TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TERO IHAMÄKI

FPGA BASED ETHERNET MEDIA LEVEL TESTER

Master's thesis

Examiner: Timo D. Hämäläinen

ABSTRACT

TERO IHAMÄKI: FPGA based Ethernet media level tester
Tampere University of Technology
Master of Science Thesis, 67 pages, 3 Appendix pages
April 2018
Master's Degree Programme in Information Technology
Major: Embedded Systems
Examiner: Professor Timo D. Hämäläinen

Keywords: FPGA, VHDL, Ethernet, testing, debugging, monitoring, low level

Ethernet is a mature technology with wide usage area in devices communicating with each other. Internet of Things is constantly increasing the number of devices in the world-wide network. Security of aspects of these devices should be considered carefully, creating the need to test devices on Ethernet level.

This thesis presents design and implementation of a test device functioning on media level of Ethernet. By modifying, injecting and monitoring Ethernet frames transmitted on the communication media, the functionality of a device under testing can be debugged and tested in different situations. The test device is a multi-purpose communication tester, providing several possible usage areas, but is originally developed for debugging and testing purposes for a custom communication protocol used in Ethernet based automation system. At the time of writing this, the tester is already in use as a part of other test equipment of the automation system.

The tester is implemented using existing automation device, consisting four Ethernet ports and an FPGA chip. All functionality is implemented on FPGA, making the implementation work on the hardware level. This brings new possibilities compared to testing Ethernet devices with multifunctional processor based design. The tester is to be connected to the communication media between the tested devices. In a normal situation, the tester device is routing unmodified frames through it with only insignificant delays, making it invisible to surrounding devices. However, based on user commands, frames can be modified runtime or inject new frames to inputs of a device under testing.

There are some related commercial Ethernet testing devices already on the market. However, those are found to be expensive and does not provide similar features for frame modifying than the device introduced here. Main drivers for the project was to develop low cost or free testing and monitoring device with the possibility to update the features based on needs found in future.

The thesis presents basics of Ethernet with two lowest OSI-layers, functional principles of FPGA chips and other needed aspects on theory level. The theory is then followed by introducing functionalities, design, and implementation of the device. Afterwards, we take a look at the actual tests, how those function and observations noticed during testing. Lastly, we summarize briefly the thesis and consider the possible future development of the project.

TIIVISTELMÄ

Tero Ihamäki: FPGA based Ethernet media level tester
Tampereen teknillinen yliopisto
Diplomityö, 67 sivua, 3 sivua liitteitä
Huhtikuu 2018
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Sulautetut järjestelmät
Tarkastaja: Timo D. Hämäläinen

Avainsanat: FPGA, Ethernet, VHDL, testaus, debuggaus, monitorointi

Ethernet on pitkään käytössä ollut ja laajasti vakiintunut kommunikaatiomedia. Internet of Things (IoT) laitteet kasvattavat verkkoon liitettyjen laitteiden määrää entisestään. Näiden laitteiden tietoturvaan täytyisi kiinnittää erityistä huomiota, synnyttäen tarpeen laitteiden testaamisella Ethernetin laitteistotasolla.

Tässä diplomityössä on rakennettu Ethernetin laitteistotasolla toimiva testaus- ja monitorointijärjestelmä. Muuttamalla ja lisäämällä uusia paketteja sekä monitoroimalla Ethernet väylää voidaan väylälle liitettyjä laitteita testata ja selvittää niissä piileviä ongelmia. Laite tarjoaa monikäyttöisen, useaan eri sovellusympäristöön käytettävän testilaitteiston, vaikka olikin alun perin suunniteltu automaatiojärjestelmässä käytettävän Ethernet pohjaisen kommunikaatioprotokollan testaukseen. Tätä kirjoitettaessa testauslaitetta käytetään jo muun testausympäristön tukena automaationjärjestelmää testattaessa.

Testauslaite on toteutettu olemassa olevan automaatiojärjestelmässä käytettävän sulautetun laitteen päälle, joka sisältää neljä Ethernet porttia ja FPGA piirin. Kaikki toiminnallisuus on toteutettu FPGA piiriä käyttäen, joten laitteisto toimii käytännössä rautatason toteutuksena. Tämä tuo uusia mahdollisuuksia Ethernet laitteiden testaukseen verrattuna perinteiseen prosessoriarkkitehtuuriseen tapaan. Testauslaite kytketään Ethernet-mediaan testattavien laitteiden väliin. Toteutettu laitteisto modifioi verkkoliikennettä reaaliaikaisesti käyttäjän komentojen mukaisesti. Muun ajan Ethernet väylälle liitetty laite pysyy näkymättömissä muulta verkolta reitittäen verkolla liikkuvat paketit lävitseen, aiheuttaen vain merkityksettömän viiveen.

Joitakin vastaavia laitteita on jo olemassa markkinoilla, jotka ovat kuitenkin hyvin kalliita eivätkä vastaa ominaisuuksiltaan tässä työssä esitettyä laitetta. Työn tärkeimpinä tavoitteina oli kehittää edullinen tai jopa ilmainen testaus- ja monitorointijärjestelmä, jota voidaan päivittää käyttäjäkokemusten perusteella.

Työssä esitellään Ethernetin kaksi alinta OSI-kerrosta, FPGA-piirin toimintaperiaate sekä muita työn tekemiseen vaadittavia tietoja teorian tasolla. Lisäksi esitellään toteutetun testauslaitteen perusperiaatteita ja arkkitehtuuria. Tämän jälkeen esittelemme toteutettuja testejä, näiden käyttöä ja huomioita testejä tehdessä. Lopuksi esittelemme tulokset ja loppupäätelmät sekä mietimme mahdollisia kehityskohteita.

ALKUSANAT

Tämä diplomityö on tehty Tampereen teknillisen yliopiston tietokonetekniikan laitokselle. Työn ohjaamisesta vastasi Timo D. Hämäläinen. Työn aikana sain ohjaajilta arvokkaita neuvoja sekä ohjausta työn tekemiseen. Kiitokset läheisilleni ja ystävilleni sekä erityisesti tyttöystävälleni tuesta ja kannustuksesta.

Tampereella, 22.4.2018

Tero Ihamäki

CONTENTS

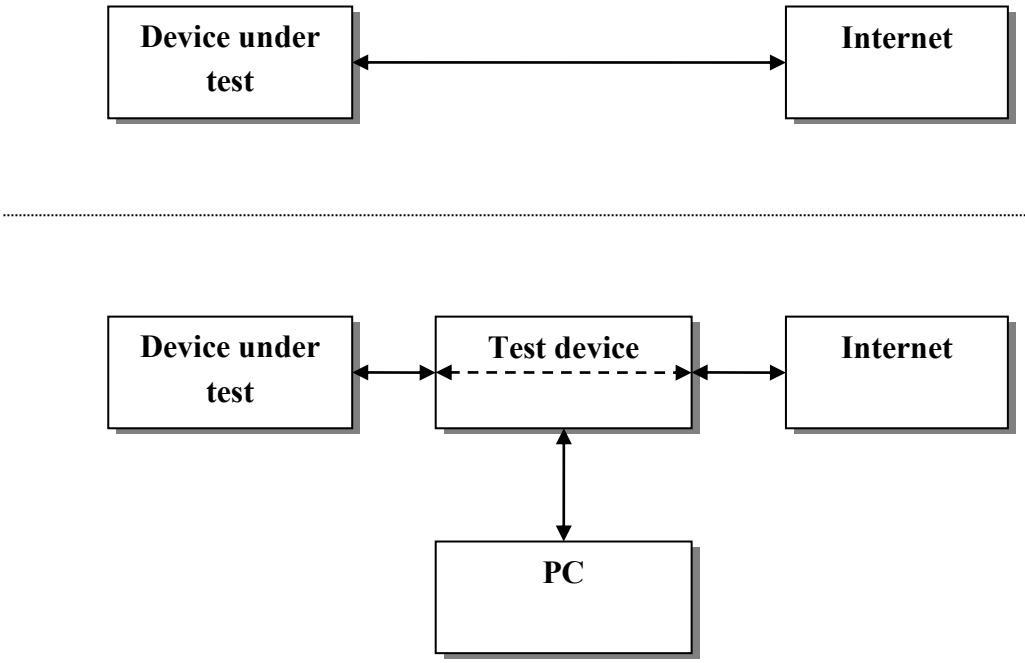
ABBREVIATIONS

ASIC	application specific integrated circuit
CLB	configurable logic block
CRC	cyclic redundancy check
DDR	double data rate
FF	flip-flop
FIFO	first-in-first-out
FPGA	field programmable gate array
HSR	High-availability Seamless Redundancy
IEEE	Institute of Electrical and Electronics Engineers
IOT	Internet of Things
IP	Internet Protocol
IPG	InterPacket Gap
ISO	International Organization for Standardization
LUT	look-up table
MAC	media access control
MII	media independent interface
MITM	man-in-the-middle (attack)
OSI	Open Systems Interconnection
PHY	physical (layer)
RAM	random access memory
RX	receive, reception
TCP	Transmission Control Protocol
TX	transmit, transmission
UDP	User Datagram Protocol
VHDL	Very High Speed Integrated Circuit Hardware Description Language

1. INTRODUCTION

From the beginning of the introduction of the Internet in the 1960s, the number of electronic devices connected and communicating with each other has been rising rapidly. In 2017, almost half of world habitants are using Internet and more than 80 % in developed countries [6]. Internet of Things (IoT) boom in the late 2010s marked another remarkable rise in a number of devices. Devices are no longer controlled only when humans are controlling those, but instead, information is shared automatically between devices as well. Suddenly many the of most traditional consumer appliances like TVs, refrigerators, cars and even toothbrushes are connected to the Internet and sharing information to cloud services. When companies are pushing more and more devices to the Internet, profits are often prioritized over security aspects. When the amount of Internet devices increases, increases also a number of possible targets to be attacked and hacked. [4] Hacking consumer devices usually places the individual and his/her personal information in danger, but when considering technology in industrial areas like power plants, the danger rises to national or even global level [5].

The goal of the thesis is to introduce one concrete approach to improve security aspects of the devices pushed to the Internet. All the devices that are connected are relying on same principles of sharing information by sending and receiving data packets through communication media. By injecting a device acting as a tester to the communication media level itself, the devices using the communication media can be tested by monitoring and modifying the data packets on the media. This way inputs and outputs of a device can be tested in its real usage environment. Usage and connectivity example can be seen in Picture 1.1. This thesis introduces a test device with twisted pair cabling Ethernet ports, but also devices with wireless media WLAN (wireless local area network) can be tested if connection media is first converted to Ethernet e.g. by using a router.



Picture 1.1 *Overview of the test system*

In order to test in Ethernet media level, a hardware level system design is needed to avoid long delays introduced by the tester itself. Traditional approach using the software on microprocessors is not fast enough, because functions, like reading memory and handling interrupts, introduce too much delays. With microprocessors, different functionalities have to compete for available resources, which will delay throughput from input to output further every time a new piece of functionality is added. That is a common problem for all sequential systems like microprocessors, a problem that won't be truly solved even by adding a higher and higher performance processors.

However, designing system in hardware offers a new kind of approach by providing a parallel handling of data throughput for different parts of functionalities. In the past designing a system in a hardware has been cumbersome and only a talent of a few. However, designing and implementing digital systems in hardware has become easier after the introduction of programmable hardware logic and hardware description languages. Many design aspects and challenges in the implementation of a digital circuit with Application Specific Integrated Circuit (ASIC) can be abstracted away when using Field Programmable Gate Arrays (FPGA). In terms of digital circuits, using areas of FPGA circuits are almost limitless, as FPGA can implement basically any digital circuit. Compared to ASIC, FPGA also provides the possibility to update the features afterwards. This said, ASIC was not considered as an implementation technology, as FPGA is more suitable and readily available technology for this project.

The test device project was ordered by an international industrial company working in the area of engine mechanics. Products of the company also include automation system for monitoring and controlling the engine products. The automation system consists of several embedded device communicating with each other and with computers using communication media. Previously Controlled Area Network (CAN) bus was used for the communication purposes, but as the amount of data is increasing, the CAN bus is replaced by Ethernet media providing more communication bandwidth. The nature of usage of the automation system forces it to be as robust as possible, introducing a need for a test system injecting unexpected anomalies to the communication media and test the ability of automation system to cope with such situations. During the development of the communication protocol used in the automation system, problems of lost frames were faced. Means to monitor the communication media is required to solve these problems. On the other hand, a way to reproduce and create seen and unseen problems with the protocol requires a way to modify frames or inject completely new frames would have been highly useful during the development. The test device implemented for this thesis is supposed to tackle all these issues.

A development board for the test device development project is provided also by the company. The development board is actually an embedded device used as a part of the automation system, which will be re-used for a completely different purpose in this project than what it was originally designed for. However, apart from various other inputs, outputs, and even a microcontroller, the device also consist four Fast Ethernet 100 Mb/s physical layer transceivers with RJ45 ports and an FPGA chip with straight connections to the transceivers. So it is also an ideal device for this project, even though it does consist one major restriction as well. The Ethernet speeds in this project are limited to Fast Ethernet 100 Mb/s category, ruling out commonly used Gigabit Ethernet with 1000 Mb/s speeds.

Work on this thesis has been mostly focused on implementation of the test device. The purpose of this document is to describe fundamentals of the system. Firstly we go through the theory used around the system, describing used technical aspects. Secondly, we go through features and facts of the system, giving a technical overview of architectural and algorithmic decisions. Lastly, we will show results of the project with example tests, as well as observations and possible next steps with the implementation.

2. THEORY

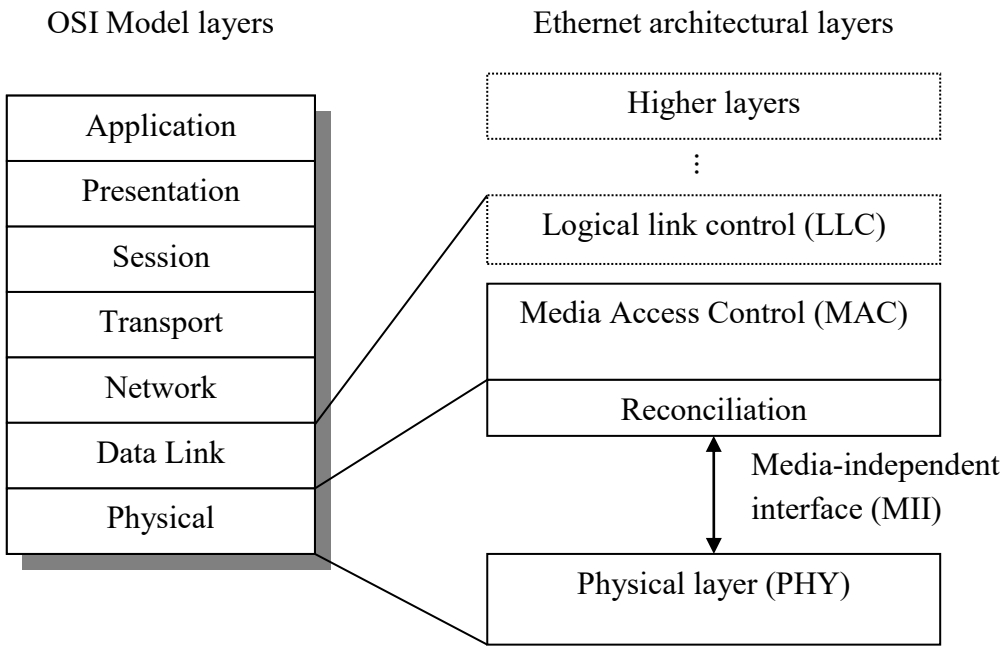
To understand the implementation aspects of the test device with the usage of FPGA technology, a preview of the used technologies will be needed. We will first look into Ethernet technologies used in the test system. After that, we will focus on fundamentals of programmable hardware in terms of FPGA technology, how it works and what kind of design aspects is needed.

2.1 Ethernet

Group of technologies of Ethernet is standardized by IEEE. The first standard for IEEE Std 802.3 was introduced in 1983 with a half-duplex communication at a maximum data rate of 10 Mb/s [8]. After that, the Ethernet technology has been advancing with a full-duplex operation and higher data rates up to 10000 times the original 10 Mb/s, with new 100 Gb/s standard. Robust performance and constant development have placed Ethernet to dominant technology in local area networks (LAN) with wide usage in the industry as well. The latest IEEE standard revision is IEEE Std 802.3-2015, published in 2016.

2.1.1 Overview

Hardware level testing of Ethernet can be concentrated on two lowest layers of ISO Open Systems Interconnection model (OSI model, ISO/IEC 7498-1 [9]). The goal of OSI model is to abstract communication system to a layered presentation based on different functionality and protocols used in each layer. The model is divided into seven abstraction layers, lowest being a physical layer (PHY). PHY is responsible to handle transmission of opaque data from the device to another through a physical medium. The second layer in OSI model is a data link layer, providing means of encapsulating data into frames to be communicated over the PHY. Data link layer also has other responsibilities like detecting errors during transmissions and addressing nodes connected to the networks. Data link layer is divided into two sub layers, media access control (MAC) and logical link layer (LLC). From these two only MAC layer is mandatory from Ethernet point of view, LLC layer being unspecified among IEEE Std 802.3 standard. Architectural model of Ethernet described in IEEE Std 802.3 standard is intentionally similar with physical and MAC sub layer in the OSI model, as shown in Picture 2.1.



Picture 2.1 *IEEE 802.3 Ethernet layer architecture. Adopted from [7].*

Layers communicate only with layers directly above and below them and as interfaces between them are supposed to be kept unchanged, internal implementation of the layer can be changed freely. Thus, one implementing, for example, a MAC layer to a device does not have to consider the internal implementation of the PHY. Conceptually MAC layer is only sending and receiving single bits, which introduces the need for slim reconciliation layer between MAC sub layer and Media-Independent Interface to capsulate single bits into a format accepted by MII. Media-independent interface is on the other hand used to hide the implementation of different PHY devices, making it possible to intermix PHYs with separated MAC device. MII implementation is tied into PHY implementation, but in practice, all PHY devices supporting same transmission speeds will use same MII interface, thus the name media-independent. [7] Because Ethernet technology is based on the layered architecture, a developer does not need to know the inner functionality of physical layer level device in case MAC and PHY is separated in the design. Knowing interface of the PHY device is enough to connect MAC and implement a working Ethernet device.

IEEE 802.3 specifies many physical layer technologies. The most used are 100BASE-TX and 1000BASE-T. Both are base on twisted-pair Category 5 copper cabling, but latter uses four pairs instead of two in the former [10]. Also, 10 Mb/s transmission, 10BASE-T, was based on same shielded twisted-pair cabling, but newer technologies have taken over and not used widely anymore. The cabling method described under

standard ANSI/TIA/EIA-568-A is the most widely used Ethernet media technology and used in this thesis project as well.

