Real Time Packet Processing with FPGAs

A network security toolbox with encryption features designed for FPGA logic-fabrics

Joint Master's Degree in

Nanotechnologies for ICTs

held by

Politecnico di Torino

Institut Polytechnique de Grenoble

École Polytechnique Fédérale de Lausanne

Master's thesis dissertation of

Matteo Collura

Supervisor:
Prof. Luciano Scaltrito

Co-Supervisor:
Prof. Sergio Ferrero

External Advisor:
Mike Stengle

External Co - Supervisors:
Prof. Angelo Geraci
Prof. Yusuf Leblebici

Abstract

The goal of this thesis is to design an application based on a Field Programmable Gate Array (FPGA) architecture to be used in a security environment. Such application consists in processing the network packets exchanged across a wired Ethernet connection in real time. This allows to filter the exchanged data according to a specific set of conditions, programmed by the user. Moreover, in order to guarantee a secure communication between two peers, it also includes an encryption scheme to manage encryption/decryption of data in real time. As a result, this application is referred to as a *toolbox*, due to the existence in hardware of different options of configuration, to be enabled by the user, that consist in a transparent interaction with the data flow, at a cost of a negligible latency (less than a couple of hundreds of nanoseconds). This works even when data are changed on-the-fly, e.g., during the encryption/decryption process.

The device used for this purpose is a hybrid solution, defined as System on (Programmable) Chip (SoC), that embeds a classic CPU architecture into a FPGA logic-fabric. As a result, depending on the implemented resource balance, three different strategies were followed to achieve the goals of this thesis, but only one fully succeeded to achieve all the goals, with a design started from scratch. All the three designs will be explained thoroughly, indicating the reasons of failure or success compared to the achieved results. In the end, an overview on the future upgrades will be provided.

Acknowledgments

First and foremost, my research would have been impossible without the aid and support of Knowledge Resources GmbH. I would like to thank my supervisor and CEO of the company, Mike Stengle, for allowing me to work in his company on such promising research. His suggestions were fundamental to develop new ideas and to find a way out when I was stuck at standstills. Furthermore, I am profoundly grateful to all the other colleagues, especially Luis Gaemperle, for his patience and willingness to help me when I was in trouble with my work: even though the solution was not clear, he always spotted the right path. Many thanks to Marco Fellmann, hardcore worker, for his tolerance to my overtime hours in the office, and to Marie Stengle for her moral support.

I would also like to thank my supervisors Professor Luciano Scaltrito and Professor Sergio Ferrero from Politecnico di Torino, and my external Co-Supervisor Professor Angelo Geraci from Politecnico di Milano. A special acknowledgment to Professor Yusuf Leblebici from École Polytechnique Fédérale de Lausanne, without whom I would not have known Knowledge Resources GmbH.

Heartfelt thanks go to my family and closest friends, because of their constant and fundamental support. In particular, there are a few well deserved explicit mentions.

To Martina, for being such an extraordinary lady and for her infinite patience and support, especially in these last months; as we know, we can be heroes.

To Piero and Giancarlo, whom I consider as brothers, with whom I grew wiser; they have always been present, despite the physical distance, in any circumstance, for better and for worse. Needless to mention all the beautiful days lived together, the endless laughs and the perfect synergy we have always had.

To Matteo, best friend for my whole life and brilliant mind, with whom I share my

favorite hobbies and interests; thanks for all the advice.

To Carola, Alexandro, and Eugenio, for always keeping in touch and caring for each other, even though it is hard to meet; thanks for all the endless night calls.

To Giovanni, Roberta, and Alessandra, for being always willing to find an opportunity to meet, whenever possible, and to spend beautiful moments together.

Finally, my deepest and heartfelt gratitude, as well as the most important acknowledgments of the whole list, go to my parents, whom made me into the person I am today, and without whom I could not have even started my career.

