this arbiter can be modified to accept any number of inputs, this allows for any number of Tx Services there too.

Some services were designed for the stack as way of testing its correct performance and to test more complex operations that can be used in the future in a business or industrial environment were the stack can be used as a port of an FPGA design in a real system.

3.9.1. Echo reply service

The first service created was the Echo reply service, also called as Application 0 or App0. This service is very simple and its main purposes is the testing of the stack, it has also been created with the purpose of serving as an easy example template for all the stack services, as its functionality is very simple and it works with transmission and reception.

The service has been designed to receive UDP messages with any set of data attached to them. This data is copied and transmitted back to the original source as another UDP/IP message. If the original source receives, a message that is identical to the one initially sent it means that the stack is working properly without losing data.

The service consists on a very simple FSM:

- In the IDLE state it will wait for the UDP Destination Port to match our own port number, when done it will flag a DETECTED_PORT signal and wait for the UDP RX to finish its operation. Once finished we will go to the DETECTED state.
- In this next state, we will send a request to the arbiters and wait for the granted, once granted we will start a transmission (RUN_APP = 1), and go to the next state RUN_TX. In RUN_TX we will wait for the UDP TX block to flag us, thus allowing us to write our data inside the Tx FIFO.
- Once allowed, we will reach the READ_WRITE state, here we will write from the UDP FIFO and write the same data in the Tx FIFO (as we just want to Echo that data). When we finish doing that (as we get told by the UDP FIFO with DATA_in(8) = '1') we will go to the EOP from where we will tell the Ethernet TX block that we are done with FIFO_TX_EOP = 1.

3.9.2. DAQ interface service

As it was initially planned a DAQ interface has begun to be added as a stack's service. This DAQ hasn't been fully implemented yet but its design and creation have already been started and the service is already operating correctly. It is being implemented as a transmission service that receives data from an outside source (a real DAQ or during testing a FPGA counter), the received data is modified and then encapsulated inside as an UDP/IP Ethernet message that is sent through the stack.

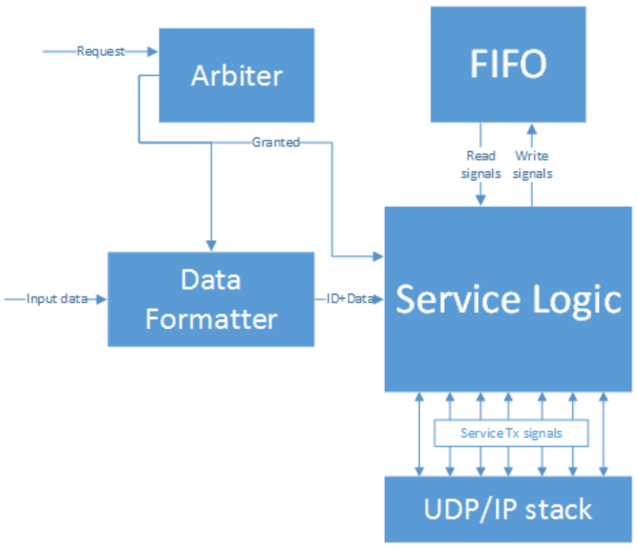


Figure 22. Block diagram designed DAQ interface service.

The DAQ service allow us to acquire and sent data from a variable number of external writers. As we can see in service. Figure 22 the structure of the DAQ is formed by four main blocks.

The system has and arbiter (round-robin) to set through the different writers connected to the DAQ interface. The data formatter reads the data from the input bus and knowing the arbiter's granted bus it extracts the writer ID and builds a frame with the ID and the incoming Data. Finally, it adds a padding of zeroes to match the frame length. These blocks store all the data into the FIFO.

The FIFO is a dual port RAM with it has programmable depth, and the width is selected during compilation (it is adjusted to the biggest writer).

The Service control logic is the DAQ's main core, and manages all the other blocks and the interactions with the communications stack. It generates all the control signals for the formatter block and the reading and writing of the FIFO. It also controls all the signals needed to work as a stack service.

3.9.3. Adding a new service to the stack

In order to add a service to the stack it was necessary the creation of high hierarchy entity, that would include the UDP/IP stack (called *cyclone10_eth*) and all the additional services that would work with it. This block is called *top_entity* and it can be used as a template in order to add new services to the stack. Inside the block the stack and all the services are declared as components, and then the necessary interconnections have been made between them.

Adding a new service to the stack is an easy process, there are many signals to take into account but the system has been designed to make the process very methodical. These are the steps to follow if we desire to add a new service:

1. Increasing SERVICES_TX generic variable by N (total is number of external services that are going to transmit data). If no transmission external services are required, the variable should be set to 0.
2. Increasing SERVICES_RX generic variable by N (total is number of external services that are going to receive data). If no receiving external services are required, the variable should be set to 0.
3. Add the required signals in order to communicate with the stack following the created guideline (Appendix C).

For the creation of a service the Echo Reply service is a good template and example for the creation of Rx and Tx services as it's a very simple, short and easy to understand (around 150 lines of code).

3.10. Clock constrains and timings

In this section we will talk about all the problems that have arisen around the projects timings and all the design decisions related to it.

3.10.1. Design Constrains

Three constrains files have been added to the project, the three of them are related to the RGMII clocks and timings. The first file created, named *rgmii_clocks.sdc* contains the declaration of all the clock domains, the 125MHz internal clock, the two transmission PLL clocks TXCLK_00 and TXCLK_90 and the received clock from the physical media RXCLK.

In the files *rgmii_input.sdc* and *rgmii_output.sdc* are set the input and output delays and some false path and multicycle path are defined, clock uncertainty is added. This improves the timing simulation of the project and it eliminates some false warnings that appeared in the compiler.

3.10.2. Timing violations

The design has very long combinational paths that go all over the protocols, in some cases timing violations have been an issue in the design. In order to solve them, some paths have been split in two. The split of path has been performed by pipelining the signals in critical steps of the path, were it was more convenient to do it.

Using timing tools, the worst-case path was found, and some of that signals were pipelined to avoid future timing problems. The latch of signals makes necessary to readjust the interactions between protocols, which can be a problem if issues arise later.

4. Results

Once the system started to be designed in VHDL it was necessary the implementation of the blocks to a real FPGA system in order to test the operation of all parts of the design. In this section we will first focus in the hardware and software tools used for the design, implementation and testing of the system, then we will analyze the generated timing and resource reports, and finally we will talk about the final performance test made to the system in order to check the correct operation of the full design.

4.1. Hardware and Software tools

4.1.1. FPGA board

The FPGA used for the implementation of the hardware design has been the Intel Cyclone 10 LP 10CL025 [30]. The FPGA model has been selected by ICCUB, as it is the most used in its own projects in conjunction with the Cyclone III that is nowadays discontinued. The design should be easy to translate to other FPGA models.

The Cyclone 10 LP family is optimized for high-bandwidth, low power consumption and low-cost. They are built on 60nm. The used FPGA the 10CL025 has the following characteristics:

Table VII. FPGA Cyclone 10 hardware characteristics

Cyclone 10 LP 10CL025	
Logic elements (LEs) (K)	25
Memory blocks (9K)	66
Memory blocks (Kb)	594
18x18 Multipliers	66
Phase-locked loop (PLL)	4
Global clock networks	20
LVDS channels	52
Maximum I/O	150

The used model is one of the low tier ones, but it is good enough to deal with complex designs without problems of space of resources. The implementation has been performed in top of a development board, the Cyclone 10 LP FPGA Evaluation Kit.

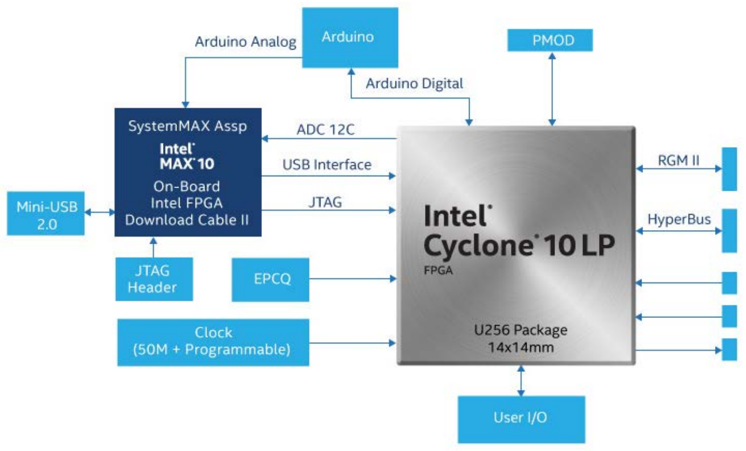


Figure 23. Intel Cyclone 10 LP FPGA Evaluation board, block diagram [30].

This kit allows an easy and fast way to test designs on a Cyclone 10 LP FPGA and it is also compatible with Arduino UNO shields, PMOD, GPIO and Ethernet. As we were designing an Ethernet stack, the most interesting characteristics for us were the ones regarding this:

- 10/100/1000Mbps Ethernet
    - Intel XWAY PHY11G PEF7071
    - RGMII Interface to Intel Cyclone 10 LP FPGA (as MAC)
    - MDC/MDIO as management interface
    - RJ-45 for Ethernet Cable

Other interesting characteristics are the 125 MHz Ethernet Clock to FPGA and the USB port that can be used for powering up the board. In Figure 23 we can take a look at the block diagram of the board and see other available contents of the kit.