Auto-negotiation feature available in PHY is made to select the highest speed possible between linked devices. It is fully backwards compatible, meaning that Gigabit device can communicate with old devices using 10 Mb/s speeds by using fastest possible common speed of 10 Mb/s. The half-duplex mode was first a dominant but has now been almost completely been replaced with full-duplex mode. In half-duplex one communication bus was shared with all linked devices, resulting in collisions of Ethernet frames when more than one linked device started transmission at the same time. Collisions were handled with a procedure called carrier sense multiple access with collision detection (CSMA/CD). Simply put, this rather complex mechanism made the linked devices to listen to changes in voltage in the cable segment. If a cable segment seemed to be at idle, the device could start the transmission. However, if another device started a transmission at the same time, a collision could be noticed from abnormalities in the voltage levels. After collision detection a jam signal is sent to cable segment making all participants wait for a random time before sending again, thus making new collision quite improbable. Collisions, of course, affect also the average speed of the cable segment, halting all communication when a collision is detected. Thus greater amount of linked devices on a cable segment affected negatively on Ethernet speeds. Full-duplex mode, on the other hand, allows communications in both directions at the same time, making collisions physically impossible. On the down-side full-duplex mode needs two twisted-pair cables instead of one in half-duplex. Nevertheless, the gains on full-duplex are far more superior, making it possible to utilize full 10/100/1000 Mb/s speeds to both directions and making communication protocol much simpler.

2.1.2 Media-independent interface (MII)

As stated in the previous chapter, the media-independent interface is located between MAC and physical layer. MAC layer is often integrated to the same integrated circuit (IC) with the PHY, but as in our device, those can be separated as well and connected together with electrical wiring on a circuit board. To send and receive data using physical layer device with MAC sub layer implementation, it is mandatory to understand basics of MII.

Throughout the years various different MII designs have been introduced by IEEE standardization organization as well as telecommunication industry. The designs differ by a number of input and output pins, clocking speed for data sampling on the interface and supported data rates. Different MII implementations can also transmit data on either both clock edges, which is called double data rate (DDR), or only with rising clock edge on single data rate (SDR) implementation. Increasing data rate on the interface forces design either to use more pins, higher clock speeds, double data rate, or a combination of these three. In Table 1 is shown the most common MII interfaces.

Table 1 MII interfaces overview

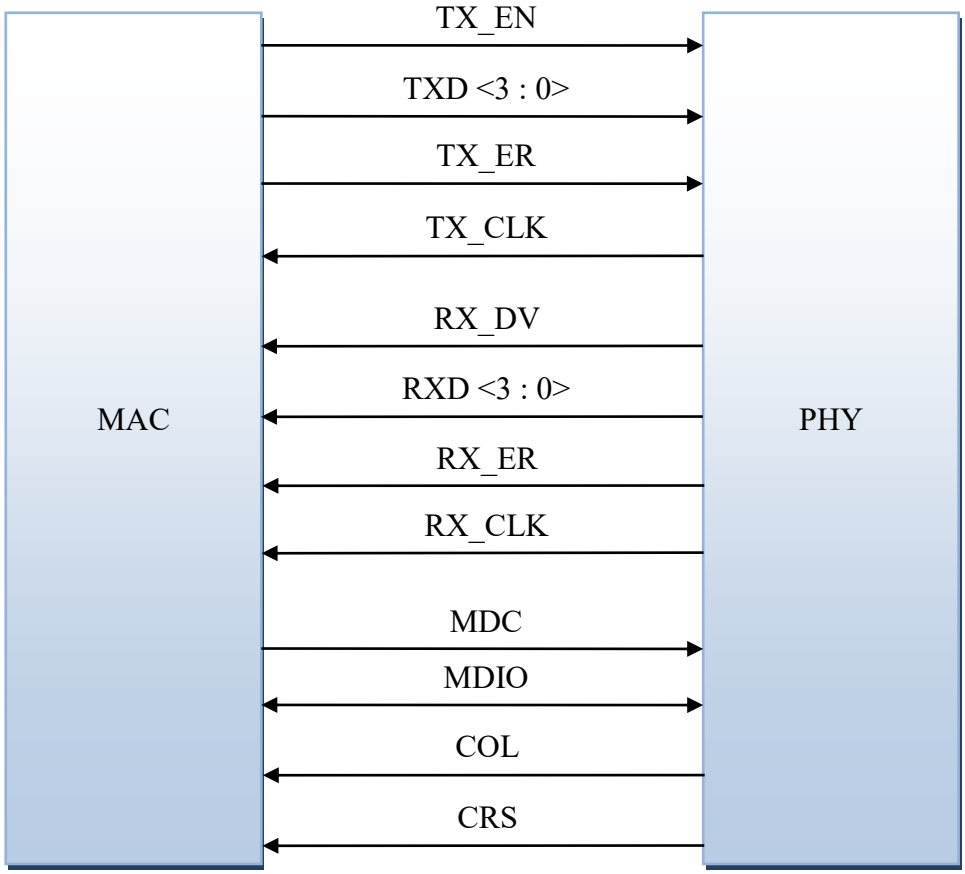
Name	Data rate [Mb/s]	Number of pins	Clock rate [MHz]	DDR
MII **Media-independent interface**	10, 100	16	25	No
RMII **Reduced media-independent interface**	10, 100	8	50	No
GMII **Gigabit media-independent interface**	1000	24	125	No
SGMII **Serial gigabit media-independent interface**	10, 100, 1000	4	625	Yes
RGMII **Reduced gigabit media-independent interface**	10, 100, 1000	12	125	Yes
TBI **Ten-bit interface**	1000	24	125	No

Selecting MII affects overall hardware level pin count and chip size on the system design. In recent years MII, GMII, and TBI have been giving way to reduced interfaces with lower pin count. This happens due to ASICs associated with MIIs are often required to have an increasing number of peripheral ports even though dye sizes are getting smaller, thus requiring as low interface pin counts from peripherals as possible [11]. On the other hand, smaller chip sizes allow higher clock rates as clock routing lengths will reduce, thus higher needed interface clock rates are usually not a problem. Still, MII designs with higher pin counts like MII and GMII still have uses in several areas. FPGA chip implementing associated MAC is often one of those.

Using high clock rates on FPGA can be often an issue as clock routing can get complicated at least in big designs. Thus it is often a good trade-off in FPGA to use more interface pins, allowing bigger designs with lower clock rates. This is also the case in the system presented in this thesis, using a separate physical layer device providing Fast Ethernet speeds at 100 Mb/s and MII interface (without any prefix) to be used with

FPGA implementing the MAC layer. More about FPGA design fundamentals will be presented in chapter 2.2.

Media-independent interface (MII) is the oldest of the presented interfaces and defined in IEEE 802.3 standard chapter 22 [7]. It was defined for Fast Ethernet family with 100 Mb/s speed and also supports older 10 Mb/s speed. Selecting the speed is made possible with the auto-negotiation feature, but it can be also forced to a selected data rate through MII management interface (MIIM). An overview of media-independent interface signals is illustrated in Picture 2.2.



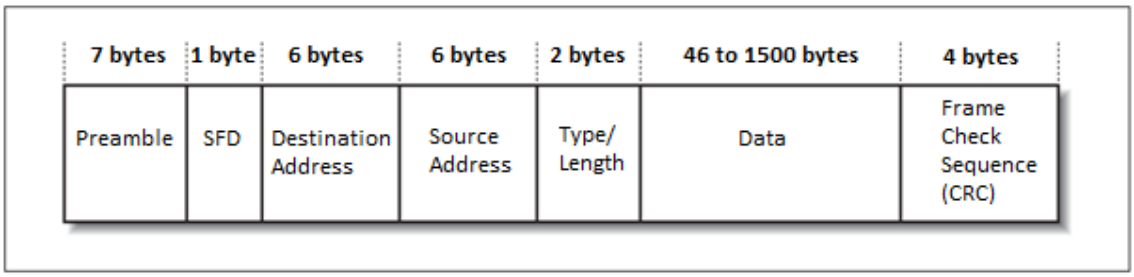
Picture 2.2 *Media-independent interface signals Adopted from [7].*

Full-duplex operation is enabled in MII with completely separate transmit and receive data paths. These data paths named TXD and RXD are both 4 bits wide. The data is synchronized with rising edges of TX_CLK and RX_CLK respectively. TX_CLK and RX_CLK are fed by PHY, so MAC has to synchronize its data handling functions with these external clocks. Clocks have a frequency of 25 MHz when operating at speed 100 Mb/s and 2.5 MHz when using 10 Mb/s speed. When sending data with TXD, TX_EN signal has to be asserted and otherwise de-asserted. TX_ER signal is optional and can

be asserted if MAC notices some issue during transmitting a frame. If TX_ER is asserted, PHY will deliberately corrupt the ongoing frame so that receiver notices the frame is not valid. When RXD has valid received data, RX_DV bit is asserted. RX_DV will stay set for the whole frame and new four bit chunk of data can be read with every rising edge of RX_CLK. RX_ER bit can tell MAC that the received frame is corrupted and should be discarded. CSR and COL signals are meaningful with half-duplex operation only and used with CSMA/CD feature. CRS (carrier sense) is set when communication medium is in use and COL is set when a collision is detected on the medium. MDC and MDIO signals are related to MII management interface. It is a serial bus used to control PHY features like auto-negotiation, speed, full-/half-duplex mode, perform a reset and detect if a link has been established.

2.1.3 Frame structure

In the layered Ethernet architecture, media access control layer was separated from other layers, meaning the same conceptual MAC layer is used with all Ethernet variants. This means that the one Ethernet and MAC frame format presented here is applicable to all transmission technologies and speeds, making the media access control device design much simpler without a need to consider all possible associated technologies. IEEE Std 802.3 describes the standardized Ethernet frame structure and is shown in simplified form in Picture 2.3.



Picture 2.3 *Ethernet frame. Adopted from [10][7].*

The frame starts with a preamble, with 7 bytes of concurrent one and zero bits (10101010). In old 10 Mb/s systems, this was used to detect the start of frame and synchronize for receiving the frame at the PHY. In never systems constant signaling is used and detection of frame start is done with other means, but preamble is still kept to keep the frame structure intact [10]. Preamble stops with start frame delimiter (SFD), setting last two bits as ones (10101011) to detect frame starts. The actual frame starts right after SFD byte, having 6-byte destination and source MAC addresses at the beginning. Every Ethernet device has individual MAC address, which can be used to target a frame to a specific device. Type/length field is used to either specify the length of the frame or contain a MAC client protocol indication. If value is 1536 or above, field is interpreted as Ethertype. Values of 1500 and below is assumed to contain frame size in bytes instead. Some fundamental EtherType values are represented in Table 2.

Table 2 *EtherType examples*

EtherType	Protocol
0x0800	Internet Protocol version 4 (IPv4)
0x0806	Address Resolution Protocol (ARP)
0x86DD	Internet Protocol version 6 (IPv6)
0x88F7	Precision Time Protocol (PTP) over Ethernet (IEEE 1588)
0x892F	High-availability Seamless Redundancy (HSR)

One Ethernet frame can contain a maximum of 1500 bytes of data payload and a minimum of 46 bytes. If a client wants to send less than 46 bytes of data through Ethernet, MAC layer needs to add padding to the payload so that at least 46 bytes of data will be sent. The frame is ended with 4 bytes long Frame Check Sequence (FCS), using a widely used error-detecting algorithm called Cyclic Redundancy Check (CRC). The 4 byte CRC value is computed as a function of the contents of MAC frame, from destination address to end of data, thus excluding the FCS itself. Transmission errors like single flipped bits can usually be noticed with frame check sequence algorithm [12].

The frame is ended when TX_EN signal is set low. In practice, in 10 Mb/s systems the signal in Ethernet channels is put to idle and in faster systems, a specific end symbol is sent by PHY [10]. Special symbols are possible because a total of 5 bits are used to represent 4 actual data bits on Ethernet bus, giving room to symbols that have a specific meaning other than data bits. These specific meanings include the start of the stream, stop of the stream, transmit error, idle and sleep. Definitions of 5-bit encoding symbols can be found e.g. from Spurgeon [10].

IEEE Std 803.2 defines InterPacket Gap (IPG) in clause 4.2.3.2.2 [7]. It states that between every Ethernet frame a minimum duration of 96 bit times of inactivity is required.

2.1.4 Internet protocol v4 (IPv4) and User datagram protocol (UDP)

As the communication from a user PC to a test device is using User Datagram Protocol (UDP), it is beneficial to take a look briefly at this higher level protocol as well. UDP works on transport layer of the OSI model. Thus under UDP frame header, a network layer protocol from OSI model is needed as well. We will focus on Internet protocol v4 (IPv4) as it the most common and used in our device as well.

The UDP protocol is connectionless communication model, which exposes applications using it to some degree of unreliability of the frames. There is no guarantee that the frames sent will reach the destination through automatic re-transmission of lost frames and ordering of the frames might differ from the original. However, as a trade-off, we get a minimal and simple protocol with a wide support from readymade software libraries and simple usage.

Table 3 *IPv4 and UDP headers. Adopted from [12]*

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	Source Port																Destination Port															
24	192	Length																Checksum															
28	224+	Data																															

Internet protocol v4 | User Datagram Protocol

Table 3 shows Internet Protocol header containing a UDP frame as enclosed data. These will be then enclosed to the data field of the MAC frame presented in section 2.1.3. MAC frame type field is set to 0x0800 to define IPv4 as an enclosed protocol. Protocol field of IPv4 header contains number 0x11 (17) to identify UDP protocol respectively. IPv4 and UDP headers contain source and destination IP address and port field to identify correct device and application when sending and receiving data to and from device to device. Detailed information and description of header fields can be found e.g. from Tanenbaum & Wetherall [12].

2.1.5 Address Resolution Protocol (ARP)

IPv4 and UDP protocols cannot be used without implementing address resolution protocol as well. For UDP packet to reach successfully its destination it needs to know destinations MAC address, IP address and port number. Without these e.g. Windows machines doesn't approve sending a packet even if the user would know all of the needed addresses. Thus, if the control application is trying to send a control message to the test device, it will first send an ARP request if the test devices MAC address is unknown to it. However, ARP protocol is fairly straightforward and possible to be implemented in FPGA as well. If the source device needs to know the MAC address of another device having a specific IP address, it sends an ARP request broadcasted to all devices in the network. If another device in the network notices it owns the IP address, it will reply to the source device with another ARP frame with its own IP and MAC addresses filled in. [22]

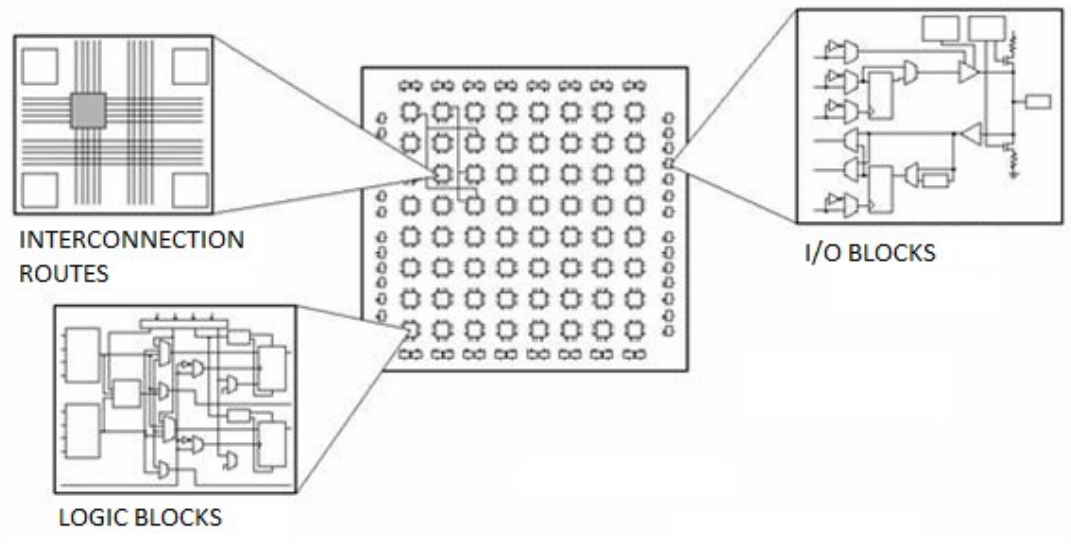
Table 4. *ARP frame [24]*

Offsets	*Octet*	*0*								*1*							
Octet	*Bit*	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
0	*0*	Protocol type (PTYPE)															
2	*16*	Hardware address length								Protocol address length							
4	*32*	Operation (OPER)															
6	*48*	Sender hardware address (SHA) (first 2 bytes)															
8	*64*	(next 2 bytes)															
10	*80*	(last 2 bytes)															
12	*96*	Sender protocol address (SPA) (first 2 bytes)															
14	*112*	(last 2 bytes)															
16	*128*	Target hardware address (THA) (first 2 bytes)															
18	*144*	(next 2 bytes)															
20	*160*	(last 2 bytes)															
22	*176*	Target protocol address (TPA) (first 2 bytes)															
24	*192*	(last 2 bytes)															

2.2 FPGA

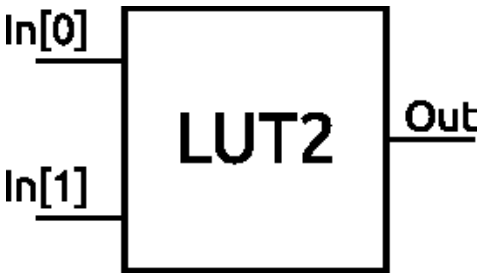
FPGA can be thought as an application specific integrated circuit (ASIC), which application is to produce a hardware that can be programmed after manufacturing to realize the desired functionality. FPGA is composed of generic logic blocks in which combinatorial logic is programmable, as well as programmable interconnections to other logic blocks. Complex digital circuit can be created by combining these small configurable logic blocks (CLB) with programmable interconnections. To communicate with peripherals connected to the FPGA chip, there are I/O blocks as a third basic element.

Picture 2.4 shows the relation between these elements.



Picture 2.4 *Basic elements of FPGA [2]*

Logic blocks are main components of the FPGA and the choice between different FPGA chips is often based on the number of available logic blocks. The more the available logic blocks, the bigger and more complex functionality the FPGA chip can implement. The logic block can be thought to consist of series of flip-flops which will remember their previous state, as well as combinatorial logical operands like AND, OR, NAND, and XOR. However, in practice logic blocks don't contain actual combinatorial logic, but rather a small chunk of RAM memory which will implement a look-up table (LUT) [18]. As a simplified example, we'll take a look at 2-input LUT in Picture 2.5.



Picture 2.5 *2-input LUT [3]*

It is important to understand that because the LUT is implemented using RAM, the output can be set to any value. Thus, it can implement any logic gate, but also other things like shift-registers or RAM. As an example, Table 5 shows a 2-input LUT implementing an AND logic gate.