To my dad Ugo, whom always challenged and pushed me to the top of my capabilities, and even beyond. He is a fortress, resilient, and never gave up, even when all around was falling apart. Thanks for making me curious and passionate about science and, in general, this world. Thanks also for building values into me such as ethics and moral. Such precious advice has been fundamental.

To my beloved mum Silvana, whom was the most beautiful person in this world and should deserve way more than those few lines. Thanks for having been such an enthusiast reference to me, for having taught how to live and to behave among people, for having always been interested in my work and supported me throughout any difficulty. I am profoundly grateful for her genuine love, and I will never forget her smile nor her teachings. She will be my role model, forever.

Contents

List of Figures

List of Tables

Alla mia cara Mamma,

per sempre, nel mio cuore.

Chapter 1

Introduction

In today's world, most of the sensitive data are exchanged through wires and digital signals. Although there are still several institutions collecting old fashioned and bulky piles of papers, the central archives are now hosted by cloud services, making the whole paper just a backup resource with a legal value. This is indeed a good achievement concerning smart solutions, considering the evolution of the digital era: the existence of substantial digital archives is now possible and they are much easier to be queried just with one click. On the other hand, there is another perspective to be taken into account: the usually underestimated *security*.

Nowadays the electronics/IT market pushes to provide everyone with high-speed network connections, allowing to exchange as much information as possible in the shortest amount of time: for instance today the most common wired connections run at a rate of 1 Gigabit per second (Gbps). One might be interested in establishing secure and encrypted communication with another peer. There are mainly three questions one can ask when considering the security of digital data:

- What type of data is about to be stored?
- Who does guarantee that data are stored in a safe place?
- How are such data transmitted from one digital entity to another?

The answer to the first question is quite straightforward: every single action producing data is logged somewhere; therefore all kinds of data are stored in a history-like docu-

ment, keeping track of every single operation performed by the device under the scope. This collection is useful to whom is providing a service or selling a device: by applying mathematical models to the recorded data, it is possible to draw accurate statistics and forecasts. The complexity of such models is not a problem anymore, because the limits of computational power are overcome year after year. As a result, data have both a technical and economical value, and thus can be traded or sold. Therefore, they must be protected.

On the other hand, the second question, requires an ethical answer, rather than a mathematical one. Before the digital era came into being, all the data logs were stored in a place under the responsibility of the service provider: banks and other bureaucratic institutions were holding papers inside their buildings guaranteeing privacy for the customers and hiding all the sensitive data in a safe. To mention another example, two companies that wanted to exchange a message through post mail had to pass through the post office. The recipient was supposed to head towards such post office, validate his identity and eventually collect his message. Today, communication between people is mostly mediated by e-mails, and these digital documents are stored somewhere according to the mail service provider policies. As a result, the user delegates possibly another party to manage the communication, and potentially own the data. Here comes the main difference introduced by the digital revolution, i.e., the possibility of keeping a copy of the data exchanged, transparently and silently. Considering the previous example again, if a third party had tried to read the shipped message, he would have opened the envelope, causing irreversible damage. Today, even if the message is encrypted, one can still make a copy and work on it later, forwarding the original message, without any trace. Connected to this topic of data ownership, that will not be further discussed, the last question of the previous list is the starting point of this thesis.

Question: *How is digital data transmitted from one entity to another?*

The proposed question can be approached in two ways: the first one is looking more at the means, or the carriers, to bring the information, whereas the other is mostly concerned with the information itself. Starting from the first approach, two classic setups for

connecting two different entities that build a network consist in choosing a cabled connection (usually Ethernet) or a wireless one. The latter is in general not recommended for several reasons: first of all, the maximum speed for transmitting data is limited by the medium (air), which exposes the signal to interferences and impacts the overall latency, i.e., the amount of time a message needs to travel from the sender to the receiver; secondly, the information is exposed to the public, and thus anyone in range with respect to the wireless source might be able to eavesdrop on the communication.