4.1.2. Software tools

The main software tool used for designed VHDL hardware has been Quartus Prime Software. This is the software provided by Intel to design hardware for FPGA and SoC, for a large array of systems like Intel Stratix, Arria or the Cyclone family.

Quartus Prime software (previously called Quartus II) allows the analysis and synthesis of HDL designs. It has compilation tools and the ability to perform timing analysis and RTL diagrams. It also includes simulation and debugging tools. All of this can be implemented in different hardware description languages like VHDL, Verilog or AHDL.

The design was difficult to simulate, as the creation of a communications stack needs very complex inputs that give us very complex outputs. This complexity highlights the need of use alternative debugging and testing methods. Most of the debugging of the design has been made then with tools that allowed the system to be debugged and tested in real time, with real inputs being sent using packet sender software. The following list of tools has been used inside the Quartus Prime Software:

- **Signal Tap Logic Analyzer**: Signal Tap Logic Analyzer is a very easy but powerful tool that allows the designer to capture signals from the internal nodes of the FPGA while the device is running. Summarizing, this tool gives us a non-intrusive way of monitoring internal signals of the design. Signal Tap allows the creation of very specific triggers that makes the whole process much easier.

- **In-System Sources and Probes**: This tool allows the designer to read and write data from/to a running design using the JTAG interface. Input and output ports are created as sources (write) and probes (read). This is very useful for debugging the designs as we are allowed to read internal nodes of the FPGA and to change internal signals while running our design.
- **RTL Viewer and Technology Map Viewer:** The RTL Viewer shows us the digital logic implemented in the HDL code; the technology map shows us the implementation that Quartus has made inside the FPGA. Both tools are very useful for optimization of the HDL code and debugging.
- **TimeQuest Timing Analyzer:** TimeQuest is timing analysis software, it can make timing reports of a compiled design and then it gives the user tools in order to solve the possible timing problems of the design. This software is critical to find and solve timing violations, as are impossible to find with other ways, and very difficult to debug without specialized software. TimeQuest allows the user to implement time constraints and other changes without the need to rerun the compilation, making the process faster and easier.

We have also used some third-party software to help us with the testing and debugging:

- **Wireshark:** Wireshark is and open source packet analyzer which allows to capture all the packets in a specific network and to easily filter them and extract information about them. This software has been used to capture the communications between the FPGA, the testing PC and the DHCP server, capturing the messages that have been sent between them and checking that no information has been lost, damaged or malformed.
- **Packet sender:** This software is a very simple program that allows the user to send UDP, TCP and SSL packets to a specific IP address and Ports. This software has been used from the Testing PC as a way to send test packets to the FPGA. The program incorporates a log panel that receives the answers from the FPGA, but Wireshark is more efficient in that.

4.2. Timing reports

Timing reports have been generated with TimeQuest Timing Analyzer. The clocks analyzed are the following ones, each one of them has been analyzed with three different models of the FPGA:

- ENET_CLK_125M (CLK125)
- ENET_RG_RXCLK (CLKRX)
- ENET_RG_TXCLK_00 (CLK00)
- ENET_RG_TXCLK_90 (CLK90)
- altera_reserved_tck (CLKA)

The timings reports are only focused on the UDP/IP Ethernet stack, external services are excluded for this analysis, these are the obtained results:

Table VIII. Timing reports for model (slow, 1200mV, 85ºC)

Slow 1200mV 85C Model					
	Setup (ns)	Hold (ns)	Recovery (ns)	Removal (ns)	MPWS (ns)
CLK125	0.103	0.399	4.304	1.817	3.503
CLKRX	0.696	0.413	4.359	2.104	3.574
CLK00	1.205	4.424	X	X	3.519
CLK90	X	X	X	X	3.519
CLKA	43.419	0.453	95.810	1.440	49.436

Table IX. Timing reports for model (slow, 1200mV, 0ºC)

Slow 1200mV 0C Model					
	Setup (ns)	Hold (ns)	Recovery (ns)	Removal (ns)	MPWS (ns)
CLK125	0.664	0.382	4.507	1.619	3.426
CLKRX	0.858	0.395	4.554	1.872	3.581
CLK00	1.774	3.913	X	X	3.519
CLK90	X	X	X	X	3.519
CLKA	43.877	0.402	96.141	1.295	49.289

Table X. Timing reports for model (fast, 1200mV, 0ºC)

Fast 1200mV 0C Model					
	Setup (ns)	Hold (ns)	Recovery (ns)	Removal (ns)	MPWS (ns)
CLK125	4.584	0.132	6.276	0.736	3.179
CLKRX	1.855	0.089	6.313	0.890	3.179
CLK00	4.663	2.006	X	X	3.811
CLK90	X	X	X	X	3.971
CLKA	47.234	0.186	98.014	0.589	49.442

4.3. Resource usage

Table XI. Resource usage report summary of the UDP/IP Ethernet stack

Fitter Resource Usage Summary	
Resource	**Usage**
Total logic elements	3,691 / 24,624 (15 %)
-- Combinational with no register	1328
-- Register only	462
-- Combinational with a register	1901
Logic element usage by number of LUT inputs	
-- 4 input functions	1529
-- 3 input functions	783
-- <=2 input functions	917
-- Register only	462
Logic elements by mode	
-- normal mode	2658
-- arithmetic mode	571
Total registers*	2,373 / 25,304 (9 %)
-- Dedicated logic registers	2,363 / 24,624 (10 %)
-- I/O registers	10 / 680 (1 %)
Total LABs: partially or completely used	277 / 1,539 (18 %)
Virtual pins	0
I/O pins	33 / 151 (22 %)
-- Clock pins	2 / 8 (25 %)
-- Dedicated input pins	0 / 9 (0 %)
M9Ks	14 / 66 (21 %)
Total block memory bits	67,840 / 608,256 (11 %)
Total block memory implementation bits	129,024 / 608,256 (21 %)
Embedded Multiplier 9-bit elements	0 / 132 (0 %)
PLLs	1 / 4 (25 %)
Global signals	5
-- Global clocks	5 / 20 (25 %)
JTAGs	0 / 1 (0 %)
CRC blocks	0 / 1 (0 %)
ASMI blocks	0 / 1 (0 %)
Oscillator blocks	0 / 1 (0 %)
Impedance control blocks	0 / 4 (0 %)
Average interconnect usage (total/H/V)	3.9% / 3.9% / 3.9%
Peak interconnect usage (total/H/V)	20.1% / 19.8% / 20.6%
Maximum fan-out	2099
Highest non-global fan-out	1884
Total fan-out	20374
Average fan-out	3.34

* Register count does not include registers inside RAM blocks or DSP blocks.

Table XII. Resource usage report summary of the UDP/IP Ethernet stack with DAQ and Echo Reply services

Fitter Resource Usage Summary	
Resource	**Usage**
Total logic elements	4,310 / 24,624 (18 %)
-- Combinational with no register	1546
-- Register only	506
-- Combinational with a register	2258
Logic element usage by number of LUT inputs	
-- 4 input functions	1765
-- 3 input functions	938
-- <=2 input functions	1101
-- Register only	506
Logic elements by mode	
-- normal mode	3134
-- arithmetic mode	670
Total registers*	2,774 / 25,304 (11 %)
-- Dedicated logic registers	2,764 / 24,624 (11 %)
-- I/O registers	10 / 680 (1 %)
Total LABs: partially or completely used	323 / 1,539 (21 %)
Virtual pins	0
I/O pins	33 / 151 (22 %)
-- Clock pins	2 / 8 (25 %)
-- Dedicated input pins	3 / 9 (33 %)
M9Ks	15 / 66 (23 %)
Total block memory bits	68,352 / 608,256 (11 %)
Total block memory implementation bits	138,240 / 608,256 (23 %)
Embedded Multiplier 9-bit elements	0 / 132 (0 %)
PLLs	1 / 4 (25 %)
Global signals	7
-- Global clocks	7 / 20 (35 %)
JTAGs	1 / 1 (100 %)
CRC blocks	0 / 1 (0 %)
ASMI blocks	0 / 1 (0 %)
Oscillator blocks	0 / 1 (0 %)
Impedance control blocks	0 / 4 (0 %)
Average interconnect usage (total/H/V)	4.4% / 4.3% / 4.5%
Peak interconnect usage (total/H/V)	17.9% / 17.7% / 19.2%
Maximum fan-out	2412
Highest non-global fan-out	2050
Total fan-out	23603
Average fan-out	3.31

* Register count does not include registers inside RAM blocks or DSP blocks.

4.4. Board implementation and testing

We have performed a series of test on the full UDP/IPv4 stack with the objective of check the correct performance of all the protocols involved. The test has the job of checking the correct performance of the following parts of the core:

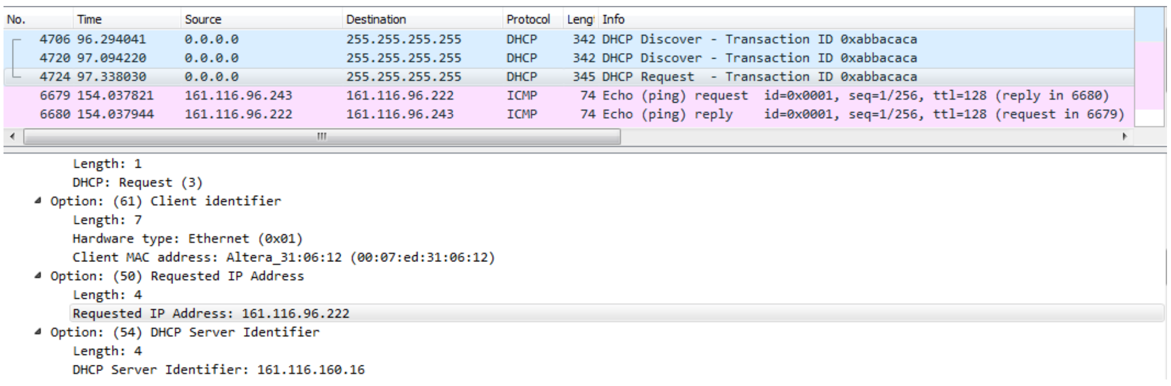
- DHCP discover of server and request of IP.
- ARP replies and request, use of ARP cache.
- UDP/IP reception of packets and filtering of wrong MACs, IPs and Ports.
- UDP/IP transmission of packets.
- Use of an application protocol.
- Ping from a computer.
- DAQ Interface.