Table 5 *2-input LUT implementing AND gate*

Address (In([1:0])	Value (Out)
00	0
01	0
10	0
11	1

Needless to say, a real FPGA chip uses LUTs with n-inputs and one CLB contains several LUTs. Further, modern FPGA chip can contain millions of CLBs.

While CLBs handle the logic in FPGA, interconnection routes signals between CLBs and to and from I/O blocks. Routing comes in different types, including shorter and lower speed connections connecting CLBs together, as well as long horizontal and vertical lines spanning the chip. With long, global and high speed connections it is possible

to maintain low-skew in clock signals throughout the whole chip and ensure global signals passing data from one end of the chip to other within a clock cycle.

Additionally, to these three basic elements of routing, logic blocks and I/O blocks, FPGA chips usually consist a defined number of fixed components, providing often used functionalities like arithmetic operations and memory circuits. With the usage of these highly optimized fixed blocks scattered throughout FPGA chip, it is possible to reduce the size of the system design and optimize performance. An FPGA synthesis tool often handles optimizing the logic placement so that these functions are taken into use the optimal way, but might need some additional information so that for example array of bit vector is can be utilized as memory. FPGA vendors provide also ready-made FPGA blocks to be used for often used but complex functionalities like memory controllers, digital signal processing blocks or data bus implementations to use as a part of the design.

By using these sets of components it is possible to construct bigger entities, finally forming a final circuit. By connecting the created circuit to another component through I/O blocks to for example external memories, peripherals or general purpose microcontrollers, the FPGA design can act not as a separate system, but as a part of a bigger system on a chip. In modern technology industry, it is common to use an FPGA chip with a multipurpose microcontroller, FPGA handling time-critical parts of the functionality while microcontroller handles the generic part of software [2]. FPGA vendors offer also ready-made soft processors running on FPGA itself, offering same generic software possibilities without a separate microcontroller.

Generally speaking, using FPGA chips instead of ASIC designs is beneficial when production amounts are small, time-to-market is wanted to be kept low, the product is not needed to be optimized for size or power consumption or updatability or bug fixing is required after putting the product on market. Designing cost of ASIC is often multiplications of those on FPGA designs, but on the other hand manufacturing costs with ASIC are usually smaller at least with big quantities. General purpose processors, on the other hand, can offer many similar features than FPGA with lower cost, but in areas that require concurrent execution, these two aren't even competitors. [1]

2.2.1 Hardware description language

Cost benefits of the FPGA design derive from a used process in developing a device. Logics and functionality of a designed chip are described using hardware description language (HDL), which has many similarities with traditional general-purpose programming languages. It uses similar kind of imperative and procedural structures as many other programming languages, thus implemented system doesn't need to be designed in logic port level but abstracted to a level of normal e.g. mathematical expressions and conditional statements. There are two major hardware description languages,

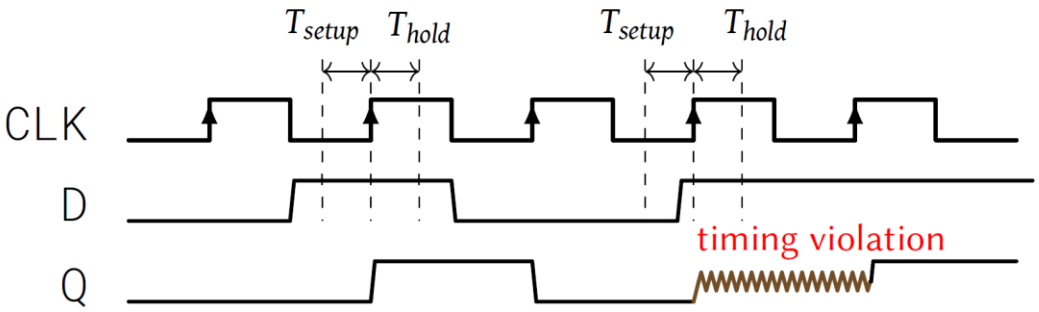
VHDL and Verilog. We will concentrate on Very High Speed Integrated Circuit Hardware Description Language, which is shortened to VHDL for obvious reasons. It is standardized as IEEE 1076. A practical introduction to the language offers, for instance, Kafig in [19], while a thorough discussion of all aspects can be found in "The Designer's Guide to VHDL" [20] by Ashenden.

2.2.2 Clocking and metastability

Probably the biggest difference between traditional software programming languages and hardware description languages is transparency for clock signals. Most of the functionalities done with hardware description languages have to be synchronized with clock signal edges, thus needing a slightly different mindset compared to programming a multipurpose processor. This is mandatory as with HDL we are designing a digital circuit instead of software and processing in the components happens synchronously with a clock in the FPGA.

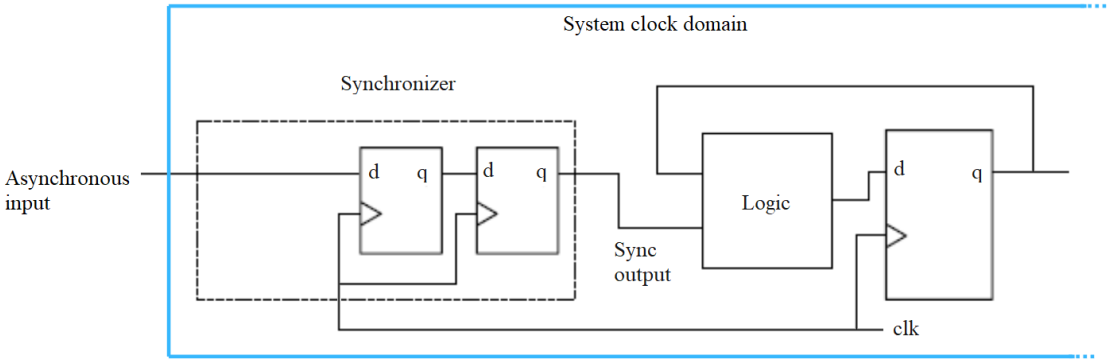
Synchronization with clock signals can rise problems unknown to a programmer used to develop software to general purpose processors. One of the most problematic situations can occur when working in multiple clock design and passing signals and data from clock domain to another. If done wrong, this can cause random error situations hard to debug and locate.

A root cause of the problem is named metastability, a physical phenomenon found in digital devices. For example, a D-type flip-flop samples its data at every rising clock edge. For the data sampling to be successful, flip-flop needs data to be stable for some time before and after the clock edge. The time before clock edge is commonly called setup time T_{setup} and time after clock edge is hold time T_{hold} [14]. If these timings are violated, the flip-flop may enter a metastable state and the output of the flip-flop will be undefined. Output can stay between valid 0 or 1 states or oscillate between those values [14]. Picture 2.6 shows an example of the metastability state in flip-flops output Q issued by timing violation in input D.



Picture 2.6 *Metastability in flip-flops output Q*

When another component is then using the output of a flip-flop in a metastable state, it can interpret the (voltage) value either logical 0 or 1, resulting in possible undefined errors in the system [14]. In theory metastable state can continue indefinitely, but in practice, it has been noticed that the probability of a component stabilizing its state increases exponentially over time. If there is not known phase difference between the clocks from where and to signals are to be synchronized, it is not possible to make fail proof implementation of a signal synchronizer. As Kleeman et al. stated: "it is generally accepted that a perfect synchronizer cannot be physically realized."[23]. Because the system cannot be made totally free from metastability issues, it should be made as metastability tolerant as possible [15]. For this purpose, commonly used technique is to use synchronizers when passing signals from clock domain to another. The purpose of the synchronizers is to synchronize asynchronous signal to system clock by just providing enough time for the metastable condition to be resolved before using it in the system [15]. Most commonly used synchronizer is two-FF synchronizer, shown in Picture 2.7



Picture 2.7 *Two-FF synchronizer [15]*

Using two-FF designs are enough in most cases, but in the second flip-flop can get the metastable state as well even though the probability is very small. Adding more chained

flip-flops to the synchronizer increases mean time between failure (MTBF) rate, but also increases delay even further from two clock cycles of two-FF design.

The two-FF synchronizer is usable only for single bit synchronization because concurrent synchronizers cannot guarantee to deliver separate bits at the same time to the following logic. For data other means of synchronization is required. One of the simplest methods is to let the data be asynchronous, but synchronize only "data valid" bit from clock to another. When this control signal is synchronized, it can be assumed that the data is stabilized as well due to the synchronization delay time. However, when multiple adjacent data chunks are needed to be transferred at a maximum throughput in an asynchronous environment, some kind of acknowledgement signal is needed to be synchronized from data receiver to sender as well to tell a data chunk read is completed. This is forming a so-called hand shaking protocol [15]. To maximize data throughput further, a first-in-first-out (FIFO) buffer can be used. In FIFO design writer can push data to the buffer without waiting for every separate acknowledgement from the reader if there is available space in the FIFO. Addresses of the FIFO can then be synchronized by using "grey code" method where only a single bit of the address changes at a time. Using these methods it is possible to create an asynchronous FIFO for data transferring with separate write and read side clocking. A full and empty signal, as well as approximation of data amount in the FIFO, is possible to be implemented as well for flow control purposes.

Implementing such behavior can get rather complicated. However, FPGA vendors usually provide readymade solutions for generating asynchronous FIFOs in user design for free of charge. If used clock speeds on both sides of the FIFO are same, reading can be done at full clock frequency after an initial waiting time for first data chunk. A usable feature for FIFOs is also first-word fall-through (FWFT), which makes the first chunk of data to be usable in the following logic before explicitly reading the data out from the buffer first. For next data to be seen in the output an acknowledgement for the first one is needed to be given.

2.2.3 Implementation process and simulation of FPGA

Even though designing systems for FPGA still brings up these challenges from logic gate level, it is also clear that with HDL implementing systems has become easier not only with ease of design but also from a verification point of view. For VHDL codes it is possible to use e.g. peer-review methods in the code level itself. For VHDL code to be used in the FPGA, it is first needed to be synthesized, a term relative to compilation in terms of traditional programming languages. The synthetization also gives errors and warnings regarding syntax errors in the VHDL codes.

Before loading the design to FPGA, the design can be simulated first. Simulator mimics the functionality of the FPGA device and can give a visual representation of digital sig-

nal states of the device in function of time. By investigating the internal digital signal state of a function block under consideration, the developer can spot possible logical errors from the design without debugging the design in actual hardware. Simulation usually requires at least one test bench, which is defining external inputs and outputs states for the testable system. This means, for example, defining external clock and reset signals and peripheral input states in regards to the clock states. Often it is also common to define separate test benches for different function blocks of the design and checking outputs of the system with a function of inputs to automatically check passing or failing of a test case. This can be compared to unit testing in general software programming. Test benches can be run after changing the implementation to check that nothing has been broken between the development steps. As with unit testing, it is often beneficial to write the test benches even before the actual functionality.

It should be noted still that the simulation environment gives an ideal representation of a logical behavioral model of the device and doesn't necessarily expose all errors found in the real device. One good example of this is the metastability issue discussed in section 2.2.2. Also, even though testing in the simulation is run successfully, it is possible that the actual implementation of FPGA logic placement and routing fails. This can happen for example because the routing tool cannot implement clock routing that can meet timing requirements of the design or just by running out of LUT components of the FPGA chip, just to name a few. However, if the routing and placement are successful, the compilation tool gives a report for example of used resources of the FPGA chip and clocking characteristics. As a final output, the synthetization tool gives an FPGA placement and routing description file which contains information how to use registers, LUTs and connecting wiring in the FPGA chip. FPGA chip is programmed using this description file to construct FPGAs logic cells and interconnect wirings in right order to achieve designed hardware system.

3. PRODUCT OVERVIEW

3.1 Development environment

The primary customer for the project is a major industry company. Their main products are controlled with automation system containing embedded devices. One of the main requirements for the project was to use one of these devices designed originally for control automation purposes, but still containing all needed components for the Ethernet tester device as well. All it needed was a complete FPGA redesign and implementation. There was also already available a method to update FPGA chip with new FPGA configuration binary, which helped slightly to get first versions of the tester up and running. At the beginning of the project, it was also found out that it was even more beneficial to use older prototypes of the automation device, which were unusable for the original use but still totally fine for tester purpose.

The automation device has following technical details:

- FPGA chip, Xilinx XC6SLX150 [21]
  - Spartan-6 Series
  - 184304 slice registers
  - 92152 slice LUTs
  - 4824 Kb RAM blocks
  - 180 DSP48A1 slices, each containing 18 x 18 multiplier, an adder, and an accumulator
- DDR2 memory, Micron Technology Inc. MT47H128M16RT-25E AAT
  - Size 2 Gb
  - Width x16
  - Clock rate 400 MHz
- 4 x Ethernet PHY, Texas Instruments DP83848T
  - Fast Ethernet 10/100 Mb/s
  - MII interface

The device has lots of other features as well like separate microcontroller and different kinds of I/Os, but those are not used in this project and thus not needed to go through here.

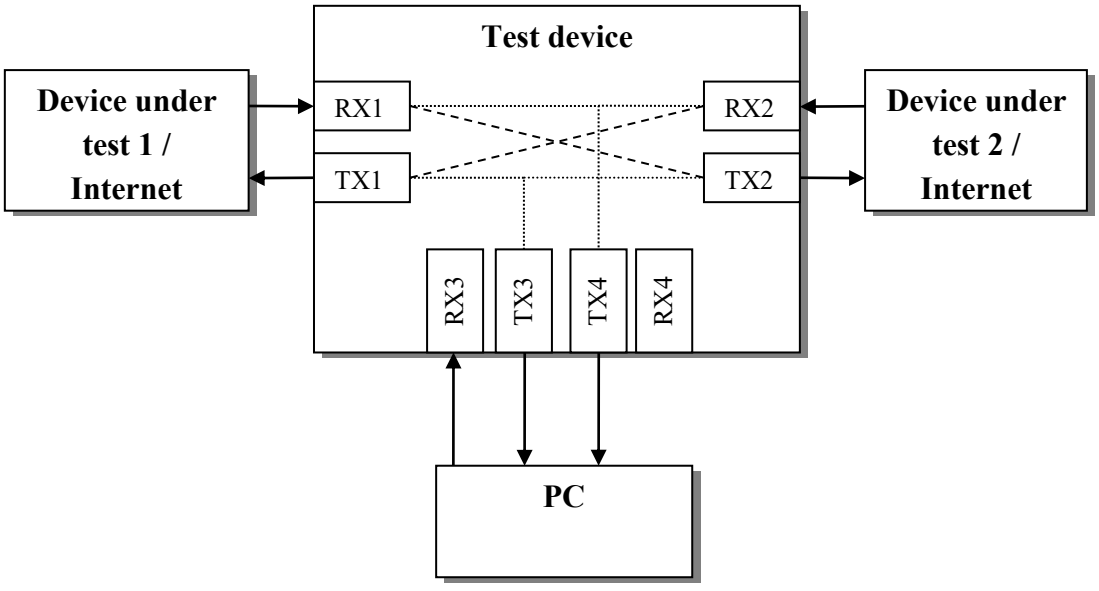
FPGA development was done with Xilinx ISE design software. There is also a newer version of FPGA development environment available from Xilinx, called Vivado Design Suite, but it does not support Spartan-6 series devices and thus cannot be used here.

FPGA simulation was done with ISim HDL simulator, which was installed together with Xilinx ISE. During the development ISim had serious stability issues so, for example, ModelSim might have been better for the purpose. The actual VHDL coding was done with XEmacs text editor even though Xilinx provides an editor with ISE as well. XEmacs was selected because its wide VHDL support giving fundamental features like automatic indentation, color coding, and auto-complete, just to name a few.

A PC tool for controlling the test device was implemented with Qt and C++. Qt is a fairly popular cross-platform application framework, which also includes easy to use UDP packet sending and receiving libraries as well as providing GUI designer with a reasonable learning curve. Qt and C++ are also widely used in the customer's other projects and was the clear choice for that reason as well.

3.2 Usage overview

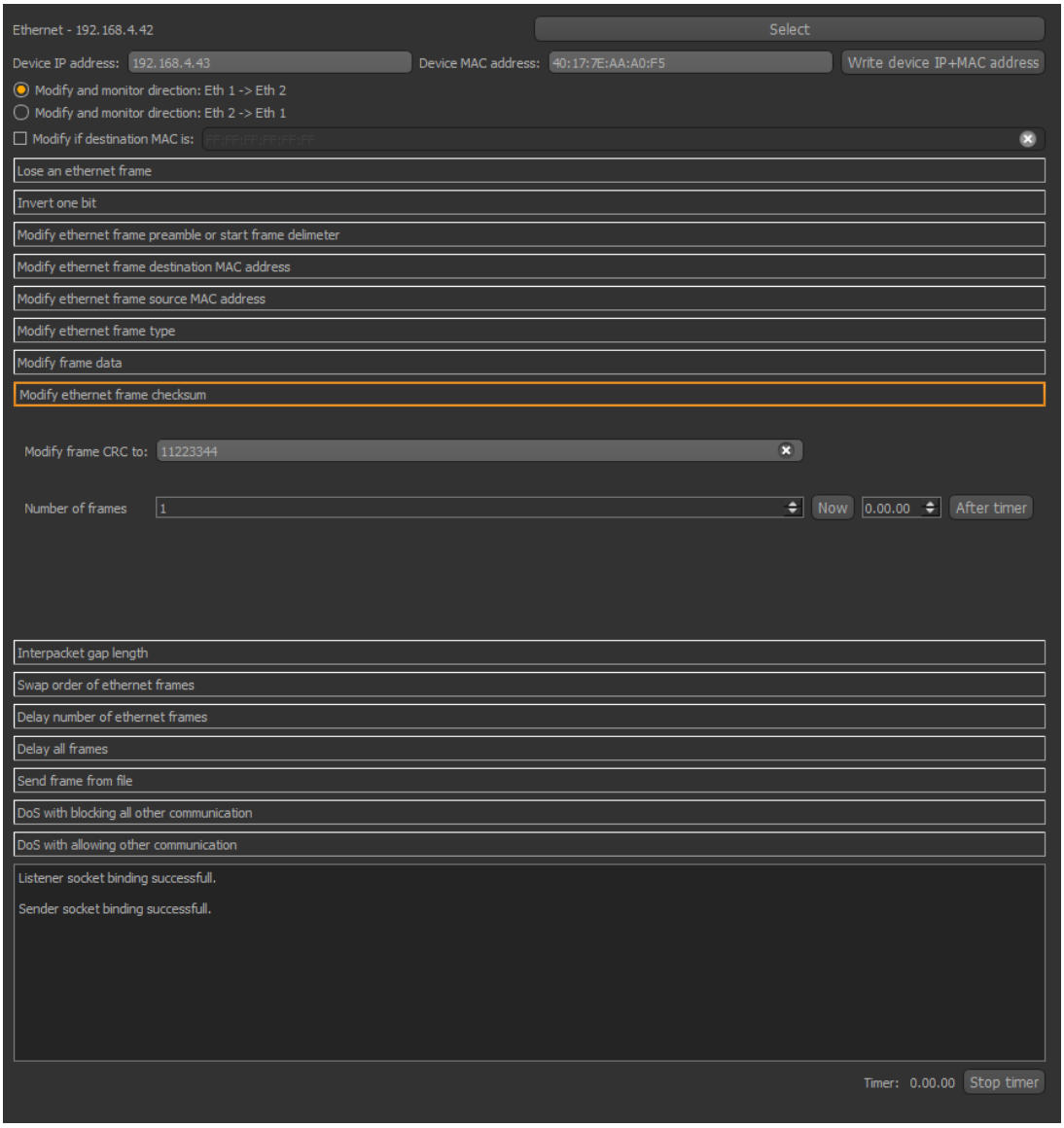
The test device is supposed to be connected between the two devices communicating using Ethernet. The device itself has four Ethernet ports providing TX- and TX+ pins and RX- and RX+ pins. Of course, commonly used RJ-45 connector can be connected to the pins with custom made cabling for easy connectivity to existing systems. Two of the Ethernet ports are connected to PC and are used to monitor Ethernet frames from tested communication line receive and transfer end of the test device. This means that frames are duplicated and rerouted on the fly to monitoring PC even though it might not be part of the tested local area network (LAN) itself. One of the Ethernet ports going to PC is also used to send control and command frames to the test device for user interaction.