That being said, Ethernet looks like the most stable and safe way to transmit a piece of information. In the following chapters, it will follow a more detailed analysis of the Ethernet standard and the related protocols, because they play a relevant role in the whole thesis. The goal, however, is to find a way that guarantees a higher level of reliability as regards the transmitted information. This might well be a decent encryption algorithm: if one could even eavesdrop on the communication, he would retrieve a bunch of encrypted data. Eventually, encryption does not guarantee the 100% security of transmission, because eavesdroppers can still spend a significant amount of time attempting to decrypt the message, depending on the strength of such an algorithm.

Nonetheless, the encryption solution seems reasonable, but it assumes that the recipients of the initial message are fully aware of the implemented schema. Luckily, a new architecture for transmitting data is not necessary, because the existing Ethernet standard can be implemented with the addition of a cryptographic layer *on top*, like in a Virtual Private Network (VPN).

A *Virtual Private Network* is a technology that allows a user to connect his device to a private network (as the network of a company, or the private home network) via a safe and encrypted connection, even though he is accessing the internet from a public network. Formally, a VPN can extend a private network across a public one, enabling the user to send and receive data as if he is directly connected to the private network. The data travel along secure tunnels that can be accessed only through authentication methods like a password or other unique identification methods. Figure 1 summarizes this concept. If one is not able to ensure the safety of a transmission line, a reasonable solution consists in

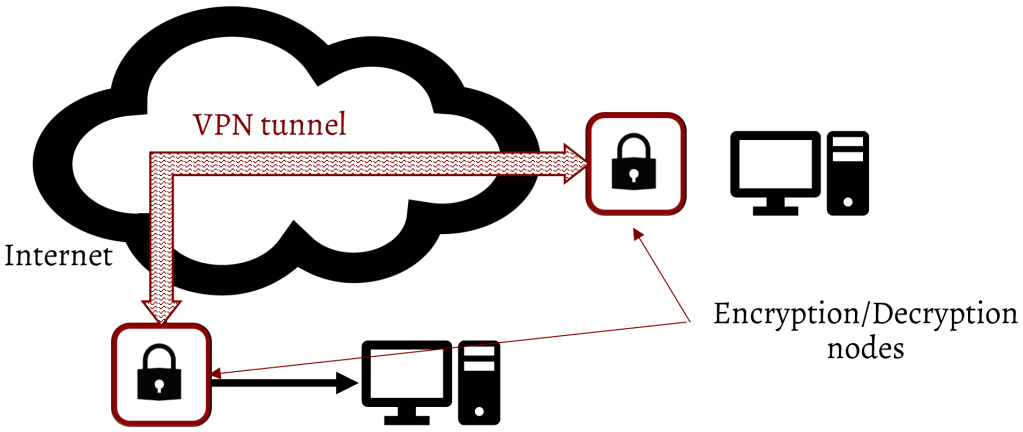

Figure 1: VPN tunneling scheme

creating a custom set of encryption rules to be applied *on top of* the primary architecture.

It is now necessary to understand how to design a useful application of this technology, that is possibly innovative. Since there are already many VPN software services, usually sold and hosted by big providers (e.g., Cisco, or ProtonVPN [1]), a different approach was adopted. First of all, the final user of such a product was meant to be a company or a private, with a local network to be securely accessed. From a security point of view, one should also be interested in inspecting the data exchanged through such a network and set some conditions before actually accepting them. As a result, the goal is to provide a device able to be installed directly in such a network and providing security features, both similar to a VPN or a firewall. A firewall enables the network owner to filter the incoming traffic according to user-defined criteria. Considering a traditional computer architecture, the Central Processing Unit (CPU) needs to interact many times with the memory to evaluate such conditions and eventually provide active modifications to the received data. However, the high traffic rate is comparable to the processor speed, that might struggle with performing these tasks in due time. This shuttling of data between local memory and processor is typically referred to as Von Neumann bottleneck and leads to large expenditures of time and energy. To solve this problem, one can change the architecture and opt for a non-Von Neumann class of devices, e.g., *Field Programmable Gate Arrays* (FPGAs).