The test has been performed using a switch between the FPGA board and the network router. This switch is connected to the FPGA device, the router and a PC that has been used to send and receive packets from the FPGA.

a) DHCP – Request of IP

When the FPGA is turned on the DHCP protocol is the first to act. The DHCP protocol will activate if the system does not have an IP assigned or if the IP lease has expired. First, the protocol is going to send a BOOTP message for the discovery of DHCP servers, the server then is expected to answer with an IP offer. The offer will contain an IP, we will take this IP and broadcast a Request with it, finally if the server is fine with the request it will send a final ACK answer. After that, we are expected to keep the IP address during an IP lease time that is set inside the ACK message. This time is usually of 24 hours.

Turning on our FPGA we are expected to see first Discovery message and a Request after that, as we are using a switch between our device and the router, the answers of the server are hidden so they will not show in Wireshark. If everything has been done correctly, we will be able to send a Ping after the DHCP assignations.



No.	Time	Source	Destination	Protocol	Leng	Info
4706	96.294041	0.0.0.0	255.255.255.255	DHCP	342	DHCP Discover - Transaction ID 0xabbacaca
4720	97.094220	0.0.0.0	255.255.255.255	DHCP	342	DHCP Discover - Transaction ID 0xabbacaca
4724	97.338030	0.0.0.0	255.255.255.255	DHCP	345	DHCP Request - Transaction ID 0xabbacaca
6679	154.037821	161.116.96.243	161.116.96.222	ICMP	74	Echo (ping) request id=0x0001, seq=1/256, ttl=128 (reply in 6680)
6680	154.037944	161.116.96.222	161.116.96.243	ICMP	74	Echo (ping) reply id=0x0001, seq=1/256, ttl=128 (request in 6679)

```
        Length: 1
        DHCP: Request (3)
    ◢ Option: (61) Client identifier
        Length: 7
        Hardware type: Ethernet (0x01)
        Client MAC address: Altera_31:06:12 (00:07:ed:31:06:12)
    ◢ Option: (50) Requested IP Address
        Length: 4
        Requested IP Address: 161.116.96.222
    ◢ Option: (54) DHCP Server Identifier
        Length: 4
        DHCP Server Identifier: 161.116.160.16
```

Figure 24. DHCP session and ping request to test acknowledgment of address

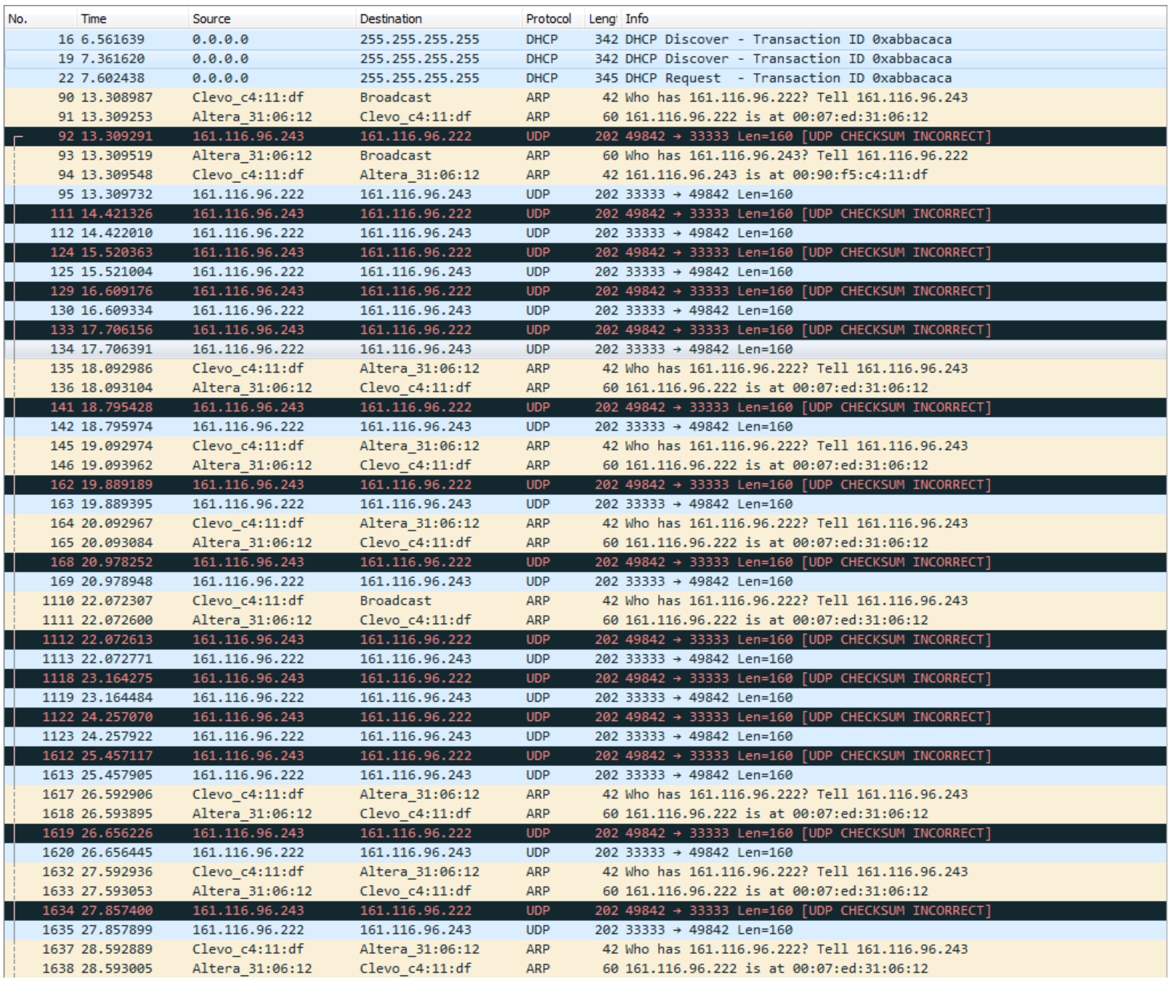
The obtained packets are the ones we were expecting, the FPGA has send two discover packets because the offer reply was taking too long, after the second discovery request we have received and offer that has triggered a Request of the offered IP 161.116.96.222. We try then to send a ping to the IP to test if the acknowledge has sent, and in fact, it has, as we can see the ping is correctly answered.

b) Application layer (and ARP) - Echo of a message

This is the main test that will be performed, and will help to check the Rx and Tx UDP/IPv4 protocols and the ARP replies, replies and use of its cache. The test will consist on the incorporation of an Echo Service at the top layer (application). The Echo block will receive all the valid packets that we sent to him (port 3333) and will create an answer with the same data.

It is expected that before the first message is sent, we will receive an ARP request that will be correctly answer. Before we sent our first Echo reply, it is also expected that the device will check the ARP cache trying to find the MAC of the destination device; once it fails (the list starts empty), it will send an ARP Request and save the answer to the list. After that, all the next messages will be performed without more ARP requests.

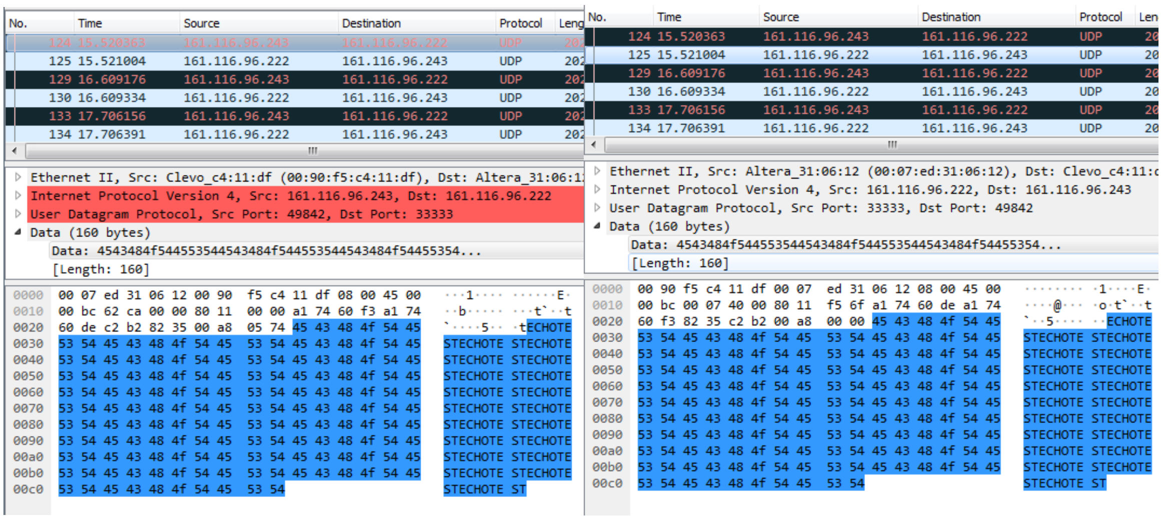
We will send echo request with 1-second resend delay between them; these are the results:



No.	Time	Source	Destination	Protocol	Leng	Info
16	6.561639	0.0.0.0	255.255.255.255	DHCP	342	DHCP Discover - Transaction ID 0xabbacaca
19	7.361620	0.0.0.0	255.255.255.255	DHCP	342	DHCP Discover - Transaction ID 0xabbacaca
22	7.602438	0.0.0.0	255.255.255.255	DHCP	345	DHCP Request - Transaction ID 0xabbacaca
90	13.308987	Clevo_c4:11:df	Broadcast	ARP	42	Who has 161.116.96.222? Tell 161.116.96.243
91	13.309253	Altera_31:06:12	Clevo_c4:11:df	ARP	60	161.116.96.222 is at 00:07:ed:31:06:12
92	13.309291	161.116.96.243	161.116.96.222	UDP	202	49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
93	13.309519	Altera_31:06:12	Broadcast	ARP	60	Who has 161.116.96.243? Tell 161.116.96.222
94	13.309548	Clevo_c4:11:df	Altera_31:06:12	ARP	42	161.116.96.243 is at 00:90:f5:c4:11:df
95	13.309732	161.116.96.222	161.116.96.243	UDP	202	33333 → 49842 Len=160
111	14.421326	161.116.96.243	161.116.96.222	UDP	202	49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
112	14.422010	161.116.96.222	161.116.96.243	UDP	202	33333 → 49842 Len=160
124	15.520363	161.116.96.243	161.116.96.222	UDP	202	49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
125	15.521004	161.116.96.222	161.116.96.243	UDP	202	33333 → 49842 Len=160
129	16.609176	161.116.96.243	161.116.96.222	UDP	202	49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
130	16.609334	161.116.96.222	161.116.96.243	UDP	202	33333 → 49842 Len=160
133	17.706156	161.116.96.243	161.116.96.222	UDP	202	49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
134	17.706391	161.116.96.222	161.116.96.243	UDP	202	33333 → 49842 Len=160
135	18.092986	Clevo_c4:11:df	Altera_31:06:12	ARP	42	Who has 161.116.96.222? Tell 161.116.96.243
136	18.093104	Altera_31:06:12	Clevo_c4:11:df	ARP	60	161.116.96.222 is at 00:07:ed:31:06:12
141	18.795428	161.116.96.243	161.116.96.222	UDP	202	49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
142	18.795974	161.116.96.222	161.116.96.243	UDP	202	33333 → 49842 Len=160
145	19.092974	Clevo_c4:11:df	Altera_31:06:12	ARP	42	Who has 161.116.96.222? Tell 161.116.96.243
146	19.093962	Altera_31:06:12	Clevo_c4:11:df	ARP	60	161.116.96.222 is at 00:07:ed:31:06:12
162	19.889189	161.116.96.243	161.116.96.222	UDP	202	49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
163	19.889395	161.116.96.222	161.116.96.243	UDP	202	33333 → 49842 Len=160
164	20.092967	Clevo_c4:11:df	Altera_31:06:12	ARP	42	Who has 161.116.96.222? Tell 161.116.96.243
165	20.093084	Altera_31:06:12	Clevo_c4:11:df	ARP	60	161.116.96.222 is at 00:07:ed:31:06:12
168	20.978252	161.116.96.243	161.116.96.222	UDP	202	49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
169	20.978948	161.116.96.222	161.116.96.243	UDP	202	33333 → 49842 Len=160
1110	22.072307	Clevo_c4:11:df	Broadcast	ARP	42	Who has 161.116.96.222? Tell 161.116.96.243
1111	22.072600	Altera_31:06:12	Clevo_c4:11:df	ARP	60	161.116.96.222 is at 00:07:ed:31:06:12
1112	22.072613	161.116.96.243	161.116.96.222	UDP	202	49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
1113	22.072771	161.116.96.222	161.116.96.243	UDP	202	33333 → 49842 Len=160
1118	23.164275	161.116.96.243	161.116.96.222	UDP	202	49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
1119	23.164484	161.116.96.222	161.116.96.243	UDP	202	33333 → 49842 Len=160
1122	24.257070	161.116.96.243	161.116.96.222	UDP	202	49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
1123	24.257922	161.116.96.222	161.116.96.243	UDP	202	33333 → 49842 Len=160
1612	25.457117	161.116.96.243	161.116.96.222	UDP	202	49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
1613	25.457905	161.116.96.222	161.116.96.243	UDP	202	33333 → 49842 Len=160
1617	26.592906	Clevo_c4:11:df	Altera_31:06:12	ARP	42	Who has 161.116.96.222? Tell 161.116.96.243
1618	26.593895	Altera_31:06:12	Clevo_c4:11:df	ARP	60	161.116.96.222 is at 00:07:ed:31:06:12
1619	26.656226	161.116.96.243	161.116.96.222	UDP	202	49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
1620	26.656445	161.116.96.222	161.116.96.243	UDP	202	33333 → 49842 Len=160
1632	27.592936	Clevo_c4:11:df	Altera_31:06:12	ARP	42	Who has 161.116.96.222? Tell 161.116.96.243
1633	27.593053	Altera_31:06:12	Clevo_c4:11:df	ARP	60	161.116.96.222 is at 00:07:ed:31:06:12
1634	27.857400	161.116.96.243	161.116.96.222	UDP	202	49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
1635	27.857899	161.116.96.222	161.116.96.243	UDP	202	33333 → 49842 Len=160
1637	28.592889	Clevo_c4:11:df	Altera_31:06:12	ARP	42	Who has 161.116.96.222? Tell 161.116.96.243
1638	28.593005	Altera_31:06:12	Clevo_c4:11:df	ARP	60	161.116.96.222 is at 00:07:ed:31:06:12

Figure 25. Full stack test

As we were expecting, after the DHCP requests at the beginning we receive an ARP request at line 90, the request is answered successfully at line 91. After that, the first Echo request is received at line 92, as we do not have the destination MAC of the PC; we perform an ARP request at line 93, answered at 94. Finally, with the MAC saved, we sent the echo answer at line 95. After that, the rest of the Echo request are answered without more ARP intervention in the device part, as we already know the MAC related to that IP. The PC keeps asking for our MAC with ARP request between messages all of them are correctly answered. In the next figure we can see that the data sent is the same as the data received, thus the Echo is correctly working too:



```
No.  Time       Source          Destination     Protocol  Leng
 124 15.520363  161.116.96.243  161.116.96.222  UDP       202
 125 15.521004  161.116.96.222  161.116.96.243  UDP       202
 129 16.609176  161.116.96.243  161.116.96.222  UDP       202
 130 16.609334  161.116.96.222  161.116.96.243  UDP       202
 133 17.706156  161.116.96.243  161.116.96.222  UDP       202
 134 17.706391  161.116.96.222  161.116.96.243  UDP       202

▷ Ethernet II, Src: Clevo_c4:11:df (00:90:f5:c4:11:df), Dst: Altera_31:06:1
▷ Internet Protocol Version 4, Src: 161.116.96.243, Dst: 161.116.96.222
▷ User Datagram Protocol, Src Port: 49842, Dst Port: 33333
◢ Data (160 bytes)
    Data: 4543484f544553544543484f544553544543484f54455354...
    [Length: 160]

0000  00 07 ed 31 06 12 00 90  f5 c4 11 df 08 00 45 00   ···1···· ······E·
0010  00 bc 62 ca 00 00 80 11  00 00 a1 74 60 f3 a1 74   ··b····· ···t`··t
0020  60 de c2 b2 82 35 00 a8  05 74 45 43 48 4f 54 45   `····5·· ·tECHOTE
0030  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
0040  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
0050  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
0060  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
0070  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
0080  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
0090  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
00a0  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
00b0  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
00c0  53 54 45 43 48 4f 54 45  53 54                     STECHOTE ST
```