Picture 3.1 *Test device connections*

Picture 3.1 shows used connections for the test device. Received frames from RX1 or RX2 are transmitted from ports TX2 or TX1 respectively. Additionally for monitoring purposes, based on user selection of modifying/monitoring direction, received frames are rerouted to TX4 from either RX1 or RX2 port. Respectively outgoing frames from either TX1 or TX2 are rerouted to TX3. RX3 port is used to send control and command frames to the test device.

When the test device is started, it is first needed to set MAC and IP addresses for the device. Otherwise, it couldn't respond to ARP requests sent by control PC and direct communication between those two couldn't be established. Control PC is able to select MAC and IP addresses for the test device and send those to the test device using Ethernet broadcast MAC and IP addresses. By defining a frame identifier to used communication protocol it is possible to distinguish address setting frame from other broadcasted frames. Communication protocol will be defined in more detail in chapter 4.1.



Picture 3.2 *Command and control PC tool user interface*

Picture 3.2 shows the user interface for controlling the test device from PC. First, the user selects Ethernet interface connected to test devices Ethernet port 3. Device IP address is filled automatically to adjacent address from selected interface IP address, but it can be modified freely if needed. A test MAC address is filled in by default, but it can be modified also in cases of possible address collisions. After initialization of addresses, the device is ready to be used. The following chapter will define features available in the test device.

3.3 Tests and features

Testing an Ethernet frame consist tests that are focused on the physical and data link layers on the OSI stack. In theory, most of the modified packets should be blocked already on physical or data link layers by receiving hardware, if frame check sequence is not modified at the same time when modifying frame data bits. Thus, if hardware handling the communication is working correctly modified packets within the tests should not stress CPU, but rather drop the frames before reaching upper layers in the OSI stack. In many cases, an operating system or application receiving the packets will request retransmission in case a missing packet is noticed. Also, the sender can notice missing acknowledgement from the receiver and send again individually. However, there are many ways to handle modified packets and the purpose of these tests is to check that participants handling the communication recover reasonably from unexpected error situation.

Many of the errors simulated by the tests are usually not possible to happen when communicating software parts are functioning normally and correctly. However, often it is not enough to assume tested software gets only valid data, but it is also feasible to test the software from other security aspects, like man-in-the-middle (MITM) attack or misbehaving transmitter software. In MITM attack a malicious third party takes control of the communication channel and can eavesdrop and modify victims' communication traffic [17]. This is why tests include also functionalities like changing the address on the fly and manipulating FCS to match with modified data.

Several tests that are not actually physical or data layer related, but still tightly connected to Ethernet communication is also implemented. These tests simulate possible real life scenarios that might cause unexpected behavior in receiving end device. These tests modify mostly entire frames instead of individual bytes or bits in the frames.

Lastly, some general testing and debugging tools are also implemented within the device. These are introduced to help in the development and debugging of new network devices by providing a view on ongoing frames and injecting user defined frames to the communication line.

Some features are not tests itself, but rather support the tests to provide more possibilities in the testing process. These features are listed below.

1. The direction of the modified packets can be altered. This means that user can select if incoming or outgoing packets from the testable device will be modified. This can provide more useful if there is a network with two or more testable devices and the testing device is placed in between of the communication media. This feature is available for all tests as well as monitoring.
2. A user can set a timer to trigger a test after a defined time has elapsed. This feature is available for all tests.
3. Frame check sequence (CRC) can be modified also to match new data bytes in a modified frame. This way also the modified frames get through cyclic redundancy check, which is often done already in hardware level and possibly to the application under testing. This is available for most of the tests that are modifying data bytes in the frames.
4. The user can set the test to only take place for frames sent to a defined destination MAC address. This will help to target only specified test subject device especially when test area consists more than one devices. This is available for many of the tests, where implementation was possible without big sacrifices in terms of code complexity (from development time perspective) and delaying test frames. This feature will be called conditional frame modifying in the next chapters.
5. The user can modify a defined amount of consecutive frames with one modifying test command. All consecutive frames are modified with the same manner described by the test.
6. The direction of modified packets can be changed on the fly. This means that if Ethernet frames coming from device A are modified when going to device B, when the direction is changed, frames coming from device B to device A is modified instead.
7. All transmitted frames from the test device are also transmitted from the Ethernet port used for controlling the device additional Ethernet port to the user PC for monitoring purposes. Additional fourth Ethernet port is used to send all Ethernet frames that are seen on receiving port of modified direction. These are especially useful to see the contents send on communication media with an external test computer with an appropriate monitoring application like Wireshark.
8. Advanced monitoring feature sends monitored frames encapsulated to UDP frame to allow seeing also broken frames in user PC. If UDP frames sent to valid MAC, IP and port combination, broken frames can be received in a listening socket on user PC without interruptions from e.g. frame checksum check in lower layers.
9. IP and MAC addresses of the control port of the test device can be modified.

In next chapters, all currently available tests are described in detail. If some of the supportive features are not available for a specific test, it is explained why it is left out.

3.3.1 Preamble and start frame delimiter (physical layer)

The user can modify next incoming Ethernet frame to consist selected amount of preamble nibbles (half a byte, 4 bits). The user can also select if start frame delimiter byte is sent or not with the next incoming frame.

The preamble is used for Ethernet physical layer hardware to detect incoming frame and synchronize receive data with receive clock. Amount of preamble bytes can be selected between a range of 0 - 38 nibbles. The amount is handled as nibbles to increase resolution, but also because fast Ethernet physical layer hands data over in 4 bit chunks, thus making the implementation slightly easier.

The conditional frame modifying is not supported by this test. The reason is that the preamble that is supposed to be modified, comes before destination address. Thus implementation would require buffering of data, which also means delaying of the frame. Implementation of conditional frame modifying feature could be done, but it was decided to be left out to avoid delaying of the frame.

3.3.2 Destination address (data link layer)

The user can modify destination MAC address of a next incoming frame to a user given 6 byte hexadecimal value. This can be used to test for example what happens when an Ethernet frame meant to be sent to device A is sent to other device B, because of a software bug or a man-in-the-middle attack. In theory physical and data layers should discard the frame before reaching upper layers to stress CPU of the testable device. When testing, it can be beneficial to turn on automatic frame check sequence correction to make sure frame is not discarded because of invalid CRC bytes.

Contrary to the preamble and start frame delimiter test, conditional frame modifying is supported here even though it does require buffering and delaying of the frame. Buffering is mandatory because destination address has to be first checked for a possible match before actually modifying the data. Amount of delaying buffering space is defined based on need, which is currently at a maximum of six bytes.

3.3.3 Source address (data link layer)

The user can modify source MAC address of a next incoming frame to a user given 6 byte hexadecimal value. Depending on the application that the frame is used for, this might or might not have an impact on testable device behavior.

3.3.4 EtherType and length (data link layer)

The user can modify EtherType or length field. Check behavior when unknown/unsupported Ethernet frame type is sent.

When used as EtherType, the length of the frame is determined by other means. Modern Ethernet PHYs uses special symbols to signal frame start and stop, but size can be also calculated based on the location of the SFD byte and InterPacket Gap. Length can be also defined in upper layers, but physical or data link layers will not handle those bytes. Thus, it is fairly possible that no frames will be discarded based on type/length field if frame check sequence is corrected.

3.3.5 Payload modifying (data link layer)

Modifying payload is done by giving an offset from payload start to bytes that will be modified, a number of bytes to replace with given data and the data actual data bytes. A maximum number of data bytes is restricted to 6 mainly to save logic gates in FPGA due to the handling of the data. Data amount could be easily expanded if there are free logic gates to use.

3.3.6 Frame check sequence (data link layer)

Modify frame check sequence value. The receiver will probably discard the frame due to invalid CRC if frame check sequence handling is implemented correctly at the testable device. Because correct place of FCS bytes will be known only after frame stop has been detected, 4 bytes will have to be buffered from the modified frame. When frame end is detected, buffered bytes will be swapped to user selected ones.

3.3.7 InterPacket Gap length

Modify InterPacket Gap (IPG) length. According to IEEE 802.3 standard, Ethernet devices should allow minimum idle time of 96 bit times. Bit time is a time respective to sending one bit on the medium. For 100 Mbit/s Fast Ethernet, this means the idle time of minimum 0.96 µs.

However, due to nature of the test, some restrictions have to be defined when modifying InterPacket Gap. To make the gap according to user defined length, data should be available for two consecutive frames. Normally cannot happen if gap length is shortened below minimum stated length in the standard and even with longer gap times would require incoming frames at close to maximum bandwidth throughput. To make the feature reliable, the previous frame have to be buffered to memory. When the second frame receiving starts, the first frame is started to be read from memory while the second is written to memory. After complete send of the first frame, the second

frame is sent from memory as quickly as possible, allowing a gap of user specified time. However, there is still some minimum modified gap length time, restricted by delays introduced by reading the memory. The minimum gap length time could be made smaller with aggressive caching when reading the memory.

3.3.8 Send a custom frame from a file

The user can define any custom Ethernet frame in a text file format, which will be sent from the testing device. The frame will be read from a file, encapsulated in a UDP packet and sent to the testing device through command Ethernet port. Frame check sequence is also calculated automatically and appended to a defined frame, so the user doesn't have to calculate it manually.

When custom Ethernet frame is received on the testing device, it will be saved straight to DDR memory. As soon as there is no ongoing receive towards the testable device, the custom frame will be read from memory to transmit Ethernet port and send to testable device. If any data is received during transmitting custom frame, it will be buffered to DDR memory instead. That is because the custom frame has to be sent completely before continuing normal operation. After the custom frame is sent, normal frames will be read from memory and ingoing frames will be buffered as well. This will be continued for so long that memory will be emptied from all data. Thus no frames will be lost from the communication media, but frames that should be forwarded at the same time with custom frame will be delayed by the amount of custom frame length.

The maximum length for the custom frame is restricted to largest commonly approved UDP data size of 512 bytes guaranteed to be sent without frame fragmentation. This is because currently UDP protocol handler implemented in test device does not support reconstructing of fragmented UDP packets.

3.3.9 Runt frames

A runt frame is an Ethernet frame with less that 64 bytes of length, which is defined as minimum length in IEEE 802.3 standard [10]. This test can be achieved easily by using custom frame sending and defining custom frame in the file shorter than 64 octets.

3.3.10 Swap order of frames

With this test, the user can swap the order of two consecutive Ethernet frames going to the testable device. When the test is started, the first incoming frame is taken from communication media and saved to DDR memory. Next frame is then received and transmitted normally, after which frame that is saved to memory is sent, thus swapping frames order. It is also possible to set a number of frames that are saved to memory before normal frame transmit, thus more than one frame can be swapped with the one that

comes after those. Swapping frame also means that the frames that are saved to memory are delayed.

3.3.11 Delay selected number of frames

The user can delay a selected number of frames from communication media for a selected amount of time. After test started, a selected amount of frames are saved to memory and not transmitted yet. With the start of the first frame that is saved to memory, also a timer is started. After the selected amount of frames are saved to memory, all incoming frames are transmitted normally to the testable device. When the timer has elapsed, saved frames are read and sent from memory in the order those arrived at the test device. Any incoming frames are buffered to memory while saved frames are sent to avoid losing any frames from communication media. The maximum amount of frames are set to 255 to fit the amount in 8 bits. Delay time is given in milliseconds and longest time is set to 65535 milliseconds, to fit the delay time in 16 bits. Amount limitations come from FPGA bus width, which requires setting some limit to the used byte amount. However, there is plenty of room to increase it if needed.

3.3.12 Delay all frames

This test is same than delaying a selected number of frames, but instead of giving a number of frames, all data is delayed for given amount of time. This means that there will be no incoming frames to a testable device for given amount of time, but after that, all data that were supposed to come during that time will be transmitted to a testable device with maximum speed allowed by available bandwidth. All frames are saved to memory during the time period and while reading the saved memory, all incoming frames are also buffered to memory to be sent in incoming order. This means that emptying the memory might take quite some time, at least if bandwidth usage is close to maximum with normal communication as well. Delay time is given in milliseconds and longest time is set to 65535 milliseconds, to fit the delay time in 16 bits.

3.3.13 DoS with blocking all communication

The user can simulate Denial of Service (DoS) attack, where one valid Ethernet frame is multiplied to a test device transmit port with maximum bandwidth throughput and send to testable device. This test uses a custom frame from file functionality, so the user can define the frame that will be flooded to the testable device. The user can also set attack length in milliseconds, maximum being 65535 milliseconds. With DoS attack simulation blocking all other communication, all other frames that would come through test device receive to transmit port is lost, so only user defined frames are received in the testable device when the test is ongoing.

3.3.14 DoS with allowing all communication

This test simulates DoS attack by sending one custom frame with maximum bandwidth throughput but allows also normal communication frames to the testable device through. Basically, this means that whenever there is an idle time in communication media (apart from InterPacket Gap) custom frames are sent. If any normal communication frames are received during transmission of a custom frame, it will be buffered to memory, allowing the custom frame to be sent completely first.

3.3.15 Drop frames

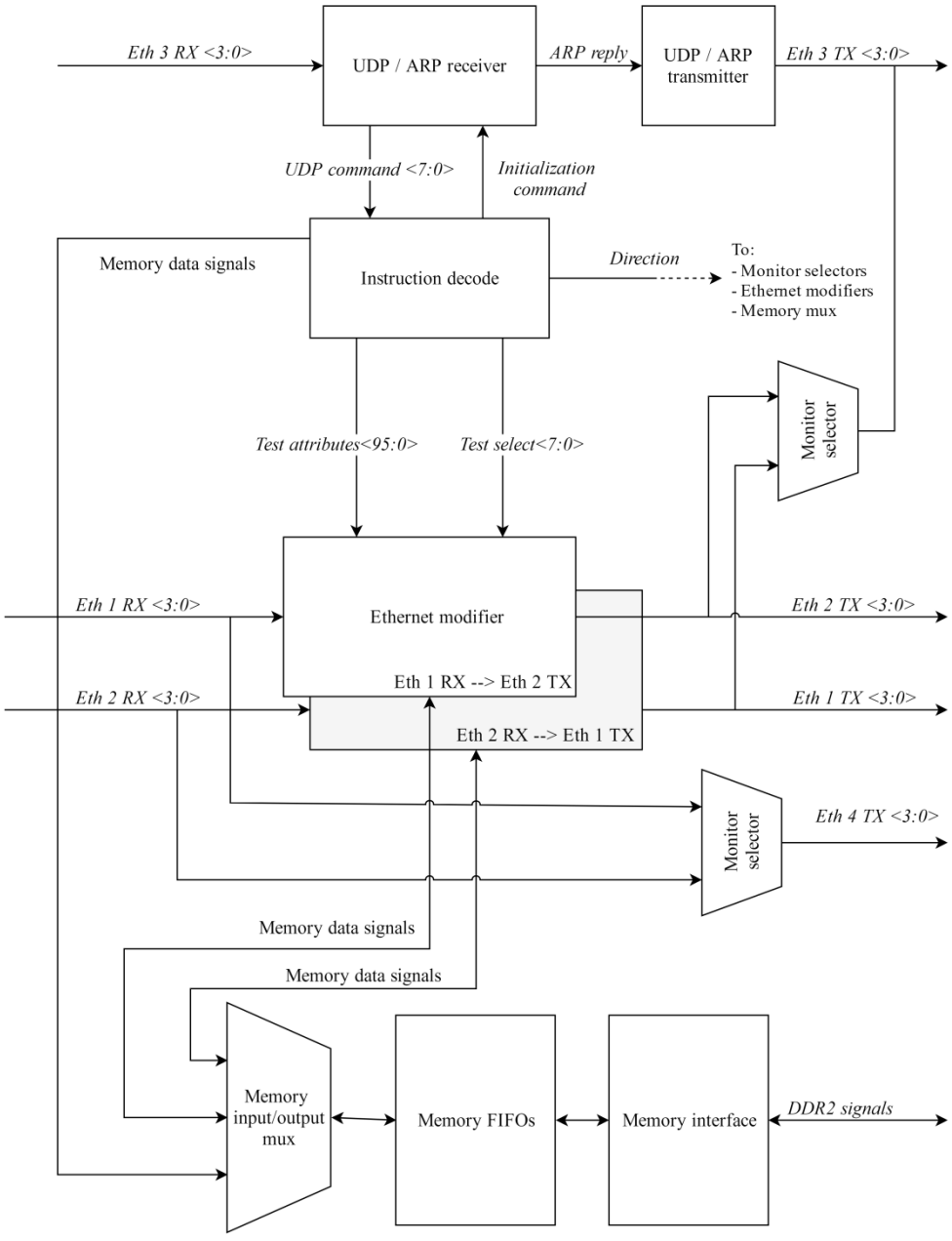
With this test, it is possible to drop one or more frames from communication media completely. The user simply sets a number of frames that will be lost, and when the test is started, a defined number of frames are not routed from the test device receive port to transmit port.

3.3.16 Invert single bit from frame

The test simulates external transmission failures that might happen due to bad cables or external interference to the communication media. Test inverts one user defined bit from a frame. The user can define which bit will be inverted, giving an offset bit count from either frame start, source address start, type start, or data start. Automatic checksum fixing is not available for this test as the purpose is to simulate normal transmission errors.