Question: *What is an FPGA and why is it a good candidate for such a design?*

FPGAs are electronic devices, whose functionalities are entirely re-programmable via hardware description languages. Among such languages, the most popular are *VHDL* and *Verilog*, and they require the programmer to have a different skillset with respect to traditional software programming, due to the absence of a processing unit executing the instructions. A Hardware Description Language (HDL), as suggested by the name, provides a *description* of logic instructions or expressions to be implemented in hardware. FPGAs show a great potential as concerns *hardware acceleration*, because of the possibility of instantiating several processing units, working in parallel on multiple sets of data. As a result, FPGAs are a family of devices belonging to *Non-Von Neumann Architectures*. These kinds of architectures can be easier to use to monitor a flow of network data because they do not suffer from the previously described bottleneck problem. The set of conditions for monitoring data is translated into a set of logic expressions, which are implemented in the programmable FPGA fabric. The data flow is driven by Finite State Machines, i.e., programmable local processing units, that preserve the original *consecutio temporum* of data, and possibly allow for some on-the-fly modifications. Such modifications are produced by a set of additional logical expressions embedded in the logic-fabric with no extra delay.

The versatility of such devices and their available resources make them very interesting for industry applications. From automotive to robotics to avionics, FPGAs are being implemented more and more because they drain low amount of power and provide help in hardware acceleration. The average power consumption of such devices is usually more economical with respect to classic architectures even though it is not possible to know it precisely a priori because it strongly depends on the number of resources used and the maximum clock speed. One of the most active features of FPGAs, also making them the best candidate for a network application, is that they can process data at a shallow level and efficiently do *pipelining* [2]. This last term belongs to the computing jargon: it indicates a form of organization in which consecutive steps of an instruction sequence are executed in turn by a series of entities able to operate concurrently so that another in-