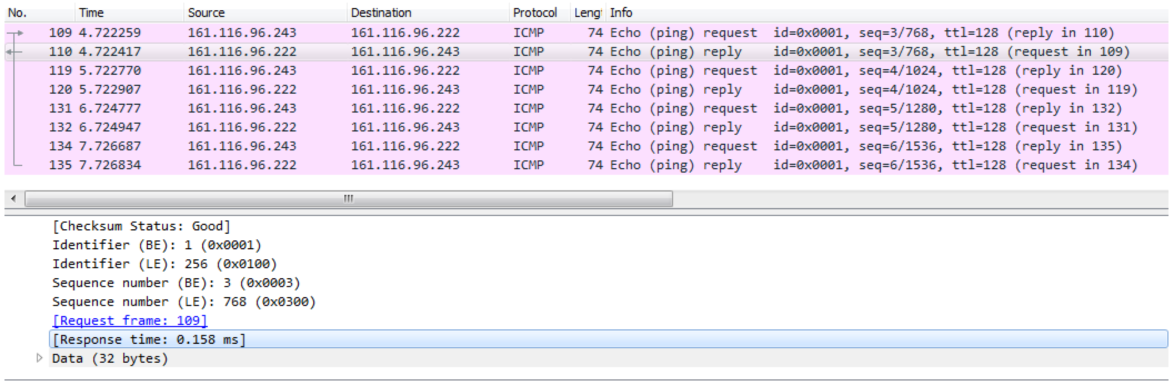
```
No.  Time       Source          Destination     Protocol  Len
 124 15.520363  161.116.96.243  161.116.96.222  UDP       20
 125 15.521004  161.116.96.222  161.116.96.243  UDP       20
 129 16.609176  161.116.96.243  161.116.96.222  UDP       20
 130 16.609334  161.116.96.222  161.116.96.243  UDP       20
 133 17.706156  161.116.96.243  161.116.96.222  UDP       20
 134 17.706391  161.116.96.222  161.116.96.243  UDP       20

▷ Ethernet II, Src: Altera_31:06:12 (00:07:ed:31:06:12), Dst: Clevo_c4:11:c
▷ Internet Protocol Version 4, Src: 161.116.96.222, Dst: 161.116.96.243
▷ User Datagram Protocol, Src Port: 33333, Dst Port: 49842
◢ Data (160 bytes)
    Data: 4543484f544553544543484f544553544543484f54455354...
    [Length: 160]

0000  00 90 f5 c4 11 df 00 07  ed 31 06 12 08 00 45 00   ········ ·1····E·
0010  00 bc 00 07 40 00 80 11  f5 6f a1 74 60 de a1 74   ····@··· ·o·t`··t
0020  60 f3 82 35 c2 b2 00 a8  00 00 45 43 48 4f 54 45   `··5···· ··ECHOTE
0030  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
0040  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
0050  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
0060  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
0070  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
0080  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
0090  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
00a0  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
00b0  53 54 45 43 48 4f 54 45  53 54 45 43 48 4f 54 45   STECHOTE STECHOTE
00c0  53 54 45 43 48 4f 54 45  53 54                     STECHOTE ST
```

Figure 26. Echo request (left) and Echo reply (right) – The data is correctly echoed.

c) ICMP test – Ping

The stack has been given of a simple ICMP protocol block. The ICMP protocol can be designed to provide many features from control of the network to error notifications or other operational information. The designed ICMP block only has the Ping feature.

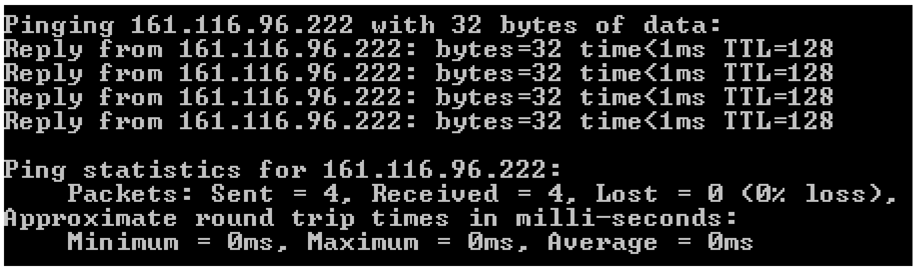
After the IP has being acknowledge, we will try to send a Ping request to the FPGA. We will send the Ping requests from a Windows PC, first we will send a single Ping, and after that, we will repeat the test, sending request during a full minute; we are expected to receive answers on most of the requests. These are the results obtained on the single ping test:



```
No.  Time      Source          Destination     Protocol  Leng Info
 109 4.722259  161.116.96.243  161.116.96.222  ICMP        74 Echo (ping) request  id=0x0001, seq=3/768, ttl=128 (reply in 110)
 110 4.722417  161.116.96.222  161.116.96.243  ICMP        74 Echo (ping) reply    id=0x0001, seq=3/768, ttl=128 (request in 109)
 119 5.722770  161.116.96.243  161.116.96.222  ICMP        74 Echo (ping) request  id=0x0001, seq=4/1024, ttl=128 (reply in 120)
 120 5.722907  161.116.96.222  161.116.96.243  ICMP        74 Echo (ping) reply    id=0x0001, seq=4/1024, ttl=128 (request in 119)
 131 6.724777  161.116.96.243  161.116.96.222  ICMP        74 Echo (ping) request  id=0x0001, seq=5/1280, ttl=128 (reply in 132)
 132 6.724947  161.116.96.222  161.116.96.243  ICMP        74 Echo (ping) reply    id=0x0001, seq=5/1280, ttl=128 (request in 131)
 134 7.726687  161.116.96.243  161.116.96.222  ICMP        74 Echo (ping) request  id=0x0001, seq=6/1536, ttl=128 (reply in 135)
 135 7.726834  161.116.96.222  161.116.96.243  ICMP        74 Echo (ping) reply    id=0x0001, seq=6/1536, ttl=128 (request in 134)

    [Checksum Status: Good]
    Identifier (BE): 1 (0x0001)
    Identifier (LE): 256 (0x0100)
    Sequence number (BE): 3 (0x0003)
    Sequence number (LE): 768 (0x0300)
    [Request frame: 109]
    [Response time: 0.158 ms]
  ▷ Data (32 bytes)
```

Figure 27: Visualization of a ping to the FPGA sent from a PC, all four request have been answered

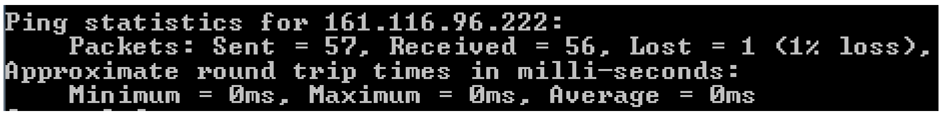
As we were expecting all the ping request have been answered the packet loss is of a 0% here, the response time has been of 0.153ms (average).



```
Pinging 161.116.96.222 with 32 bytes of data:
Reply from 161.116.96.222: bytes=32 time<1ms TTL=128
Reply from 161.116.96.222: bytes=32 time<1ms TTL=128
Reply from 161.116.96.222: bytes=32 time<1ms TTL=128
Reply from 161.116.96.222: bytes=32 time<1ms TTL=128

Ping statistics for 161.116.96.222:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Figure 28. Windows cmd results of Ping, the packet loss is of 0%, the times haven't been correctly calculated as windows doesn't calculate pings inferiors to 1ms.

The second test have given the following results:

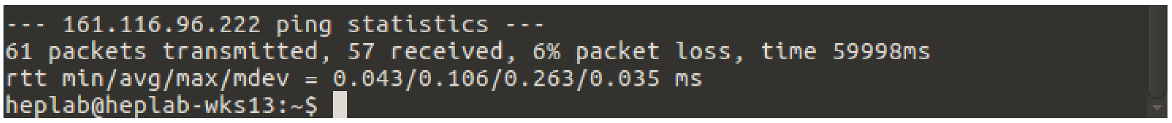


```
Ping statistics for 161.116.96.222:
    Packets: Sent = 57, Received = 56, Lost = 1 (1% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Figure 29. Windows results of ping during 60 seconds, in this case we have some packet loss (1%).

In this case the system has again worked, but one of the packets have been lost (1 % of loss). The average times are around ≃0.15 as before.

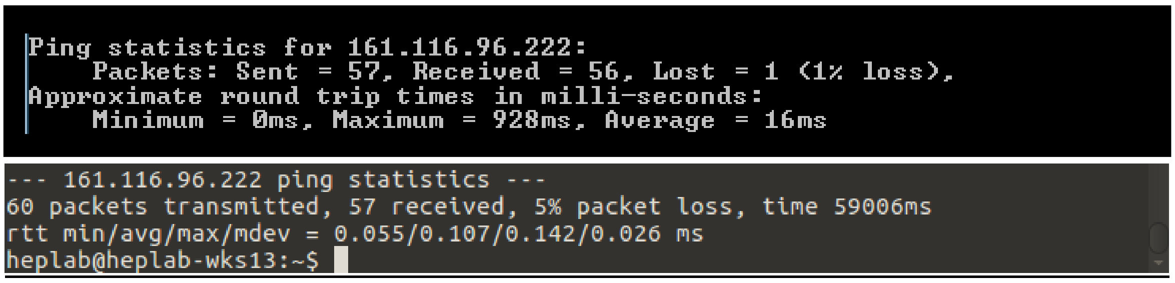
We will now repeat the second test but now using a computer with Linux (the ping will be different), later, we will make a fourth and last test, this time sending pings from both PC, windows and Linux at the same time, during the full minute, this last test will be repeated 50 times, in order to get an average value. The third test results are the following ones:



```
--- 161.116.96.222 ping statistics ---
61 packets transmitted, 57 received, 6% packet loss, time 59998ms
rtt min/avg/max/mdev = 0.043/0.106/0.263/0.035 ms
heplab@heplab-wks13:~$
```

Figure 30. Linux results of ping during 60 seconds, we have some packet loss (6%).

As we can see, the Linux test give a slightly worst result than in Windows, as when the test was run, the packet loss was always higher. Anyway, it is very low so it still passes. Finally, the fourth test results obtained are the following ones:



```
Ping statistics for 161.116.96.222:
    Packets: Sent = 57, Received = 56, Lost = 1 (1% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 928ms, Average = 16ms
```

```
--- 161.116.96.222 ping statistics ---
60 packets transmitted, 57 received, 5% packet loss, time 59006ms
rtt min/avg/max/mdev = 0.055/0.107/0.142/0.026 ms
heplab@heplab-wks13:~$
```

Figure 31. Windows and Linux simultaneous Ping for 60 seconds, result of one of the 50 test. Packet loss of 1% for windows and 5% for Linux.