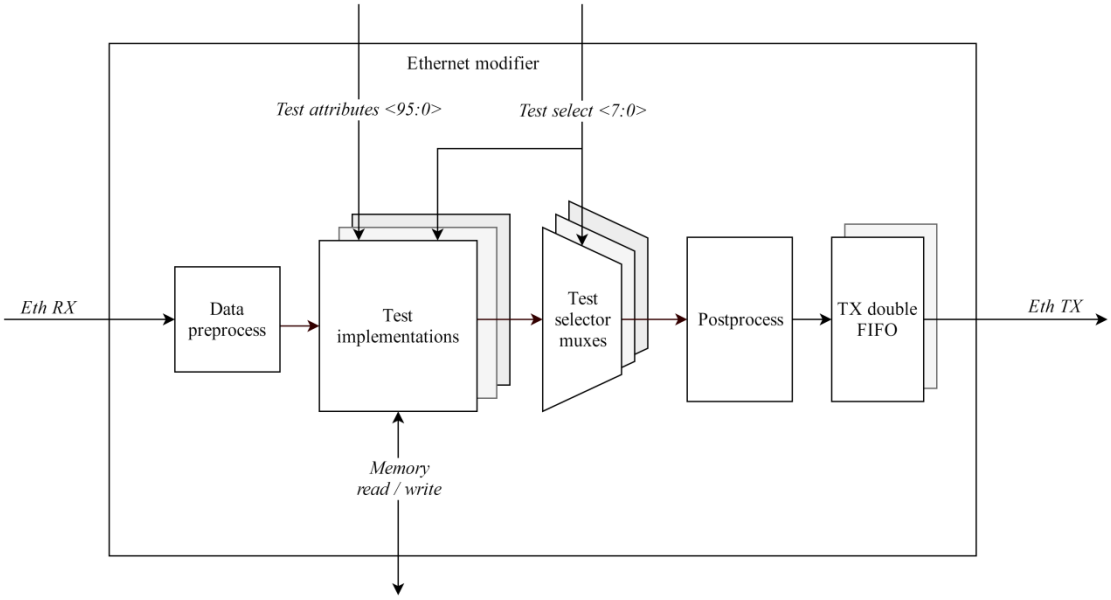
4. DESIGN

As with other software development projects, designing good software architecture for the system is often even more crucial than the actual implementation in FPGA projects. Bad architecture and design in FPGA development process increase complexity, decreases maintainability and updatability and possibly takes more resources. Picture 4.1 shows high level architecture design of the implemented Ethernet tester.



Picture 4.1 *Architecture of Ethernet tester*

More detailed view from Ethernet modifier block is shown in Picture 4.2



Picture 4.2 *Ethernet modifier block architecture*

One of the core requirements for the project was to be easily updateable with new tests and features. To add a new test to the system, one must add implementation in three places:

1. Add command parsing, detection, and control for the new test in instruction decode block.
2. Implement the actual test in a new VHDL code block. Most of inputs and outputs of the new test will probably have the same format as other existing tests, so existing ones can be used as a template.
3. Route inputs and outputs for the new test correctly using multiplexing functionality based on a command input from instruction decode block.

Functionalities around tests are purposely kept as generic as possible so that those can be used with new tests without much effort. As there are already a relatively big set of tests available, new tests can be implemented without going into too many details for the whole system, but just copying existing tests and modifying from there. Multiplexing functionality also helps to feed data to correct places with only adding new enumerated IDs to control signals of the multiplexers.

The design relies on many places to multiply a block to multiple instances. The top level "Ethernet modifier" block shown in Picture 4.1 and Picture 4.2 is defined in one common VHDL file but used twice for both directions from Ethernet port 1 RX to port 2 TX and other way around from RX 2 to TX 1. Data Transmit block is another example of this. Transmit block consist two FIFOs, allowing maximum bandwidth by allowing filling up next buffer already when the previous one is still sent. However, only one

common implementation of the FIFO is available and instantiated twice in transmit block. Further, the transmit block is multiplied to both Ethernet modifier blocks, but also used for monitoring functionality in both Ethernet port 3 and 4. Thus the system uses a total of 9 instances of a transmit FIFO, which can be used concurrently and independently from each other.

A detailed description of individual blocks will be gone through in chapter 5. Before that, we will take a look at an overall view of data flow and functionalities in each block using a couple of example tests cases described in chapter 3.3. But first, let's familiarize ourselves with a communication protocol used to control and exchange data to and from test device using the software in user PC.

4.1 Communication protocol

Communication between user PC with control and command software and test device is handled by sending predefined control protocol packets on top of UDP/IP protocol. Test device protocol packet is encapsulated into UDP frame data field. Control protocol maximum data size is deliberately kept under 512 bytes to avoid UDP frame fragmentation, as reconstructing frames on test device would increase complexity in UDP support implementation. Every command and control protocol frame is started with main protocol ID followed by a unique test or command ID. Main protocol ID is used to identify test device protocol from other possible UDP frames coming to test device and identifying possible separate test device protocols in case of features implemented in future needs such support. Test ID is used to distinguish individual tests or commands from a complete set of tests. Additionally, a punctuation mark is used to divide commands into sections and separate command attributes from each other. These three basic components are defined below.

- Main protocol ID, 9 bytes, a constant text */ETHTEST/*
- Test ID, 1 byte, an enumerated value, range 1-255
- Punctuation mark, 1 byte, constant character /

Before the device can be used, it must be initialized with valid MAC and IP addresses and port number. If device MAC address is not known by user PC operating system, MAC address inquiry is sent to the related network. If user PC does not get a reply to the ARP request, user PC refuses to send any UDP frames to the IP address and port because the underlying MAC address remains unknown. On the other hand, the test device allows dynamic setting of these addresses and doesn't have the information when booting up the device. The situation is handled using Ethernet broadcast feature with MAC address FF:FF:FF:FF:FF:FF and IP address 255.255.255.255. Test device will identify address setting protocol frame by reading out broadcasted data and identifying if the received frame is initializing frame. Address setting protocol is shown in Table 6.

Table 6. *Address setting frame*

Octet	*Description*	*Bytes*
0	Protocol main ID	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Test device IP address	4
15	User PC IP address	4
19	Test device port	2
21	User PC port	2
23	User PC MAC address	6
29	Test device MAC address	6

After addresses have been set and saved to device memory, the device is able to reply ARP request and receive command UDP frames sent explicitly to the device. As an example, let's have a look at communication protocol frames for frame source MAC address modifying test and send custom frame from file test.

Table 7 *Source MAC address modifying command*

Octet	*Description*	*Bytes*
0	Protocol main ID	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for conditional modifying	6
19	Punctuation mark '/'	1
20	Amount of frames to modify	1
21	Fix checksum	1
22	New frame source MAC address	6

In all test commands beginning of the structure from protocol main ID to punctuation mark after conditional modifying MAC address is kept same. Modify direction defines the direction of the frame to modify, whether it travels from Ethernet port 1 to 2 or another way around. MAC address for conditional modifying will check received Ethernet frames destination addresses and modify source address only after destination address match is found. If conditional modifying is not used, address bytes are set to zeroes. After constant part comes test specific attributes. Punctuation mark is not needed here as protocol itself defines a number of bytes explicitly. In fact, punctuation mark is not mandatory in current implementation at all, but it is still kept in the protocol for clarity and debugging reasons.

Table 8. *Send from file command*

Octet	*Description*	*Bytes*
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for conditional modifying	6
19	Punctuation mark '/'	1
20	Frame to send	n

As seen in Table 8, a command to send a frame from a file is otherwise kept identical, but attributes section is different. Because UDP frame is ended right after last bit of user defined frame, the protocol doesn't need to define even length of the frame to send, FPGA implementation can detect the end by other means.

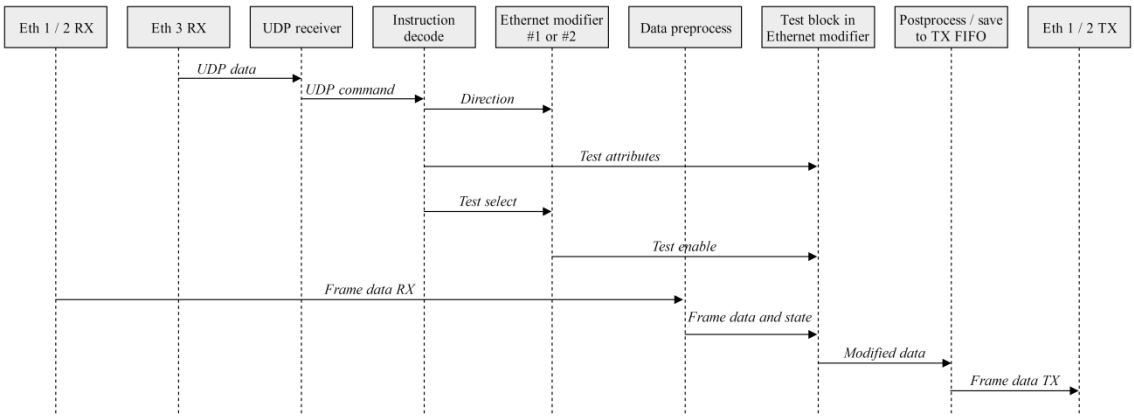
By using these three commands, let's check next how data flows in different parts of the design. A complete description of all available commands can be found from Appendix A.

4.2 Data flow

When a frame arrives from Ethernet 3 RX port, UDP / ARP receiver will investigate its contents. It's beneficial to note that data is investigated byte by byte while it arrives from PHY layer. Offset position from Ethernet start frame delimiter of the frame is tracked in a finite state machine. For example, the frame can be detected to consist IPv4 protocol by checking if Ethernet type field contains value 0x0800 when offset from start frame delimiter is 12 bytes and then further examine IPv4 header for protocol field if it contains number 17 defined for UDP protocol. In case UDP protocol is detected, UDP data bytes are passed on to instruction decode block.

Instruction decode block checks if the data contains test device protocol contents using the main ID. After successful detection of protocol main ID, test ID is checked. Depending on the test ID value, used blocks and data flow between them can differ greatly. In case of address setting command, *initialization command* signal is passed to UDP receiver and transmitter blocks to save remaining data containing MAC, IP and port information from ongoing protocol frame to on-chip memory. Data is saved to memory in a correctly arranged format for UDP/IP protocol sending, meaning sending an outgoing UDP frame can be done by reading memory byte by byte with increasing address pointer and filling up needed information on the fly when needed.

If looking at source MAC address modifying frame, data flow is completely different. Data flow principle for such test can be seen from Picture 4.3.



Picture 4.3 *Data flow for source MAC modifying test*

After finding source MAC modify test ID from protocol, instruction decode block saves the test ID to test select signal, which is then passed on to Ethernet modifier blocks. Modify direction is checked individually and right modifier where to pass input data is selected with multiplexers based on the selection. Other attributes like MAC address for conditional modifying checking and the new source address is passed on in attributes bus to Ethernet modifier. When test ID arrives Ethernet modifier block, an enable signal for related test block is asserted. Needed data inputs and outputs are also directed to the appropriate test block with multiplexers using test ID as selector signal. Input data is directed through the test block, which handles replacing correct bytes from the source MAC address fields with new ones got from test attributes bus. There is also a helper block in pre-process working concurrently with test blocks giving information of ongoing frame state, so MAC source address modifying block doesn't have to find the correct state by itself. Ethernet modifying block also provides some commonly used post-process helper functionality like CRC calculation and delaying frame sending. Frames have to be delayed e.g. when correcting frame check sequence, as frame ending isn't known before it actually ends, thus making a change of last four bytes impossible if the bytes were already sent when frame ending is noticed. Controlling of these helper functionalities are handled by test blocks itself by using output control signals.

On the other hand, there are some tests containing such amount of information in the command itself that it has to be written straight to memory first before starting actual test. One of these is sending a custom Ethernet frame from text file given by the user. In this case, instruction decode block transfers data from command frame straight to the memory controlling block. As memory can be used from Ethernet modifier blocks as well, an advanced multiplexer is needed to synchronize data input clocks to memory FIFOs based on where the data is coming from. In practice, this means that when saving data from the frame, the clock used for write side of FIFO is same than in instruction decode block (Eth 3 RX), but when data comes from either of Ethernet modifiers, Eth 1

RX or Eth 2 RX clock is used instead. When changing clocks FIFOs are reset, but change happens naturally only at times when there can't be any data in the FIFOs. When the test block itself in Ethernet modifier block notices that it is possible to send the user defined frame, it will set read enable signal to the memory interface and data is fed through test block to TX FIFOs.

5. IMPLEMENTATION

Implementing system design described in chapter 4 means writing working, synthesizable VHDL code for each logic block and defining interconnections between them. As the target of this thesis was to work in Fast Ethernet networks with 100 Mb/s speeds only, meeting timing requirements did not require much special attention. Also, even though the design did grow relatively big, different parts of the system was possible to be divided into different input clocks, thus avoiding clock skew problems introduced by long clock wiring interconnections. However, as there is a total of eight different clocks used in the system, synchronizing data between those became a crucial part of the work. It was decided to use vendor specific FIFOs for data synchronizing, as those are easily configurable and highly optimized building blocks, but still common and available for all FPGA vendors, not just Xilinx FPGA used here. The FIFOs can be easily replaced with ones from another vendor or with a custom implementation if needed in the future.

Generally, design choices described previously are not hardware description language specific and neither are the implementation specific designs. In general, this chapter is not to describe the implementation in code level, but rather to discuss implementation aspects in terms of ideas, concepts, and algorithms. The design can be divided roughly into three parts, communication handling, Ethernet frame modifying and monitoring, and memory controls. Let's have a look at implementation practices of these parts next.

5.1 Communication

5.1.1 UDP/ARP receiver

In order to receive commands and provide a user interface for the test device implemented on FPGA, it is needed to provide means of communication for user PC. Probably the simplest way to do this would have been to send raw Ethernet frames with a custom communication protocol, which would have eliminated the need for handling higher level protocol stacks like IP and UDP and related addressing. However, for example Microsoft Windows operating systems don't allow sending raw Ethernet frames, at least without implementing custom Ethernet network card drivers. Using such custom drivers would have compromised portability and usability of the system, and would have been compromised usage in such enterprise environments where users do not have administrator rights for their computers. The second easy way would have been to use broadcast for sending commands, which would have again avoided using of exact addressing. The drawback would have been that sent command frames could have

affected other devices in the network and using multiple test devices in the same network would still need some identifier initialization for each and every test device individually. Thus implementing full UDP protocol support for device communication was found to be the best solution, even though it required more work.

As described in chapter 4.2, UDP receiver and transmitter uses FPGA synthesizable memory block containing IP and UDP protocol headers for addressing information filled up at initialization phase. When address setting frame arrives, memory data in predefined addresses is filled up with MAC, IP, and port addressing information. Memory is defined using Xilinx provided distributed memory generator implementing single port RAM with 8-bit data width and 64 byte depth. We will call this memory block as protocol memory in the following text.

The UDP receiver is simply put an infinite state machine. When a new frame is received, the frame is checked byte by byte against known IP/UDP protocol frame structure. If valid information is found from incoming frame compared to device addresses initialized to the protocol memory, a UDP frame addressed to the test device can be identified. An ARP request addressed to the test device can be found in a similar way, with separate state machine steps after finding ARP type definition from Ethernet type field.

When UDP frame addressed to the test device is found, receiver passes on data from UDP data field to client parts of the system with appropriate signals set to tell about incoming data.

5.1.2 UDP/ARP transmitter

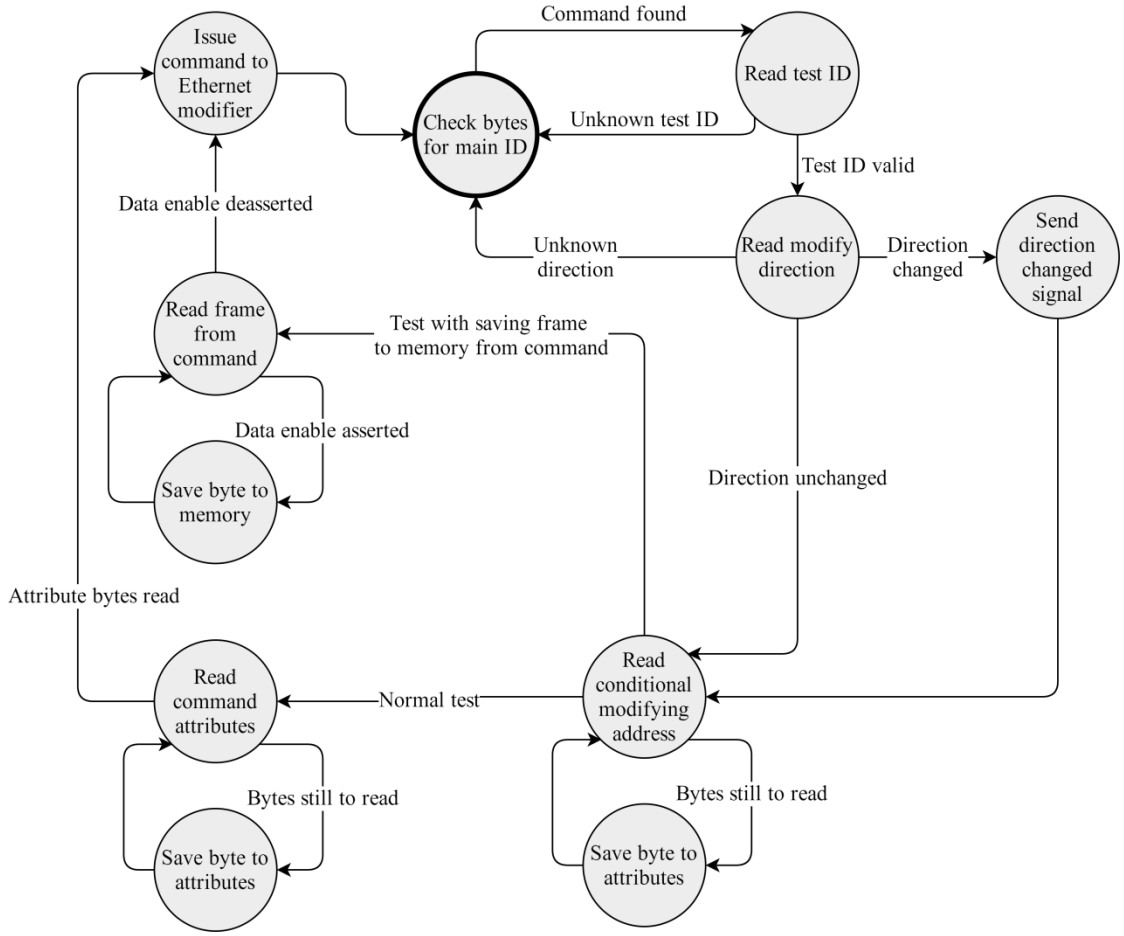
The device can also send IP/UDP protocol frames to user defined destination MAC, IP and port addresses. One very useful functionality is to enclose Ethernet frames on the monitored Ethernet bus inside UDP protocol to send those to user PC when monitoring ongoing communication. This is useful when trying to debug misbehaving communication devices losing frames due to e.g. frame check sequence checking. Rerouting the frames directly while monitoring might hide broken frames because user PC Ethernet PHY device discards those to the first OSI layer already. UDP transmitting could be also used in the future to send acknowledgements to user PC software of executed tests but that feature was left to future development projects due to time limitations. During the development, UDP frame sending was used laso for debugging purposes by sending information about the system state in failure situations.