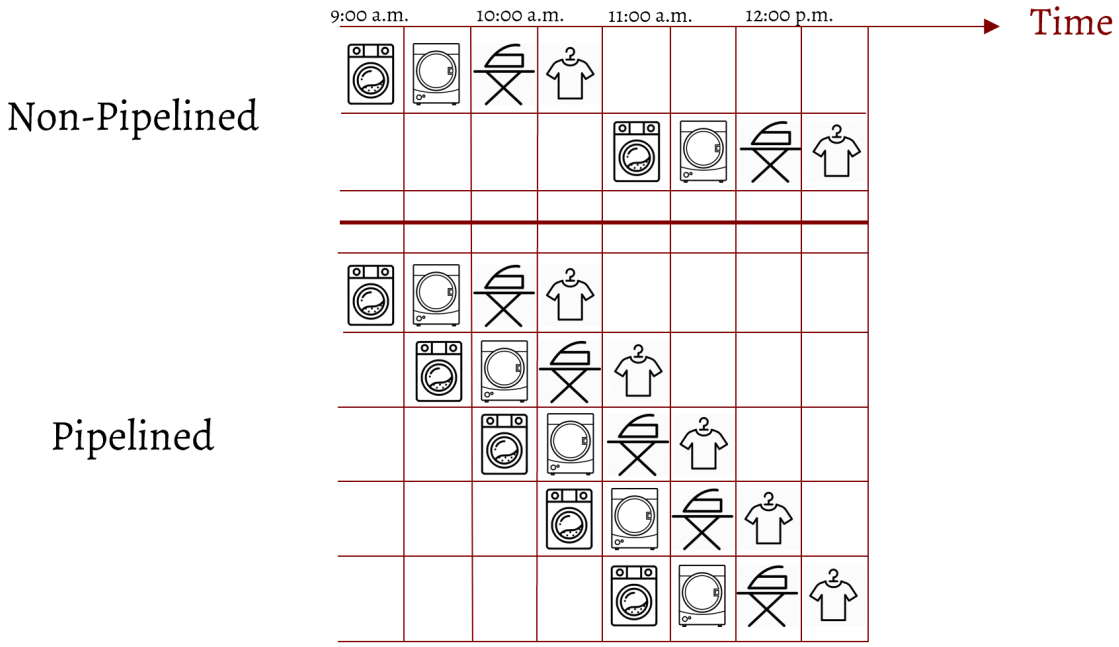

Figure 2: Pipelining applied to the laundry model: if the process is pipelined, there will be possibly more than one operation per time. In this case, in the same time window, five operations are completed against two in the non-pipelined model

struction can be launched before the previous one is finished. This concept is usually explained by the laundry example (see figure 2): suppose that a laundry has several clothes to be washed and ironed. One can distinguish four separate tasks to be done: washing, drying, ironing, storing. If the laundry operates them sequentially, it will take much time, being the size of the washing machine limited. However, if the schedule is planned accordingly, the laundry machines might be operating almost non-stop: as soon as one load is washed/dried/ironed, another one will follow, and all the tools will be running at the same time.

That being said, an FPGA device can handle the network traffic *on the fly*, taking decisions and eventually providing modifications while data is traveling inside the board. Such capability makes the whole system work transparently from the network point of view, as the data is possibly modified without being delayed much. Said better, employing a proper pipelining, there will be only a global tiny offset concerning delay. For this specific application, it was estimated to be around 0.2 µs. Considering that the minimum average latency for an average Gigabit Ethernet line is around 50 µs [3], the offset is indeed

negligible. In the following chapters, the reader will understand:

- What is the development device used and the state of the art (chapter 2)
- What is the state of the art of network protocols (chapter 3)
- Which are the three different followed approaches to build this application and achieve all the goals (chapters 4-5-6)
- What is the outcome and how to test it (chapter 7)
- What are the improvements for the future development of such device (chapter 8)

The whole activity was performed at Knowledge Resources GmbH, a leading company in the embedded systems market based in Basel, Switzerland. Such company specializes in high-end FPGA-centric solutions in image processing, high-performance computing and robotics and provides small engineering services.

Chapter 2

State of the Art

In this chapter the reader will get a broad overview of FPGA technology and its evolution across time: slightly more than thirty years passed since the first device was sold on the market. Then, it will follow a detailed description of the development board adopted for this thesis. Eventually, a description of the software necessary to interact and effectively program the FPGA will conclude the chapter.

2.1 Hardware

FPGAs are integrated circuits whose functionalities are programmable through hardware description programming languages. Such languages are also similar to that used for designing an Application Specific Integrated Circuit (ASIC). FPGAs contain arrays of Configurable Logic Blocks (CLB), which are considered to be their fundamental building block [4]. Each of those primary elements is placed in an ordered structure, all over the chip size, and is connected to the neighbors through the so-called Switch Boxes. Such interconnections are re-configurable and allow the different CLBs to be wired together. The Switch Box is a straightforward component to be explained since it can be seen as a multiplexer enabling all the possible paths between two neighboring CLBs, as shown in figure 3. Such CLBs, on the other hand, are showing different primary structures, according to the FPGA family to which they belong.

Along with time, there has been a significant evolution regarding scalability. In 1984, the company Altera brought to the market the first re-programmable logic device [6]. In