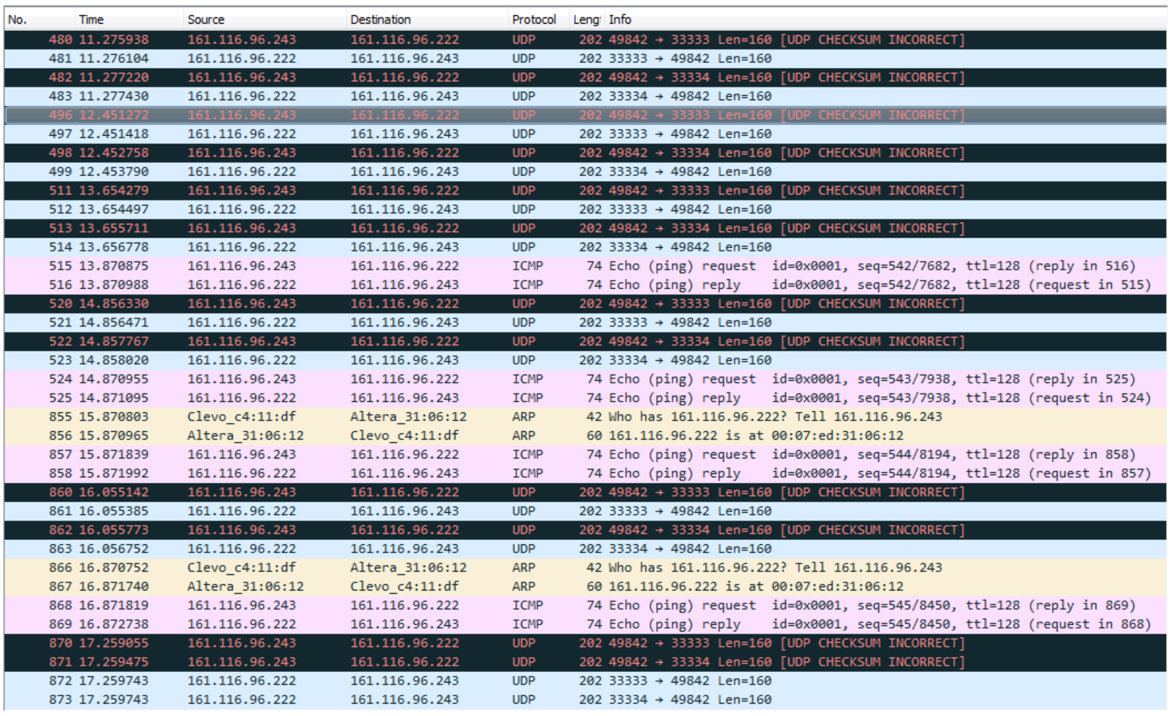
Table XIII. Average, Median, Maximum and Minimum loss values of 50 test with Windows and Linux sending pings simultaneously

OS	Nº test	Average loss	Median loss	Max. loss	Min. loss
Windows	50	2,55%	2%	5%	1%
Linux		4,8%	4 %	7%	3%

This last has given similar results to the previous ones, both pings of different OS working simultaneously does not affect the performance of the stack. Looking at the averages, maximums and minimums we see that loss of packets has a fairly constant values, the system works as expected in a stable way.

d) Arbiter test – Multiple services working simultaneously

This is a simple test where we are going to send ping and echo replies at the same time in order to check the correct performing of the arbiters and other management parts. We have added a second Echo block at port 3334 that will be requested too:



```
No.  Time       Source            Destination       Protocol Leng Info
 480 11.275938  161.116.96.243    161.116.96.222    UDP      202 49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
 481 11.276104  161.116.96.222    161.116.96.243    UDP      202 33333 → 49842 Len=160
 482 11.277220  161.116.96.243    161.116.96.222    UDP      202 49842 → 33334 Len=160 [UDP CHECKSUM INCORRECT]
 483 11.277430  161.116.96.222    161.116.96.243    UDP      202 33334 → 49842 Len=160
 496 12.451272  161.116.96.243    161.116.96.222    UDP      202 49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
 497 12.451418  161.116.96.222    161.116.96.243    UDP      202 33333 → 49842 Len=160
 498 12.452758  161.116.96.243    161.116.96.222    UDP      202 49842 → 33334 Len=160 [UDP CHECKSUM INCORRECT]
 499 12.453790  161.116.96.222    161.116.96.243    UDP      202 33334 → 49842 Len=160
 511 13.654279  161.116.96.243    161.116.96.222    UDP      202 49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
 512 13.654497  161.116.96.222    161.116.96.243    UDP      202 33333 → 49842 Len=160
 513 13.655711  161.116.96.243    161.116.96.222    UDP      202 49842 → 33334 Len=160 [UDP CHECKSUM INCORRECT]
 514 13.656778  161.116.96.222    161.116.96.243    UDP      202 33334 → 49842 Len=160
 515 13.870875  161.116.96.243    161.116.96.222    ICMP      74 Echo (ping) request  id=0x0001, seq=542/7682, ttl=128 (reply in 516)
 516 13.870988  161.116.96.222    161.116.96.243    ICMP      74 Echo (ping) reply    id=0x0001, seq=542/7682, ttl=128 (request in 515)
 520 14.856330  161.116.96.243    161.116.96.222    UDP      202 49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
 521 14.856471  161.116.96.222    161.116.96.243    UDP      202 33333 → 49842 Len=160
 522 14.857767  161.116.96.243    161.116.96.222    UDP      202 49842 → 33334 Len=160 [UDP CHECKSUM INCORRECT]
 523 14.858020  161.116.96.222    161.116.96.243    UDP      202 33334 → 49842 Len=160
 524 14.870955  161.116.96.243    161.116.96.222    ICMP      74 Echo (ping) request  id=0x0001, seq=543/7938, ttl=128 (reply in 525)
 525 14.871095  161.116.96.222    161.116.96.243    ICMP      74 Echo (ping) reply    id=0x0001, seq=543/7938, ttl=128 (request in 524)
 855 15.870803  Clevo_c4:11:df    Altera_31:06:12   ARP       42 Who has 161.116.96.222? Tell 161.116.96.243
 856 15.870965  Altera_31:06:12   Clevo_c4:11:df    ARP       60 161.116.96.222 is at 00:07:ed:31:06:12
 857 15.871839  161.116.96.243    161.116.96.222    ICMP      74 Echo (ping) request  id=0x0001, seq=544/8194, ttl=128 (reply in 858)
 858 15.871992  161.116.96.222    161.116.96.243    ICMP      74 Echo (ping) reply    id=0x0001, seq=544/8194, ttl=128 (request in 857)
 860 16.055142  161.116.96.243    161.116.96.222    UDP      202 49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
 861 16.055385  161.116.96.222    161.116.96.243    UDP      202 33333 → 49842 Len=160
 862 16.055773  161.116.96.243    161.116.96.222    UDP      202 49842 → 33334 Len=160 [UDP CHECKSUM INCORRECT]
 863 16.056752  161.116.96.222    161.116.96.243    UDP      202 33334 → 49842 Len=160
 866 16.870752  Clevo_c4:11:df    Altera_31:06:12   ARP       42 Who has 161.116.96.222? Tell 161.116.96.243
 867 16.871740  Altera_31:06:12   Clevo_c4:11:df    ARP       60 161.116.96.222 is at 00:07:ed:31:06:12
 868 16.871819  161.116.96.243    161.116.96.222    ICMP      74 Echo (ping) request  id=0x0001, seq=545/8450, ttl=128 (reply in 869)
 869 16.872738  161.116.96.222    161.116.96.243    ICMP      74 Echo (ping) reply    id=0x0001, seq=545/8450, ttl=128 (request in 868)
 870 17.259055  161.116.96.243    161.116.96.222    UDP      202 49842 → 33333 Len=160 [UDP CHECKSUM INCORRECT]
 871 17.259475  161.116.96.243    161.116.96.222    UDP      202 49842 → 33334 Len=160 [UDP CHECKSUM INCORRECT]
 872 17.259743  161.116.96.222    161.116.96.243    UDP      202 33333 → 49842 Len=160
 873 17.259743  161.116.96.222    161.116.96.243    UDP      202 33334 → 49842 Len=160
```

Figure 32. Multiple services working at the same time with the stack

There were not been a problem with multiple services working at the same time with the stack, as it can be seen, all the services request where answered during the test.

e) MAC, IP and Port filter

Different messages have been sent to the FPGA stack with wrong IPs, MACs and Ports, all the wrong packets are discarded before they can reach higher layers. Messages with the correct info are kept, that includes the device IP and MAC, or broadcast. In the case of the ports each application is responsible of answering to its own port.

f) DAQ Interface with 3 writers

The DAQ interface service created for the stack has been tested with three writers sending messages every millisecond. Each writer has been set with events of different width. One writer sends a single byte each time, another one two bytes and the last one three bytes. The writers are designed as counters, each writer granted request increments the counter by one. This eases the job of debugging the operation of the service. We have collected data for some minutes, if all the messages have been correctly received, we should see the counter when plotting the obtained data. Losses should be easily seen as flat sections in the plot, or zeroes if we decide to plot the derivative too. After performing the test these were the obtained results:

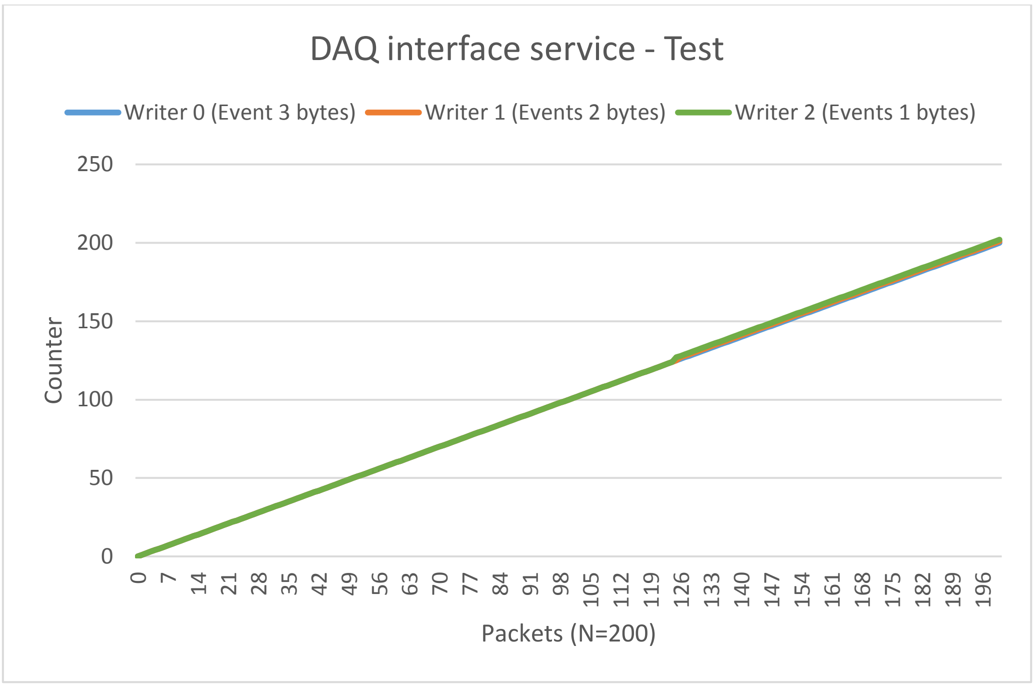


Figure 33. Data received at the DAQ interface service test, as we can see the losses are minimal.

Of 200 frames sent, we have lost a 0% for writer 0, a 0,5% for writer 1 and a 1% for writer 2. Further testing gave us similar results with an interval of losses between 0-1% approximately.

5. **Budget**

In the Table XIV we see the total cost of the project; the only cost has been in testing and designing the stack, the cost is approximate:

Table XIV. Budget Table

#	Item	Quant.	U.cost	Tax	U.cost + Tax	Total
1	HDL Engineer (Intern)	900			6€	5400€
2	Cyclone 10 Eval. Board	1	72,45€	15,21€	87,66€	87,66€
3	Ethernet cable	1	6,21€	1,39€	8€	8€
					TOTAL	5495,66€

6. Conclusions and future development

As the design has been finalized and tested, we have reached the time to summarize our work, take conclusions and talk about the possible future improvements of the project.

6.1. Conclusions

The project has been a true challenge, during the project it was needed to deepen in the operation of the main internet protocols and it has been a great exercise in HDL design, a great complement to the previous academic formation in VHDL and FPGA.

This project not only required a better understanding of HDL design, it also required the correct analysis and performance of general digital design, and a good grasp of how an FPGA works. This project, being implemented in a real system has had a lot of timing and resource variables that have been critical in the design of the stack. All these real factors, that are usually ignored in simpler or theoretical problems have been key in the development of this stack.

A greater understanding of the Internet protocol suite and its protocols had allowed to make better decisions in the design of the stack, and in the choice of the specific protocols used in its creation.

At the end, the project has ended in a successful implementation and a good learning step in the world of digital design. This project opens the possibility of much more, its modular design will allow to future designers to add features and use its current ones for its own projects.

6.2. Future development

Once finished the core of the project, we are left with a lot of space to improvement and growth.

First of all, the designed stack is not perfect, most of the development time was focused around creating the design and making it work, but the system has not been fully optimized. Some parts of the design like the ARP and DHCP have been optimized in order to use less resources, but the stack still has margin to more optimizations in order to improve the timings and to reduce the use of logic elements, registers or memory.

The design has been specifically created for a Cyclone 10 LP FPGA, although almost all the project blocks have been manually designed some specific common devices have been added using Intel macros (FIFO, RAM, DDR_IO, PLL). In order to make the stack compatible with any FPGA (manufacturer and model) the project could be improved with the creation of generic blocs. A wrapper could be made around this devices that would chose a compatible macro depending on the specific FPGA.

Focusing now on other things, we have to notice that the designed UDP/IP Ethernet stack works with just a small part of a big protocol suite, so one of the most obvious ways of improving the project would be to add more protocols to the stack in order to reach a larger number of devices and services. For example, the TCP would be a good addition to the stack as the TCP protocol is used for the opposite set of applications with respect to the UDP protocol. With TCP we will give the stack a more reliable and secure way of communication, but slower in most of the cases.

As a final front from improvements we find the Application layer, the application layer allows us to create an infinite number of custom services and user applications that will interact with the UDP/IP stack and perform its own processes. As an example, the stack has been created to work inside an FPGA set in a custom PCB with very specific functions. The design of custom services that let us interact with other ICs would be a very useful addition to the stack. Integrated circuits usually communicate between them using serial communication protocols like SPI, I2C or CAN. It has been actually proposed to add in a near future a service that will allow us to communicate using the SPI protocol via Ethernet. These services would receive instructions through Ethernet that then would be translated to SPI, this would allow the control of IC devices sending instructions from a computer, the usefulness of that is very obvious.

This is the summary of all the possible future improvements:

- Optimization of the original UDP/IP Ethernet stack.
- Generic wrappers to make the stack portable to other FPGA.
- More Internet protocols like TCP.
- Services for serial communication with ICs like SPI, I2C or CAN.
- Other custom services.

Bibliography

[1] Requirements for Internet Host – Communication Layers. IETF RFC 1122, Sept. 1989.

[2] International Organization for Standardization (1989-11-15). "ISO/IEC 7498-4:1989 - Information technology - Open Systems Interconnection - Basic Reference Model: Naming and addressing". ISO Standards Maintenance Portal. ISO Central Secretariat. 2015-08-17.

[3] *IEEE Standard for Ethernet*. IEEE Std 802.3-2018.

[4] Cadence Design Foundry. Reduced Gigabit Media Independent Interface (RGMII) – Technical data sheet. I-IPA01-0158-USR Rev 04. May 2004.

[5] Altera®. AN 477: Designing RGMII Interfaces with FPGAs and HardCopy ASICs, AN-477-2.0, January 2010.

[6] IEEE Standard for Local and metropolitan area networks--Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks--Corrigendum 2: Technical and editorial corrections. IEEE Std 802.1Q-2011.

[7] IEEE Standard for Local and Metropolitan Area Networks-Virtual Bridged Local Area Networks---Amendment 4: Provider Bridges. IEEE Std 802.1ad-2005.

[8] *Logical Link Control*. IEEE Std 802.2-1985.

[9] *Address Resolution Protocol (ARP Specification*. IETF RFC 826, November 1982.

[10] *Internet Protocol, Version 4 (IPv4) Specification*. IETF RFC 791, September 1981.

[11] *Internet Protocol, Version 6 (IPv6) Specification*. IETF RFC 2460, December 1998.

[12] *IP Datagram Reassembly Algorithms*. IETF RFC 815, July 1982.

[13] *Specification of internet transmission control program*. IETF RFC 675, December 1974.

[14] Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy. IETF RFC 1519, September 1993.

[15] *Internet Control Message Protocol (ICMP) Specification*. IETF RFC 792, Sep 1981.

[16] *User Datagram Protocol (UDP) Specification*. IETF RFC 768, 20 August 1980.

[17] *Transmission Control Protocol (TCP) Specification*. IETF RFC 793, September 1981.

[18] *Dynamic Host Configuration Protocol (DHCP) Specification*. IETF RFC 2131, March 1997.

[19] *Bootstrap Protocol (BOOTP) Specification*. IETF RFC 951, March 1985.

[20] *DHCP Options and BOOTP Vendor Extensions*. IETF RFC 2132, March 1997.

[21] "UDP/IP Ethernet". *Enclustra FPGA Solutions*. [Online] Available: https://www.enclustra.com/en/products/ip-cores/udp-ip-ethernet/ [Accesed: 13/03/2018]

[22] "1G eth UDP/IP Stack : Overview" *Peter Fall and the FIXQRL project* [Online] Available: https://opencores.org/projects/udp_ip_stack [Accesed: 13/03/2018]

[23] Lantiq. XWAY PHY Ethernet Physical Layer Devices and XWAY PHY11G Single Port Gigabit Ethernet PHY (10/100/1000 Mbit/s) PEF 7071, User's Manual, Revision 1.0, 2012-02-17.

[24] Intel®. *Intel® Cyclone® 10 LP FPGA Schematic*, 17 Total Sheets ,October 12, 2017

[25] Intel®. *FIFO Intel® FPGA IP User Guide*, Updated for Intel® Quartus® Prime Design Suite: 18.0, UG-MFNALT_FIFO, 2018-09-24.

[26] Altera Corporation, now part of Intel®. FIFO Intel® FPGA IP User Guide Double Data Rate I/O (ALTDDIO_IN, ALTDDIO_OUT, and ALTDDIO_BIDIR) IP Cores User Guide, ISO 9001:2008, 2017-06-19.