UDP frame is sent by reading protocol memory byte by byte for Ethernet, IP and UDP headers. Fields that are not static like length and checksum fields in IP protocol header are filled up from registers during sending. ARP reply is sent in state machine similarly with the receiver. The receiver sets ARP found signal for transmitter after finding valid

ARP request, after which ARP reply is done by the transmitter by reading out appropriate addresses from protocol memory and setting those to a valid format for ARP reply.

5.1.3 Instruction decode

Instruction decode block receives UDP data from UDP receiver in case of successful UDP frame recognition. Instruction decode block functions as infinite state machine described in Picture 5.1.



Picture 5.1 *Instruction decode state machine*

The instruction decode block constantly checks the data, trying to find the protocol main ID. If the first byte matches with the constant ID byte, the second byte is checked, followed by others until the whole ID is recognized. In case of mismatching bytes, the state machine will fall back to the first stage to check the first byte of the main ID. In case of state machine advancing all the way to the last byte of recognized protocol, instruction decode block will pass related command signals and attributes to the blocks performing the actual tests. If a direction change command of Ethernet modifying and monitoring is noticed by instruction decode block, a direction signal is passed on to multiplexers directing further commands to appropriate modifying block.

All test device protocol test IDs are enumerated to separate VHDL file defining used byte constants. The file can be then shared with both instruction decode block and Ethernet modifier block to avoid defining same data in multiple places.

5.1.4 Receive and transmit FIFOs

As receive and transmit ends of the Ethernet port used for communication have separate clock inputs, all data to and from the outputs and inputs have to be synchronized using FIFOs. In practice, it was found simplest to use one common clock for UDP receiver, UDP transmitter, and instruction decode blocks, as those are sharing data and control signals. This way received data from MII can be gathered in bytes and written to receive FIFO using Eth 3 RX clock for synchronization. After reading the data out from the receive FIFO it is synchronized to the common clock and can be used without further concerns of actual I/O synchronization. The same principle is used at transmitting end, FIFO handling data synchronization to Eth 3 TX MII clock at last possible point of data flow.

5.1.5 Reconciliation layer

In practice, there has to be the reconciliation layer before or after synchronizing FIFO layer for receive and transmit FIFOs adding preamble, start frame delimiter and frame check sequences to the data. When e.g. transmit FIFO is seen containing data by reconciliation layer, it will first append the preamble and SFD to the MII transmit data in four bit nibbles. The checksum is then calculated over the whole data bytes. In most basic for CRC is calculated one bit at a time, but using that method would require eight times higher clock frequency than used MII clock of 25 MHz. In FPGA CRC can be calculated concurrently eight bit at a time, using the method proposed by Sprachmann in [16]. This method uses simple one bit linear feedback shift register and letting FPGA synthesizer figure out needed logic gates when doing eight shifts in a loop in one clock period. This approach is implemented in a separate VHDL entity which is used in Ethernet modifier block as well.

5.2 Memory control

Before going to Ethernet modifier block, which handles the actual tests execution, let's take a look at the memory interface. The memory interface handles data writing and reading to the external DDR2 memory chip. Xilinx does provide a readymade memory interface for the used MT47H128M16RT-25E memory chip, which does greatly reduce the amount of development time regarding memory control. Still, there was much to do before there was easily usable and reliable memory control interface to be used by both Ethernet modifier and communication.

5.2.1 Memory Interface Generator (MIG)

Memory interface provided by Xilinx is called MIG and is mainly based on using FIFOs as data sharing and synchronization between the memory chip and user blocks. It provides a relatively easy to use wizard to create a configurable multi-port interface for a memory chip. A very useful configuration option for the test device was to provide separate read and write ports, which made read and write operations possible to be made at same clock period even though the actual memory chip is a single port only. This is possible because data rate of Fast Ethernet network limits maximum data rate to 100 Mbit/s, so simultaneous writing and reading for Ethernet receive and transmit purposes requires twice that, 200 Mbit/s. Still, DDR2 memory with a clock speed of 400 MHz and a data width of 16 bits provides a theoretical maximum of 6400 Mbit/s data rate, which should provide enough bandwidth for Gigabit Ethernet as well. Xilinx provided memory interface handles data writing and reading to and from user accessible FIFOs to the memory, so the user doesn't have to worry about e.g. memory bank addressing or other memory signaling.

The interface provides a separate command FIFO and data FIFO for both read and write ports. Command FIFO is used to give byte address of start byte and a number of bytes to read or write with a single command. When writing, data has to be first written to the write port FIFO and then a write command is given with address and length information. With reading data flow is inverted. First, a read command with byte address and length is given, after which memory interface fetches the data from memory to FIFO and then data can be read from FIFO by user side implementation.

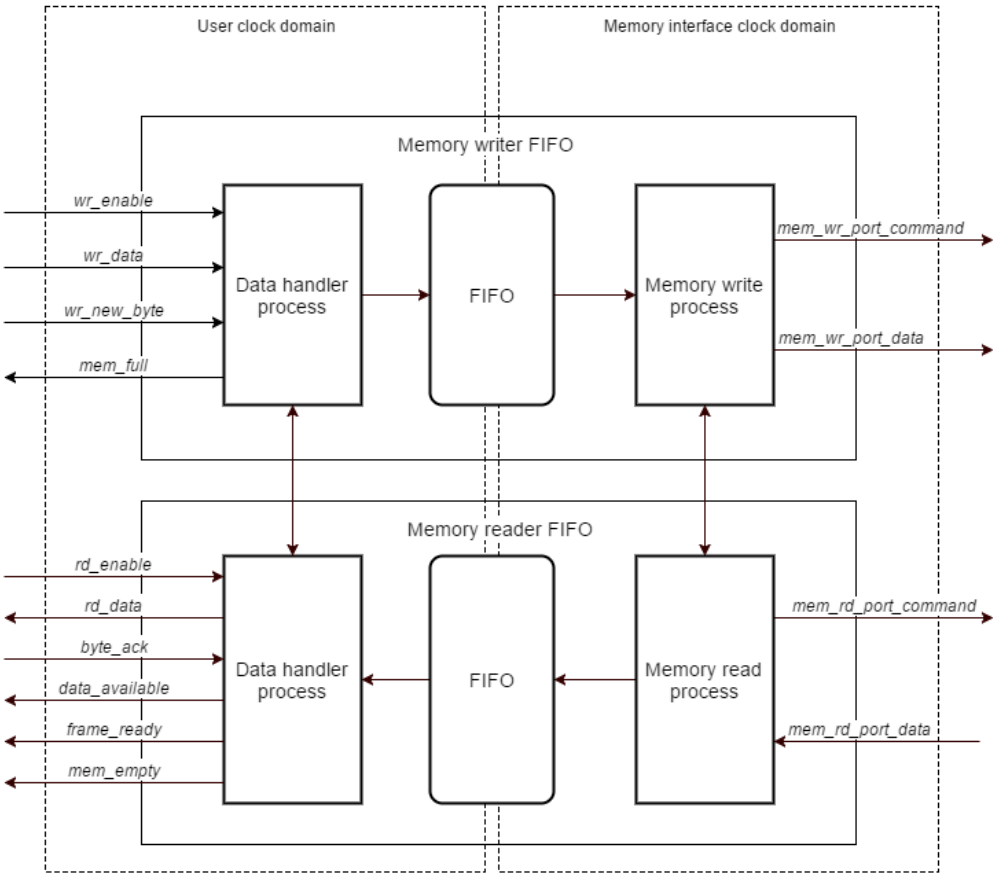
5.2.2 Memory FIFOs

Even though memory interface already uses FIFO buffers to synchronize user data to memory clock domain, it was quickly found out that for usability and implementation reasons one more layer of data buffers was needed. These reasons include:

1. Data input and output clock domain can be changed during the operational mode, by changing test/monitoring direction from Eth 1 to Eth 2 or by saving data from user command using device communication clock domain. When changing the clock, only memory FIFOs has to be reset and initialized, not the whole memory interface and memory chip as well. Because memory is calibrated after reset, which in practice might take tens of microseconds, a full reset is not the best solution in many cases.
2. Memory interface requires data and command to be given in separate clock periods, which makes using the common clock in Ethernet data handling path and memory interface path difficult. When using higher clock frequency towards memory interface it is possible to maximize both memory operations and Ethernet operations bandwidth.

3. Easier to implement separate thin layer providing memory reading and writing interface optimized for Ethernet frame data chunks.

Memory FIFOs block does not include only dummy first-in-first-out buffers to synchronize data. In fact, those are only small part of the implementation, the main reason being to provide easy to use memory reading and writing functionality for other parts of the system. Abstraction level on memory usage was taken far from general memory implementation; the user doesn't need to know for example addressing information at all. Data is rather handled as a chunks: first and last byte of a chunk is acknowledged by asserting and de-asserting write enable signal. The written chunk can be read by asserting read enable and reading data until chunk end signal is got. The whole memory is a sort of a big FIFO itself. Picture 5.2 illustrates a simplified architecture of memory FIFO block.



Picture 5.2 *Memory FIFO block architecture*

When write is started by asserting *wr_enable* and *wr_new_byte* signals and setting data byte to *wr_data* bus, write data handler starts to count bytes that are received during *wr_enable* signal is kept high. *Wr_new_byte* indicates new byte in the *wr_data* bus, so it's not necessary to have new data on the bus every rising clock edge. This is mandatory in Fast Ethernet environment because at used 25 MHz frequency we get only 4 bits of

data every clock period, but internally data is handled as bytes, resulting one full byte every other clock period.

Memory interface provides data bus width of four bytes, thus data handler gathers four bytes of data before putting it into FIFO. With the first byte of the chunk received from FIFO by the memory write process block, first four byte memory chunk is jumped over and the address of the jumped space is saved. This space is reserved for a number of bytes information got after the whole memory write chunk is received. A number of bytes information is saved to a first memory slot of the chunk because otherwise reading side implementation would turn out impossible without a separate number of bytes memory. How would reading side block know which is the number of bytes information if it is not known to be the very first byte?

When data chunk receive stop is noticed from de-assertion of the *wr_enable* signal, the data amount is transferred to memory write process through separate synchronizers. The actual amount data is injected straight to memory write process side registers, but reading is delayed until two-FF synchronized edge triggered enable signal is noticed on memory write process side, as instructed by A. Perttula in [15]. Memory write process will get information of chunk stop with byte amount signal, no need for separate stop signaling in that end.

Reading side implementation follows more or less same principles than writing, just inverted. With a *rd_enable* signal, data handler will pass on new chunk read request to memory read process using two-FF signal synchronizer. User side implementation is left waiting for data with *data_available* signal cleared. Memory read process will read first bytes from memory, containing a number of bytes in the chunk. Rest of the data will be put to FIFO. Data handler process will notice the availability of data in FIFO and read the data to *rd_data* bus together with the asserted *data_available* signal. Every byte read is acknowledged by user side implementation with a *byte_ack* signal to allow slower reading than the clock frequency. As memory read process is kept at higher data rate than data handler process, it can be assumed that chunk is stopped when FIFO is empty, avoiding separate stop signal synchronizers. Lastly, *frame_ready* is set to signal finalized chunk reading.

Because writing and reading data handler process is kept at same clock domain, it is possible to exchange data and control signal between them effortlessly. One use case for this is providing memory empty and memory full signals to the user. The total amount of bytes in the memory is kept synchronized constantly between these two processes, providing empty and full information at the same time. Same is true with memory write and memory read processes, which can share data between them. This is especially useful with a provided feature of starting a read when a data chunk is only partially written to memory. When first bytes of data is written to memory, memory write process signals read process that there is data in the memory and reading can be started. When data

writing is finished and amount of data in the chunk is known, the information can be passed straight to reader side and reader can then compare received byte amount got from memory and byte count that should be received.

5.2.3 Memory input/output multiplexer

Purpose of memory input/output multiplexer is to redirect inputs and outputs of memory FIFO to and from right block in tester device. The user of the memory FIFO can be either one of Ethernet modifiers or instruction decode block, and connections between the blocks have to be runtime configurable accordingly. In order to change input, also related clock signal has to be changed. Changing clock input is possible, but to do it reliably, it has to be clear that no data will be sampled during clock change as change might result in ripples in the clock signal. Due to uncertainties of clock change, a reset is set to associated memory FIFOs and the reset is synchronized to the new clock after deassertion.

5.3 Ethernet modifier

As stated before, Ethernet modifier block handles the actual test functionalities and Ethernet monitoring. It will handle receiving of incoming data in the Ethernet bus, modify the data if needed based on user selection and transmit the data further to the same Ethernet bus. When there is no ongoing test execution, data flow through the device unchanged. However, a slight delay is still introduced to the frames, mostly because data has to be synchronized from receive clock to transmit clock. Next chapters will describe implementation aspects of Ethernet modifier block.

5.3.1 Data receiver and pre-process

With Fast Ethernet, a four bit nibble is received at every 40 ns period (25 MHz frequency). For easier handling, the first thing to do is to gather every two nibbles to a byte. In practice, this does introduce one clock delay to every frame already and could be optimized away if data would be handled as nibbles instead. When receive is started, amount of preamble nibbles is not known beforehand, because physical layer implementation might not deliver every 15 nibbles of preamble through MII to user implementation. This was crucial notification during implementation when it was first tried to just redirect all data from RX to TX port, quickly noticing that every now and then a frame was lost from the bus. Problem was that most physical layers do not require whole 15 nibbles of preamble when sending data, but when the length is shortened too much, the frame might get corrupted in the process. As a number of preamble nibbles seen on the receiving end of MII interface change drastically, data receive to bytes is synchronized using start frame delimiter nibble and the preamble is thrown away. This will introduce the biggest delay of 15 clock periods, as when transmitting the full pre-

amble will have to be added again to the beginning of every frame. There is again a possibility of optimization if received preamble nibbles were routed to TX as well and TX block would add only needed amount of nibbles to the frame for full preamble sending.

After converting received nibbles to bytes, the data is routed through frame state handler block, which keeps track of current state of the frame and provides the information to other parts in the modifier block. When data is investigated, it is needed to be saved to a register, which will again introduce one clock period delay to the frame. The delay is still insignificant compared to the benefit of one common state handler block that can be used by every test block, without implementing the same functionality in every block and increasing amount of used LUTs with every added test in the future.

5.3.2 Test selector multiplexers

Every output from test blocks needs to be directed through multiplexers, which will select correct output routing to a single transmit bus. The selection is based on *test select* signal coming from instruction decode block. As an example, transmit FIFOs need to get data from somewhere, and based on user selection it can be either directly and unmodified from pre-processing or from any test functionality block. Writing to memory is also possible from multiple test blocks, but luckily only one is allowed at a time and can be easily handled with multiplexers as well. Inputs can be directly routed to all test blocks without any multiplexing or such because it does not matter if other blocks also get hte data meant for somewhere else if it's not using it. Only a specific block gets a test enable signal asserted at a time and only that block will use the input data and modify it. The enabled test block is selected with simple switch - case conditional checking from *test select* signal.

5.3.3 Post-process and test supporting features

Before checking into the actual tests, let's first have a look at post-processing and test supporting functionalities, which are common for all test blocks. These features are implemented separately so that those aren't needed to be included in many separate test blocks individually. These features include:

1. Frame check sequence (FCS) calculation for modified frames and replacing original with a new one when selected.
2. Delaying frame sending and buffering the data by given amount of bytes.
3. Swap delayed and buffered data to a given chunk of data, if not possible to do in test block itself.

These features are tied together in a sense that delaying of the frame is needed to be done always when using either of the other two. If the frame is modified and FCS is wanted to be modified to match the new data, delay of at least four bytes is needed. This

is because it is impossible to know when the frame stops after the very last byte coming with de-asserted *RX valid data* bit. At this point, it would already be too late to replace four last bytes of the frame containing original 32-bit CRC, as those bytes are sent already.

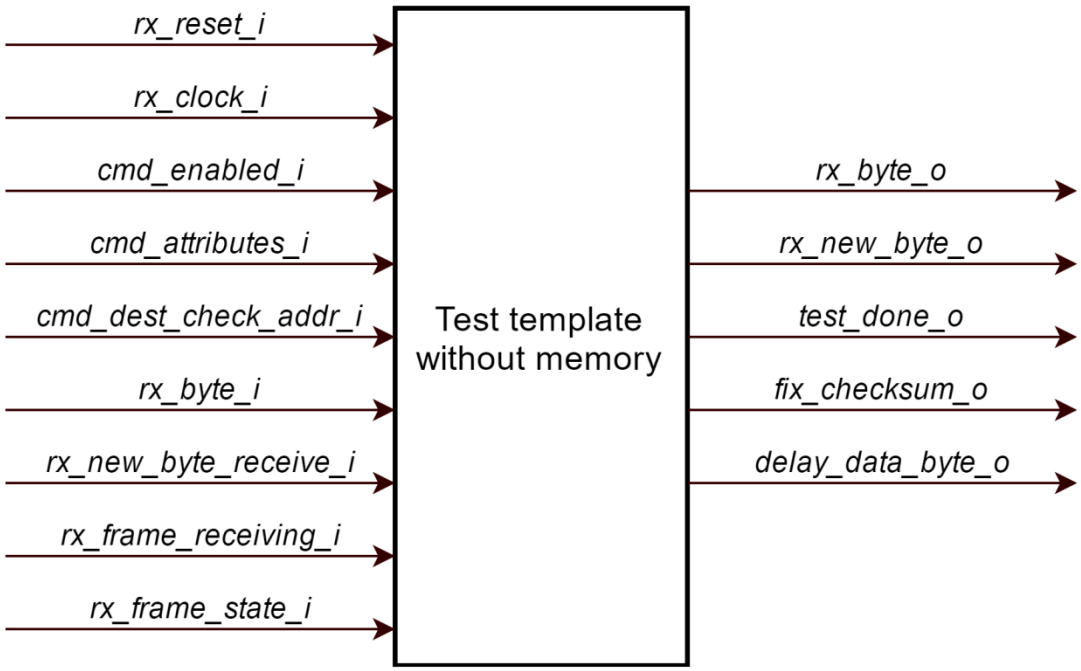
Swapping delayed chunk of data is currently used only on frame check sequence modifying test described in section 3.3.6 because it would be impossible to change the data in respective test block because of same reasons just described above. This feature would be useful for example when using the conditional modifying feature with destination MAC address modifying test also, but that part was left for future projects.