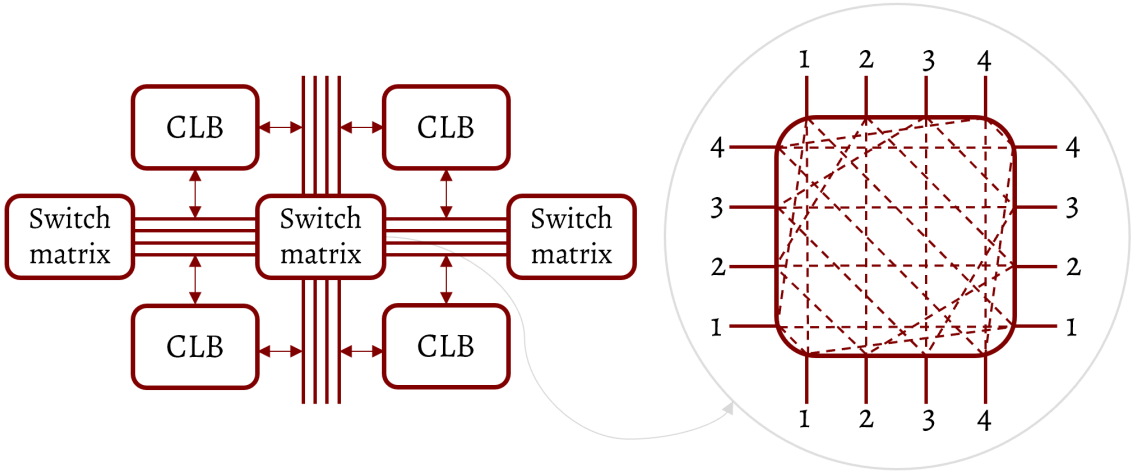

Figure 3: Interconnections between CLBs: interaction with the Switch Boxes, as explained in [5]

1985, its competitor Xilinx invented the first commercially viable FPGA [7], that was containing 64 CLBs, roughly 8000 logic gates. It took a couple of years before FPGAs started improving the number of components inside the hardware, together with the volume of production. In the end, around year 2000 they started being used in the automotive world and for industrial applications [8], having them become appealing, and also improved regarding components, roughly one million logic gates. In the recent years, together with the improvements regarding the electronic components, a different approach was adopted, especially by Xilinx: this resulted in a hybrid solution, combining the flexibility of a re-programmable logic fabric with the power of an embedded microprocessor. Such a new creation form a complete System on (Programmable) Chip (SoC). An example of a family of devices with these characteristics is the Xilinx Zynq®-7000 All Programmable SoC, that embeds a dual-core 1 GHz processor, ARM®Cortex®-A9, in the FPGA fabric, designed with 28 nm Programmable Logic.

The state of the art of the hardware logic-fabrics provided by Xilinx is the so-called *7 Series*. Such class of devices is made of four different categories, differentiating in terms of cost and performances: from the lowest to highest quality, they are *Spartan-7*, *Artix-7*, *Kintex-7* and eventually *Virtex-7*. All these families are built on high performance, low power, high-k metal gate, 28 nm process technology and contain roughly 2 million logic cells [9].