[27] “ALTPLL (Phase-Locked Loop) IP Core User Guide ". *Intel*®. [Online] Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_altpll.pdf [Accessed: 13 June 2018].

[28] Intel®. Embedded Memory (RAM: 1-PORT, RAM: 2-PORT, ROM: 1-PORT, and ROM: 2-PORT) User Guide, Updated for Intel® Quartus® Prime Design Suite: 17.0, 2017-11-06.

[29] “Memory Initialization File (.mif)". *Intel*®. [Online] Available: https://www.intel.com/content/www/us/en/programmable/quartushelp/15.0/mergedProjects/reference/glossary/def_mif.htm [Accessed: 02 August 2018].

[30] Intel®. Intel® Cyclone® 10 LP FPGA Evaluation Kit User Guide, UG-20082, 2018-02-05.

Appendices

A. UDP/IP Ethernet stack - VHDL block hierarchy

- cyclone10_eth: ethernet_stack
    - ARP:arp_protocol
        - aux_timer_fix (n times)
        - arp_fsm
            - aux_timer_fix
        - RAM_check
        - RAM_2port (ARP cache)
    - central_timer
    - DHCP:dhcp_protocol
        - aux_timer
        - RAM_DHCP
    - ICMP:icmp_protocol
        - checksum16
        - scfifo
    - UDP_rx:udp_rx_i
        - checksum16:checksum_i
        - scfifo:packet_fifo
    - UDP_tx:udp_tx_o
    - IP_RX:ip_rx_i
    - IP_TX:ip_rx_o
        - checksum16
    - eth_frame_rx
        - CRC
        - eth_Frame_validator
        - eth_frame_rx_fsm
        - dcfifo
    - eth_frame_tx
        - CRC
        - eth_frame_tx_fsm
        - scfifo
    - RGMI_in_blk:rgmii_i
        - altddio_in:rxctl_sync
        - altddio_in:rxd_ddr_component
    - RGMII_out_blk:rgmii_o
        - tx_clk_pll:pll_tx
            - altpll:altpll_component
        - altddio_out:txctl_sync
        - altddio_out:txd_ddr_component
    - arbiter:L2_arbiter
    - arbiter:L3_arbiter
    - arbiter:L4_arbiter
    - DynamicOR:L4_DynamicOR_RX
    - DynamicOR:L4_DynamicOR_TX

B. UDP/IP Ethernet stack + Services - VHDL block hierarchy

- top_entity
  - cyclone10_eth:ethernet:stack (UDP/IP Ethernet stack)
    - …see Annex A for details
  - app_0_echo:app0_echo_service (Echo Reply Service)
  - daq:daq_service (DAQ Service)
    - service_control
    - data_formatter
    - arbiter_simple
    - scfifo
  - writer:writer_daq (Writes data as if it was a DAQ)
    - data_cont_data_counter

C. User Manual: Adding a service to the UDP/IP stack

1. Increasing SERVICES_TX generic variable by N (total is number of external services that are going to transmit data). If no transmission external services are required the variable should be set to 0.
2. Increasing SERVICES_RX generic variable by N (total is number of external services that are going to receive data). If no receiving external services are required the variable should be set to 0.
3. Add the required signals in order to communicate with the stack following the next guidelines:

Note: It's important to arrange the signals in the correct order.
Note 2: If a service is going to receive and transmit both generic variables have to be increased by one

D.1. Adding a TX Service

SERVICE_DATA_TX

SERVICE_DATA_TX (8 bits)			
Signals	**Bits**	**Port**	**Description**
DATA_TX	8	OUT	Data that is going to be transmitted to the stack (8 bits)

The signals must be concatenated from service N to 0.

Example with 3 services (S2, S1 and S0):

SERVICE_DATA_TX <= S2_DATA_TX&S1_DATA_TX&S0_DATA_TX;

SERVICE_AUX_TX

SERVICE_AUX_TX (83 bits)			
Signals	**Bits**	**Port**	**Description**
IP_ADDR_DST	32	OUT	IP where we want to send (destination IP)
DST_PORT_TX	16	OUT	Destination port of Tx, port where we want to send
SRC_PORT_TX	16	OUT	Source port of Tx, this is our own port
UDP_LENGTH_LINK	16	OUT	Length of Tx packet to create
READY_TX	1	OUT	Write to TX ready
EOP_TX	1	OUT	End of packet
SOP_TX	1	OUT	Start of packet

The signals must be concatenated from top to bottom, grouped by service from service N to 0.

Example with N=3 services (S2, S1 and S0):

SERVICE_AUX_TX <=

IP_ADDR_DST_S2&SRC_PORT_RX_S2&S2_SRC_PORT_TX&S2_UDP_LENGTH_LINK&S2_READY_TX&S2_EOP_TX&S2_SOP_TX&

IP_ADDR_DST_S1&SRC_PORT_RX_S1&S1_SRC_PORT_TX&S1_UDP_LENGTH_LINK&S1_READY_TX&S1_EOP_TX&S1_SOP_TX&

IP_ADDR_DST_S0&SRC_PORT_RX_S0&S0_SRC_PORT_TX&S0_UDP_LENGTH_LINK&S0_READY_TX&S0_EOP_TX&S0_SOP_TX;

SERVICE_OR_TX

SERVICE_OR_TX (2 bits)			
Signals	**Bits**	**Port**	**Description**
RUN	1	OUT	Start of packet creation (after arbiter request)
START_TX	1	OUT	Request to arbiters

The signals have to be grouped by type signal from top to bottom, the order of the signals in each group is relevant and has to be set from N service to 0.

Example with N=3 services (S2, S1 and S0):

SERVICE_OR_TX <=

RUN_S2 & RUN_S1 & RUN_S0 &

START_TX_S2 & START_TX_S1 & START_TX_S0;

SERVICE_REQUEST_TX

SERVICE_ REQUEST_TX (1 bits)			
Signals	**Bits**	**Port**	**Description**
START_TX	1	OUT	Request to arbiters

The signals must be concatenated from service N to 0.

Example with 3 services (S2, S1 and S0):

SERVICE_IN_REQUEST <= S2_ START_TX &S1_ START_TX &S0_ START_TX;

INPUT SIGNALS TX

INPUT SIGNALS			
Signals	**Bits**	**Port**	**Description**
arbiter_granted	1	IN	All arbiters granted (L4+L3+L2)
FREE_BUS_ETH	1	IN	Stack frees all arbiters and Tx Blocks
CALL_SERVICE	1	IN	Packet created (IP+UDP), service data can be sent now

The services are always placed from N to 0. The service number X is directly related to this position. The inputs will be assigned to the following signals in the component port map as shown in the examples.

Example:

arbiter granted => SERVICE_L4_arbiter_granted(X) and SERVICE_L3_arbiter_granted and SERVICE_L2_arbiter_granted

FREE_BUS_ETH	=>	SERVICE_FREE_BUS_ETH
CALL_SERVICE	=>	SERVICE_CALL_SERVICE

D.2. Adding a RX Service

INPUT SIGNALS RX

RX SIGNALS			
Signals	**Bits**	**Port**	**Description**
DATA_in	10	IN	Data received from stack (SOP+EOP+DATA)
FIFO_RX_EMPTY	1	IN	FIFO Empty
FIFO_RX_EOP	1	IN	End of packet (UDP flag, not FIFO)
LENGTH_RX	16	IN	Length of RX packet
IP_ADDR_SCR_RX	32	IN	Source IP address from received packet
SRC_PORT_RX	16	IN	Source Port from received packet
DST_PORT_RX	16	IN	Destination Port from received packet

The inputs will be assigned to the following signals in the component port map as shown in the examples.

Example:

DATA_in	=> SERVICE_DATA_RX,
FIFO_RX_EMPTY	=> SERVICE_FIFO_RX_EMPTY,
FIFO_RX_EOP	=> SERVICE_FIFO_RX_EOP,
LENGTH_RX	=> SERVICE_LENGTH_RX,
IP_ADDR_SCR_RX	=> SERVICE_IP_ADDR_SCR_RX,
SRC_PORT_RX	=> SERVICE_SRC_PORT_RX,
DST_PORT_RX	=> SERVICE_DST_PORT_RX,

SERVICE_OR_RX

SERVICE_OR_RX (4 bits)			
Signals	**Bits**	**Port**	**Description**
DETECTED_PORT	1	OUT	Packet detected being sent to service (port filter flag)
RX_READ (RX)	1	OUT	Read FIFO

The signals have to be grouped by type signal from top to bottom, the order of the signals in each group is relevant and has to be set from N service to 0.

Example with N=3 services (S2, S1 and S0):

SERVICE_OR_RX <=

S2_DETECTED_PORT & S1_DETECTED_PORT & S0_DETECTED_PORT &

S2_RX_READ & S1_RX_READ & S0_RX_READ;

Glossary

ARP	Address Resolution Protocol
CAN	Controller area network
CAM	Content-addressable memory
CRC	Cyclic redundancy check
DAQ	Data acquisition system
DHCP	Dynamic Host Configuration Protocol
FIFO	First in First out Memory
FPGA	Field-programmable gate array
FSM	Finite-state machine
I2C	Inter-Integrated Circuit
IC	Integrated circuit (chip)
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
MAC	Media access control address
MII	Media independent interface
PCB	Printed circuit board
RAM	Random Access Memory
RGMII	Reduced Gigabit Media-Independent Interface
RX	Reception (telecommunications)
SPI	System packet interface
TX	Transmission (telecommunications)
UDP	User Datagram Protocol