Delaying of the frame can be useful individually as well for example when drop frame test described in section 3.3.15 is used with conditional frame modifying feature. In order to know if the frame should be lost or not, destination MAC address has to be checked completely. In order to be able to send the complete frame in case destination address checking returns false, the first six bytes needs to be delayed and buffered. In case of destination address checking would have returned true in above case, buffered data can be just discarded.

A crucial part of the post-process functionality is also to route the data to be sent to either of TX FIFO buffers. Usage of the two FIFO buffers are changed after every sent frame, so filling next buffer can be started already when reading of previous one is still ongoing. This ensures maximum bandwidth usage even though synchronizing data from RX clock to TX clock takes some time. When saving to buffers or switching used buffer, it is needed to be checked if data comes from delaying buffer, from some of the test blocks or straight from pre-process, which resulted in surprisingly complex conditional checking clauses for a seemingly simple feature.

5.3.4 Test blocks

Test blocks will handle modifying the frame or command other parts of the system how and when to carry out needed changes to the ongoing frame. Every test is an individual implementation, but most of them can be derived from two common templates with appropriate modifications here and there. The templates can be divided into two categories based on if the test block uses memory or not.



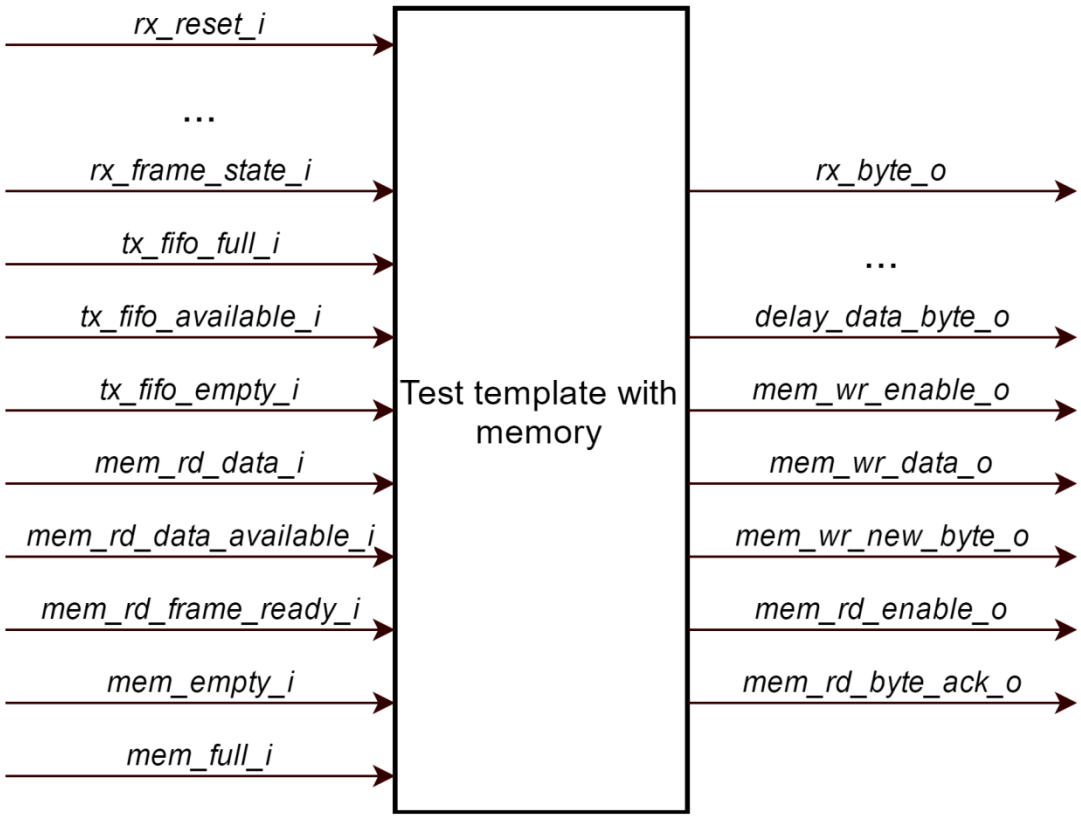
Picture 5.3 *Test template block without memory usage*

Picture 5.3 shows basic inputs and outputs for a test block that does not use memory. These input and output connections are common to all test blocks, but individual tests can add some I/Os or leave some I/Os unused if not needed. Even though some connections might not be used, the difference in FPGA resource usage is so insignificant that using a common template can be justified.

Let's go through usage and purpose of the connections shown in Picture 5.3. *Rx_reset_i* and *rx_clock_i* are self-explanatory, reset given at system startup and clock signal carrying 25 MHz Fast Ethernet clock. *Cmd_enabled_i* is asserted by test selector multiplexers, enabling a test in a test block. Usually, this means that next starting and incoming frame should be modified in a way that the test block is implemented for. *Cmd_attributes_i* gives additional information about the user configurations for the test, for example how many frames should be modified or which byte in the frame should be modified. *Cmd_dest_check_addr_i* contains the address for destination MAC address for conditional test enabling and should be set to all zeroes if not used. *Rx_byte_i* carries the actual data from receiving Ethernet port, in one byte chunks. With Fast Ethernet one clock period carries only four bits, which is gathered then to bytes, thus asserting *rx_new_byte_receive_i* signal every other *rx_clock_i* period. Handling bytes instead of nibbles is still easier as nibbles from MII needs endianess swapping and also makes possible to upgrade the system to Gigabit Ethernet in the system. *Rx_frame_receiving_i* is asserted while frame receive is ongoing and can be used for detecting frame start and stop. *Rx_frame_state_i* contains enumerated value giving information of current frame state if the received byte belongs to for example destination MAC address or data payload.

Outputs of the test blocks contain *rx_byte_o* bus *and rx_new_byte_o* bit, giving out either byte and new byte information straight from input or some way modified. *Test_done_o* is asserted when the test is over, deasserting *cmd_enabled_i* signal in test selector multiplexer block. *Fix_checksum_o* and *delay_data_byte_o* are control signals for post-process functionalities described in section 5.3.3.

In addition to signals shown in Picture 5.3, test blocks that use memory have signals shown in Picture 5.4.



Picture 5.4 *Test template block with memory usage*

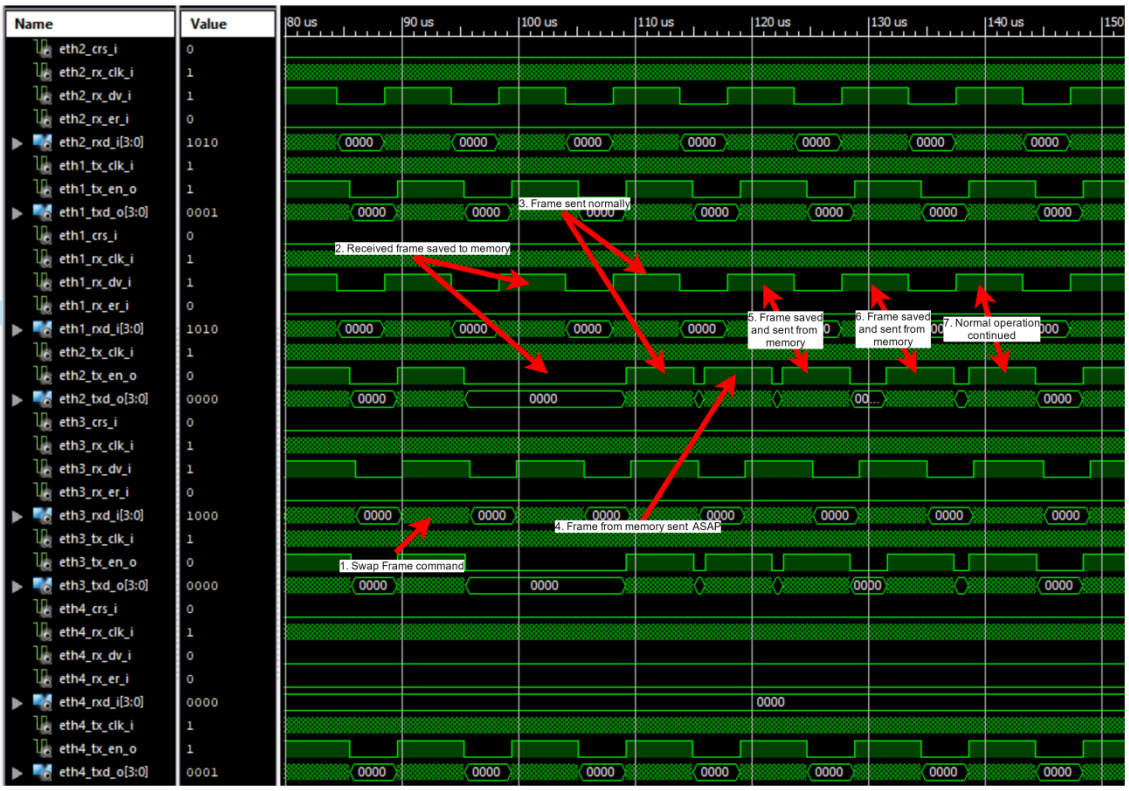
As tests that use memory usually inject complete frames into Ethernet bus, it is mandatory to have status information from transmitting FIFOs for flow control purposes. This is because memory works with higher data rate than used Ethernet speed. *Tx_fifo_full_i* tells if there is no more space available in the FIFO meaning that reading data from memory to Ethernet transmit bus has to be stopped until space is again available in the FIFO. *Tx_fifo_available_i* is used when starting to read a new frame from memory as it has information if at least one of the transmit FIFOs is completely empty. *Tx_fifo_empty_i* tells if both FIFOs are empty and used usually with *mem_empty_i* signal from memory to check if the test is completely done and can be stopped. Signals with *mem_* prefix go straight to memory FIFO block. We won't go through those here anymore, as those were already described in section 5.2.2.

Internally test blocks are relatively simple VHDL programming, with checking incoming bytes at *rx_clock_i* rising edges and swap data to a new one when certain conditions are met. Thus it won't be beneficial to go through internal implementations in more detailed view. It is good to notice that with templates there are more reusable parts of VHDL than just the interfaces. When developing new tests in the future, it can be beneficial to check existing test implementations for possible re-usable parts of existing implementations. For example, most of the tests using memory functions internally same way, having a state machine to first write something to memory, then do something in between and lastly read the saved data from memory to Ethernet transmit while buffering possible incoming frames to memory at the same time. Another highly re-usable part is a timer implementation, which is actually a separate generally usable sub-block. The user of the block can give the amount of milliseconds after when to assert a timer done signal. A test block can thus set a timer and do testing while waiting for the signal, not needing to handle time calculation by itself. Time calculation is based simply by calculating the number of clock edges of a known clock period.

6. TEST RESULTS AND OBSERVATIONS

Testing of the implemented device was done in the simulator and in real platform connecting the device between two computers. The simulation was implemented with a top level test bench design consisting the top level system design entity. This test bench was named *eth_tester_top_tb.vhdl*. The test bench contains two processes, one sending Ethernet frames to Eth 1 RX and Eth 2 RX ports to have modifiable data to the tests and other sending command frames to Eth 3 RX port controlling the tests. *Eth_tester_top_tb.vhdl* also contains simulation implementation for memory from Xilinx. This means that it is also possible to use external memory in the simulator, which was very beneficial during the development. These things mean that when designing hardware platform using FPGA chip, it is crucial to check if there are ready made support from FPGA provider to used peripherals and if possible, select peripherals based on findings. Implementing simulation for external memory would take a great amount of time otherwise.

Usually, VHDL test benches are made in a way that those can be run automatically and the tests return pass or failure based on inputs and outputs given by test bench to the system under testing. This time it was decided to cut corners slightly in this part, test bench only feeds inputs, and outputs were investigated manually by a developer from simulation waveforms. The decision was made based on the number of different features and tests combinations together with highly real time nature of the system. Implementing automated tests for different data rates and different test configuration combinations would have made the timings and condition checking highly complex on the test bench. Not saying it would have been unnecessary, automated test benches have been proven very useful in FPGA development process. This time the actual implementation of the system just took so much time by itself that something was needed to be left out. Still, all input frames are defined as named constants and can be easily handled in the test benches if the test were to be automated in the future.



Picture 6.1 *Swap frame test in the simulator*

Picture 6.1 shows an example of frame order swapping test described in section 3.3.10 as seen in ISim FPGA simulator. Red arrows show different stages of the test seen in top level Ethernet receive and transmit ports. As seen in stages 5. and 6., incoming frames during the reading of the memory are buffered to memory and sent as soon as possible. Normal operation is continued automatically after the memory has been emptied completely.

This said, simulation in this project was used more during development to verify correct operation with normal input commands, rather than full bench testing platform. More thorough testing was then performed using actual hardware, which in this case was found a more effective approach. The basic workflow for implementing a new test or functionality was done as stated below:

1. Design a command protocol for the test or feature.
2. Implement UI and command sending with Qt.
3. Send the command to the device even though FPGA implementation is not yet done.
4. Catch the command frame with PC using Wireshark. Copy the command frame data to *Eth_tester_top_tb.vhdl* as a new test command constant definition.
5. Implement the test or feature to FPGA using VHDL.
6. Verify the implementation using the command defined in step 4. Fall back to step 5 if fixes are needed.

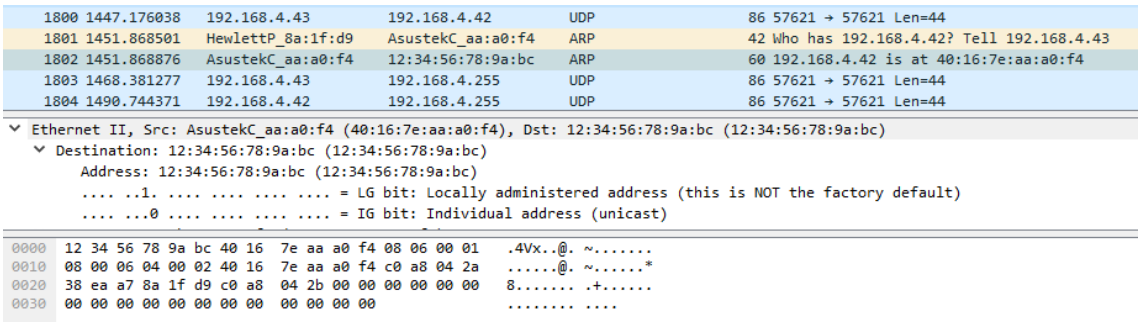
7. Synthesize the implementation and test with real hardware with different parameter combinations.

Operational behavior on the actual device is verified by using two computers with a custom application sending Ethernet frames to each other. The test device is connected in between the computers, making changes to the frames on the communication media. Changes can be then seen on the receiving computer, using a combination of the custom software and Wireshark. Wireshark is an open-source and widely-used network protocol analyzer, giving the possibility to see received Ethernet frame contents. Note that Wireshark won't see broken frames with failing frame sequence check as those are already filtered out at Ethernet interface level without using advanced monitoring feature. Oscilloscope was also considered as one possible test equipment, but triggering was found to be too hard with 100 Mb/s Ethernet using multilevel threshold-3 (MLT-3) encoding, making idling changing voltage level on communication bus as well.

Used communication test application consists host and client side software made with QT and C++. Host application sends UDP frames with the string "TEST DATA WITH VALUE: " followed with running counter value from 1 to 10. It's also possible to send user given string with the push of a button. Client software checks received counter value and reports if some frame was missing or if the frame was received but did not contain the expected string. The purpose of this method of testing was not to demonstrate real usage of the testing device, but rather a verification of functionality. Let's have a look at the tests and findings next.

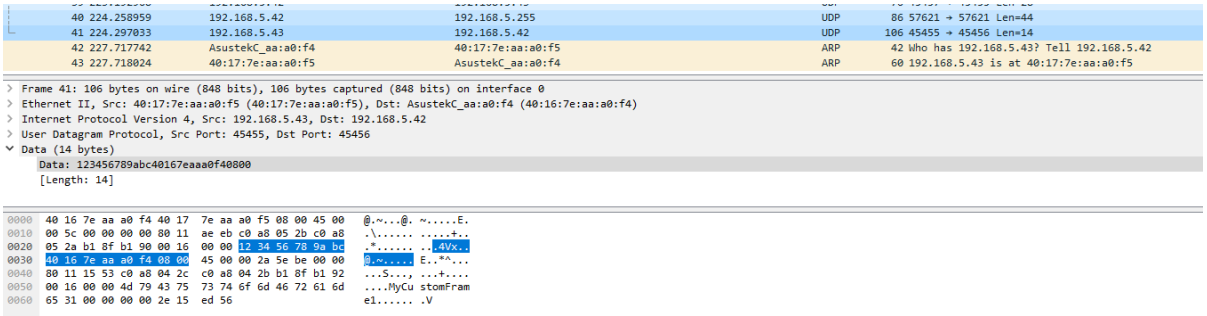
1. Preamble and start frame delimiter (physical layer)
    - Shortening preamble does not seem to have much impact on modern Ethernet interface functionality, as preamble lenght can be shortened down to three nibbles before frames are lost. Also, the rate of missed frames increases gradually when going down from three nibbles to one or zero. With three preamble nibbles about every fourth modified frame is lost, with two nibbles about every second and with one every frame is lost. Preamble field is also discussed by Spurgeon et al. [10], stating that preamble was originally designed for 10 Mb/s systems to synchronize with incoming data stream before actual payload. Higher speed systems use more complex encoding methods for frame start signaling, making long preamble unnecessary. It is still left in the standard for backwards compatibility reasons. Longer than standard 15 nibbles preamble seems not to have any difference either, other than possibly delaying the modified frames slightly.
    - Missing start of frame delimiter loses the modified frame, regardless of precending preamble length.
2. Destination address (data link layer)

- Destination MAC address modifying test is self-explanatory functionality, changing destination MAC address of a frame to a user given value. Picture 6.2 shows an example of the user changing destination address to 12:34:56:78:9a:bc from an ARP reply. Wireshark notices custom modified address from LG bit.



Picture 6.2 *Destination MAC address modified from ARP reply*

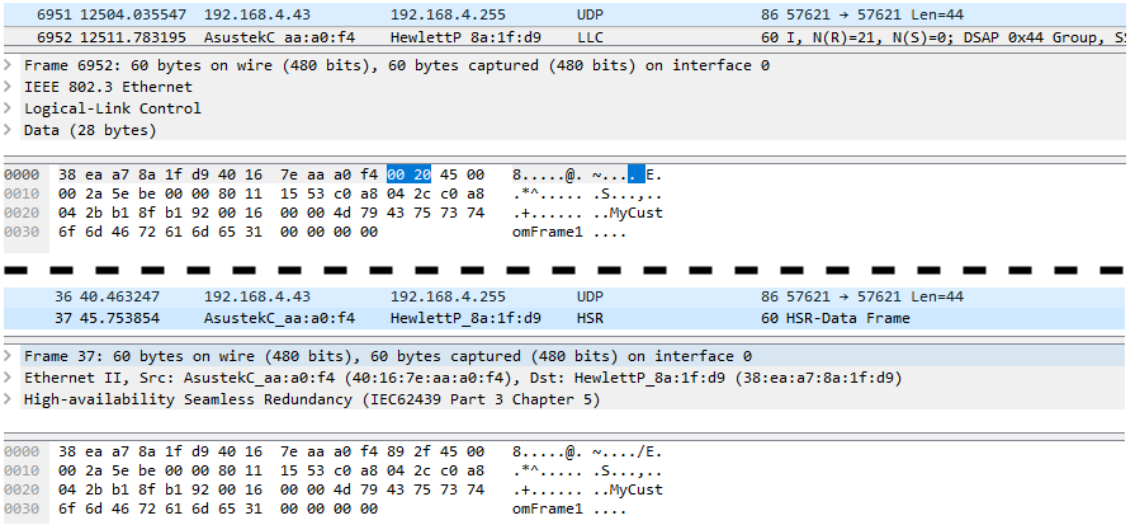
- Modifying destination address from test application send and received frames was noticed to be discarded before reaching Wireshark, even if frame check sequence of modified frame is corrected to match modified data. Modern network interface drivers or operating systems seems to check frame validity also by other means, e.g. if IP and MAC addresses do not match with known values. With advanced monitoring feature these frames can be seen as well, encapsulated to a valid frame. This feature is shown in Picture 6.3.



Picture 6.3 *Advanced monitoring with destination address modified*

3. Source address (data link layer)
    - Modifying source MAC address works similarly as destination address modifying.
4. EtherType and length (data link layer)
    - Modifying EtherType field makes receiving end device recognise frame as another type of frame than sent type. In practice, it is also possible to fool receiver to handle type field as length when value modified to be

below 1500. Picture 6.4 shows two examples of IPv4 frame type modified to original IEEE 803.2 Ethernet standard frame with length 32 (0x20) bytes and HSR frame with type field value 0x892f.



Picture 6.4 *EtherType modifying from IPv4 type*

5. Payload modifying (data link layer)
    - The payload of the frames can be changed as well. It then depends on application and operation system if the changes have an effect or not. It was seen that modified frames often gets discarded by the operating system or network interface driver before reaching test application client or Wireshark. Still, it was possible to verify the functionality with Wireshark and custom frames that were not targetted to the test application.
6. Frame check sequence (data link layer)
    - Modifying frame check sequence causes network interface discard the modified frame before reaching e.g. Wireshark. Thus the functionality is not possible to be verified with a current test setup, otherwise than seeing frames to be discarded. Verification on bit level is done on simulator only.
7. InterPacket Gap length
    - Currently, implementation supports InterPacket Gap length to be truncated to a minimum of 18 nibbles (720 ns in Fast Ethernet). This is because delay from starting a memory read to getting first bytes to transmit FIFO is too long. With test PC truncating InterPacket Gap to 720 ns did not have an effect on frame receive.
8. Send a custom frame from a file

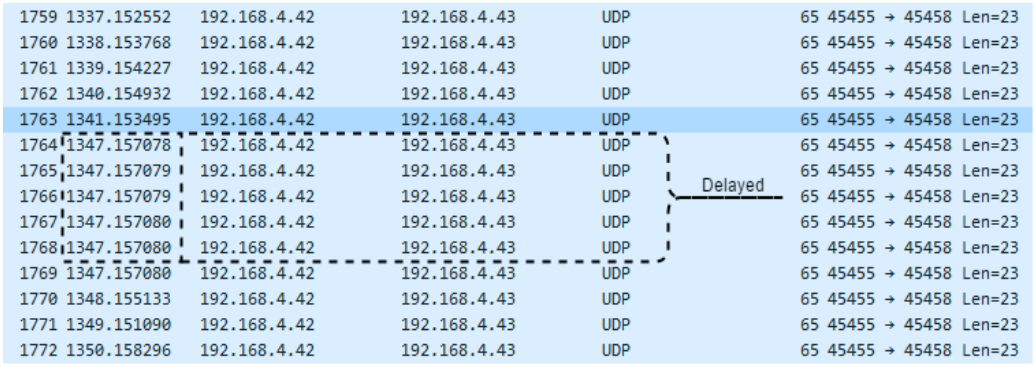
- Custom frame sending from test device works as expected. The sent frame is received normally on receiving PC and verified to contain data specified in the external test file.

9. Runt frames
    - It is possible to send shorter than 64 byte long frames using "Send a custom frame from file" feature. However, at least in test PC, Wireshark did not see runt frames, meaning those were discarded on lower layers. Length field can be reduced down to zero and the frame is still received.
10. Swap order of frames
    - Swapping order of frames works as expected, making frames come to the client application in wrong order. As the client assumes test counter in received frame increases by one after previous successfully received frame, it reports missed frames even though the frames does arrive later. Client prints can be seen in Picture 6.5.

```
OK: "TEST DATA WITH VALUE: 2"
OK: "TEST DATA WITH VALUE: 3"
OK: "TEST DATA WITH VALUE: 4"
OK: "TEST DATA WITH VALUE: 6"
Missed packet: "5"
OK: "TEST DATA WITH VALUE: 5"
Missed packet: "7"
OK: "TEST DATA WITH VALUE: 7"
Missed packet: "6"
OK: "TEST DATA WITH VALUE: 8"
OK: "TEST DATA WITH VALUE: 9"
```

Picture 6.5 *Client software prints in swap order of frames test*

11. Delay selected number of frames
    - Delaying selected number of frames test works around same principles than swapping the order of frames test. The difference is that user can specify the time in milliseconds how much a frame will be delayed. Test result prints from client show the similar result as seen in Picture 6.5, but the missed frames will be delayed more.
12. Delay all frames
    - Delaying all frames will keep the order of frames in communication media unchanged, but the frames will be delayed at user given amount of time. As the client of test software assume next frame counter value to be previous value + 1, the the test software does not report missing frames. Test functionality can be still seen with test application as well, as there is a long pause in received frames. From Wireshark delayment of 5000 ms can be seen from timestamps, as seen in Picture 6.6.



Picture 6.6 *Delaying frames for 5000 ms*

13. DoS with allowing all communication
    - Denial of Service (DoS) attack floods Ethernet media with communication packets, aiming to block services provided by the target machine. Flooding receiving port communication can have at least two concequences; taking all bandwidth from Ethernet RX port and stealing CPU resources by filling interrupt queue with inappropriate Ethernet frame handling requests. DoS with allowing all communication simulates external attack situation where other frames get through as well when there is available bandwidth.
    - Depending on the frame, flooding might or might not have a big impact on CPU load. If UDP frame data is printed on test application while Wireshark also cathing the frames, even a modern high-end CPU (Intel Core i7-4770k) takes a big hit, CPU usage rising momentarily near 100 %. Even if frames are not handled by any application, a clear rise to around 40 % in CPU load can be noticed.
14. DoS with blocking all communication
    - DoS with blocking all communication is otherwise same as above, but other frames than the ones flooding the communication are discarded on the test device. Functionality was verified using test application, making sure frames sent during flooding are lost.
15. Drop frames
    - Drop frames test is simply discarding frames full at the test device. Lost frames can be noticed with counter values of the test application.

The Ethernet media level tester was found to be working as planned. No known issues or bugs were left in the final delivery of the project. Table 9 shows a summary of the project.

Table 9 *Summary of Ethernet media level tester*

Implementation language	VHDL (FPGA), C++/Qt (PC software)
Purpose	To provide affordable and updatable low level Ethernet tester using FPGA technology and existing hardware.
Used hardware	An embedded device used in the industrial automation system. Device consists four Ethernet ports and Xilinx XC6SLX150 FPGA chip.
Size in FPGA-chip	Resource / Used / Total available Slice registers: 6,260 / 184,304 Slice LUTs: 9,033 / 92,152 Occupied slices: 3,189 / 23,038 RAMB8BWERs (block RAM): 12 / 536
Lines of code	VHDL: 13,857 C++/Qt: 3,573
Other software	C++/Qt -application to control the device

7. CONCLUSIONS

This thesis presented Ethernet media level tester design and corresponding implementation in VHDL. Main drivers were to design easy to use and updateable system with a wide variety of usage areas. There are some related devices on the market, but those are expensive and have a different set of functionalities. By selecting commonly used Ethernet protocol of UDP as an underlying command protocol, the user can write custom test scripts to run a set of tests with one command. However, this is not mandatory, as a simple UI based control software was developed also to control the test device. Control software was made with Qt libraries and C++.

Updatability was achieved through careful architectural planning. Every test was enclosed in a separate entity, but with similar input and output connections. The number of implemented tests was greatly increased during development as the development of new test did not take much more time compared to overall used hours for the project. A new test can be implemented in a single workday if no big issues arise.

Around the test entities, a set of common supportive functionalities was developed. These provide functionalities like memory usage, frames check sequence calculation and frame delaying. Functionalities were selected in a way that those are usable for all of the tests without modifications.

Even though the end result of the project was a success and all requirements were filled, some features that could be useful to users in general was left as future projects:

1. Gigabit (1000 Mb/s) support. As there was no hardware available that would support this and no requirement for the project either, it was left for future development.
2. Make the VHDL implementation FPGA vendor independent. The current implementation uses Xilinx provided FIFOs for data synchronization from clock domain to another. Also, the memory interface is provided by Xilinx. At least FIFOs could be made fairly easily as custom implementation, providing a better possibility to port the tester to devices where using Xilinx FIFOs is unfeasible.
3. InterPacket Gap length reducing test needs optimization. Minimum length could be reduced by starting preamble sending of the following frame even though transmit FIFO does not contain data yet. In case InterPacket Gap is modified, it can be also assumed that data will arrive after a certain amount of time. Possible gaps before getting actual frame data can be filled with preamble bytes, as longer than 15 nibble preamble fills IEEE 803.2 standard needs as well.

Purpose of the device is to provide means to test communication features of a device under implementation. The Ethernet tester is able to inject errors on the communication bus, which could lead to unforeseen situations without proper testing. Additional to actual tests, the device also provides highly useful features for debugging by providing a feature to reroute frames on the Ethernet media to another port and can be investigated using Wireshark or similar network analyzer software. Ethernet tester can be used also during the development of two devices communicating with each other. If for example host device is not yet available, client development can be started already if Ethernet tester is configured to mimic host device and send frames to the client. The usage does not have to be restricted to Ethernet either, as wireless devices could be tested as well if a router is connected to the device.

At the time of writing this, the Ethernet tester is already used in testing and debugging communications of the embedded automation system in relatively large Finnish software company. Advanced monitoring feature was updated to the system based on user requests and has been especially beneficial when debugging communication issues in the automation system.

BIBLIOGRAPHY

[1] Jari Nurmi. Processor Design: System-On-Chip Computing for ASICs and FPGAs, Springer, 2007, 533 p. Available at Books24x7: http://common.books24x7.com.libproxy.tut.fi/toc.aspx?bookid=30997. (visited on 5.4.2018)

[2] FPGA fundamentals, National Instruments, 2012, 3p. URL: http://www.ni.com/white-paper/6983/en/pdf. (visited on 5.4.2018)

[3] J. Rajewski, How does an FPGA work?, Embedded Micro, 2018. URL: https://embeddedmicro.com/tutorials/mojo-fpga-beginners-guide/how-does-an-fpga-work (visited on 5.4.2018)

[4] G. Eastwood, 4 critical security challenges facing IoT, Network World, 2018. URL: https://www.networkworld.com/article/3166106/internet-of-things/4-critical-security-challenges-facing-iot.html (visited on 5.4.2018)

[5] A. Greenberg, Hack brief: Hackers targeted a US nuclear plant, Wired, 2017. URL: https://www.wired.com/story/hack-brief-us-nuclear-power-breach/ (visited on 5.4.2018)

[6] ITU, ICT Facts and Figures 2017, International Telecommunication Union, 2017, 8p. URL: https://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2017.pdf (visited on 5.4.2018)

[7] IEEE Standard for Ethernet. IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012). The Institute of Electrical and Electronics Engineers, 2015. URL: http://ieeexplore.ieee.org/document/7428776/ (visited on 5.4.2018)

[8] Shuang Yu. IEEE 802.3 ”Standard for Ethernet” Marks 30 Years of Innovation and Global Market Growth. IEEE Computer Society. 2013. URL: http://standards.ieee.org/news/2013/802.3_30anniv.html (visited on 5.4.2018)

[9] Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model 2nd ed. ISO/IEC 7498-1:1994. International Organization for Standardization and International Electrotechnical Commission. 1994. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip (visited on 5.4.2018)

[10] C. E. Spurgeon, J. Zimmerman, Ethernet: The definitive guide, O'Reilly, 2014, ISBN: 978-1-449-36184-6

[11] A. Patel, Selecting Gigabit Ethernet transceivers, Electronic products, 2005. URL: https://www.electronicproducts.com/Analog_Mixed_Signal_ICs/Selecting_Gigabit_Ethernet_transceivers.aspx (visited on 5.4.2018)

[12] A. S. Tanenbaum, D. J. Wetherall, Computer Networks. 5th ed. Munich: Pearson Education Limited, 2013. ISBN: 978-0-13-212695-3.

[13] I. Grout. Digital Systems Design with FPGAs and CPLDs. Burlington: Elsevier Ltd., 2008. ISBN: 978-0-7506-8397-5.

[14] P. P. Chu. RTL Hardware Design Using VHDL. Coding for Efficiency, Portability and Scalability. Hoboken: John Wiley & Sons, Inc., 2006. ISBN: 978-0-471-72092-8.

[15] A. Perttula, Clock and synchronization, course material, TIE-50206 Logic synthesis, Tampere University of technology, Spring 2018, 50 p. URL: http://www.tkt.cs.tut.fi/kurssit/50200/S17/Kalvot/Lecture%2013%20-%20Clock%20and%20Synchronization.pdf (visited on 5.4.2018)

[16] M. Sprachmann, “Automatic generation of parallel CRC circuits.” In: Design & Test of Computers, IEEE 18.3 (May 2001), pp. 108–114. ISSN: 0740-7475. DOI: 10.1109/54.922807.

[17] M. Conti, N. Dragoni and V. Lesyk, "A Survey of Man In The Middle Attacks," in IEEE Communications Surveys & Tutorials, vol. 18, no. 3, pp. 2027-2051, 2016. DOI: 10.1109/COMST.2016.2548426

[18] Cofer, RC, and Benjamin F. Harding. Rapid System Prototyping with FPGAs: Accelerating the Design Process, Elsevier Science & Technology, 2005. ProQuest Ebook Central, URL: https://ebookcentral.proquest.com/lib/tut/detail.action?docID=270067 (visited on 17.4.2018)

[19] William Kafig. VHDL 101. Burlington: Elsevier Inc., 2011. ISBN: 978-1-85617-704-7.

[20] Peter J. Ashenden. The Designer’s Guide to VHDL. 3rd ed. Burlington: Elsevier Inc., 2008. ISBN: 978-0-12-088785-9.

[21] Spartan-6 Family Overview. DS160. Version 2.0. Xilinx, Inc. 2011. URL: http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf (visited on 17.4.2018).

[22] G. Ibanez, J. A. Carral, J. M. Arco, D. Rivera and A. Montalvo, "ARP-Path: ARP-Based, Shortest Path Bridges," in *IEEE Communications Letters*, vol. 15, no. 7, pp. 770-772, 2011.
doi: 10.1109/LCOMM.2011.060111.102264

[23] L. Kleeman and Antonio Cantoni. “Metastable Behavior in Digital Systems.” In: Design & Test of Computers, IEEE 4.6 (Dec. 1987), pp. 4–19. ISSN: 0740-7475. DOI: 10.1109/MDT.1987.295189.

[24] C. Kozierok, ARP Message Format, The TCP/IP Guide, 2005. URL: http://www.tcpipguide.com/free/t_ARPMessageFormat.htm (Visited on 22.4.2018)

APPENDIX A: COMMUNICATION PROTOCOL

IP setting and initialization command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Test device IP address	4
15	User PC IP address	4
19	Test device port	2
21	User PC port	2
23	User PC MAC address	6
29	Test device MAC address	6

Lose frame test command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for contitional modifying	6
19	Punctuation mark '/'	1
20	Frame to modify	n

Preamble and start frame delimiter test command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for contitional modi-fying	6
19	Punctuation mark '/'	1
20	Amount of frames to modify	1
21	Amount of preamble bytes	1
22	Change SFD byte to preamble byte	1

Destination address test command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for contitional modifying	6
19	Punctuation mark '/'	1
20	Amount of frames to modify	1
21	Fix checksum	1
22	New frame destination MAC address	6

Source address test command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for contitional modifying	6
19	Punctuation mark '/'	1
20	Amount of frames to modify	1
21	Fix checksum	1
22	New frame source MAC address	6

EtherType and length test command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for contitional modifying	6
19	Punctuation mark '/'	1
20	Amount of frames to modify	1
21	Fix checksum	1
22	New EtherType field	2

Ethernet data payload test command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for contitional modifying	6
19	Punctuation mark '/'	1
20	Amount of frames to modify	1
21	Fix checksum	1
22	Offset from Ethernet payload start	6
28	Size (bytes) of given payload data	2
30	New payload data	6

Frame check sequence test command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for contitional modifying	6
19	Punctuation mark '/'	1
20	Amount of frames to modify	1
21	Fix checksum	1
22	New FCS data	4

InterPacket Gap length test command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for contitional modifying	6
19	Punctuation mark '/'	1
20	New InterPacket Gap length	1

Send custom frame from file / Runt frames test command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for contitional modifying	6
19	Punctuation mark '/'	1
20	Frame, not incuding preamble or SFD	n

Swap order of frames test command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for contitional modifying	6
19	Punctuation mark '/'	1
20	Amount of frames to swap	1

Delay all frames test command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for contitional modifying	6
19	Punctuation mark '/'	1
20	Amount of time to delay (ms)	2

Delay selected number of frames test command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for contitional modifying	6
19	Punctuation mark '/'	1
20	Amount of frames to delay	1
21	Amount of time to delay (ms)	2

DoS with blocking all communication test command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for contitional modifying	6
19	Punctuation mark '/'	1
20	Amount of time to continue DoS	2
21	Frame, not incuding preamble or SFD	n

DoS with allowing all communication test command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for contitional modifying	6
19	Punctuation mark '/'	1
20	Amount of time to continue DoS	2
21	Frame, not incuding preamble or SFD	n

Invert single bit test command

Octet	Description	Bytes
0	Protocol Main Identifier	9
9	Test ID	1
10	Punctuation mark '/'	1
11	Modify direction	1
12	Punctuation mark '/'	1
13	MAC address for contitional modifying	6
19	Punctuation mark '/'	1
20	Amount of frames to modify	2
21	Bit offset to inverted bit from frame start	3