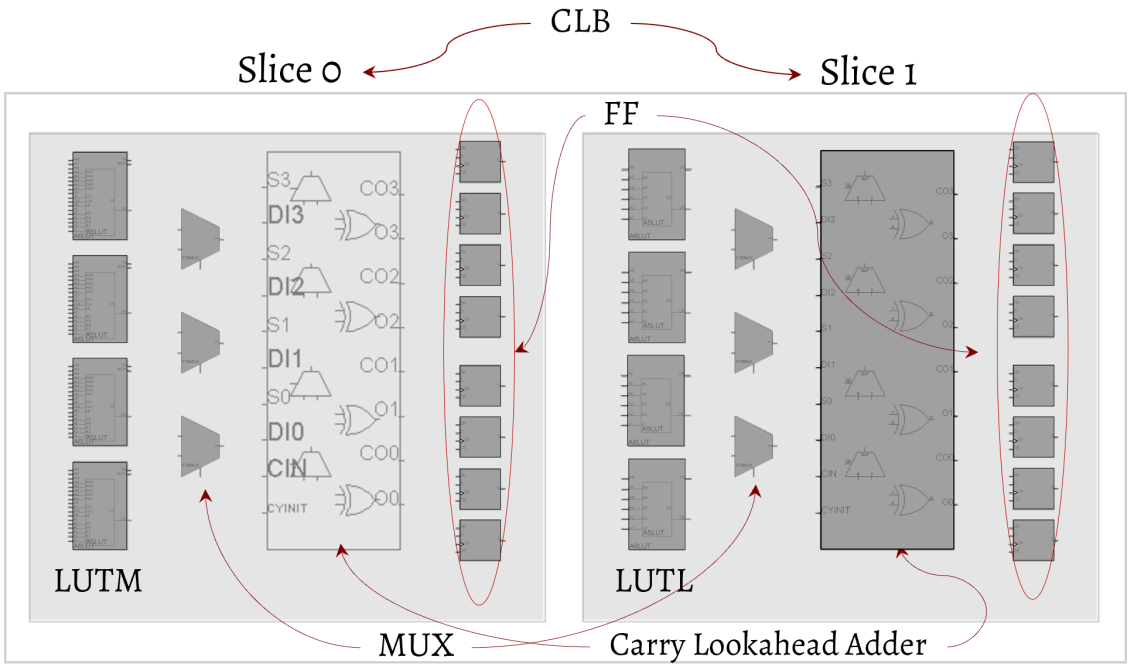

Figure 4: CLB internal components: from slice partition to its elementary components

Among the main building blocks of which the fabric is made, the most important ones are the previously mentioned CLBs, Block Random Access Memory (Block RAM or BRAM) cells, and DSP slices.

2.1.1 Configurable Logic Blocks

The Configurable Logic Blocks, as regards the 7 Series devices, provide high-performance FPGA logic. They can be imagined as a box with a bunch of partially wired components, that can be programmed to perform different functions. Among the elements inside a CLB, explained in the following, there are eight Look-Up Tables (LUTs) for random logic (LUTL) or distributed memory (LUTM), sixteen Flip-Flops, and two 4-bit Adders. A CLB is made of two slices. Each slice is made of four LUTs, eight Flip-Flops, and one 4-bit Adder [9]. Figure 4 shows the CLB inside a 7 Series fabric, with all its internal components.

♔ 6-input LUT technology that can also be configured as a dual 5-input LUT [10]

A Look-Up Table is an array of data able to replace calculus operations with a simple look-up. It is the same principle used by the old trigonometric tables: the results of such op-

erations are previously computed and stored in a dedicated memory. Then, instead of performing the calculus, the inputs will be fed to memory, to fetch the result that was stored in the corresponding position. In this case, such component can implement any arbitrarily defined six-input Boolean function [10].

- Shift Register Logic capability and Distributed memory

Each slice has eight storage elements: four of them can be configured as edge-triggered D-Flip-Flops, whereas the remaining four can also be configured as level-sensitive latches. Apart from that, the LUTs present in each CLB can be implemented as synchronous RAM resources, called Distributed RAM elements[10]. Another interesting feature is the Shift Register (32-bit) that is not using the flip-flops available in a CLB but rather the LUT elements. This way, each LUT can delay serial data from one to 32 cycles.

- Multiplexers

A single LUT can implement four-to-one multiplexers. Two LUTs can implement an eight-to-one multiplexer, and so on. Moreover, there are three multiplexing elements per slice.

- High-speed carry logic for arithmetic functions

Apart from LUTs, defined also as "function generators" due to their potential, there are also fast 4-bit look-ahead carry logic to provide quick arithmetic operations like addition and subtractions. They can be cascaded to allow a bigger input size.

2.1.2 Block RAM

The block RAM is a Random Access Memory block, able to store up to 36 Kbits of information. Such block can be configured either as two separate 18 Kb RAMs, or just a single 36 Kb[11]. According to the word or depth size, as shown in figure 5, this memory block can be programmed with many primitive *aspect ratios*. Such ratio indicates the number of bits composing each word and the number of bits required for addressing each word (and thus determining the depth, since the overall size is fixed), for example, 10x36. Sometimes, however, instead of indicating the number of the address bits, one can use the maximum

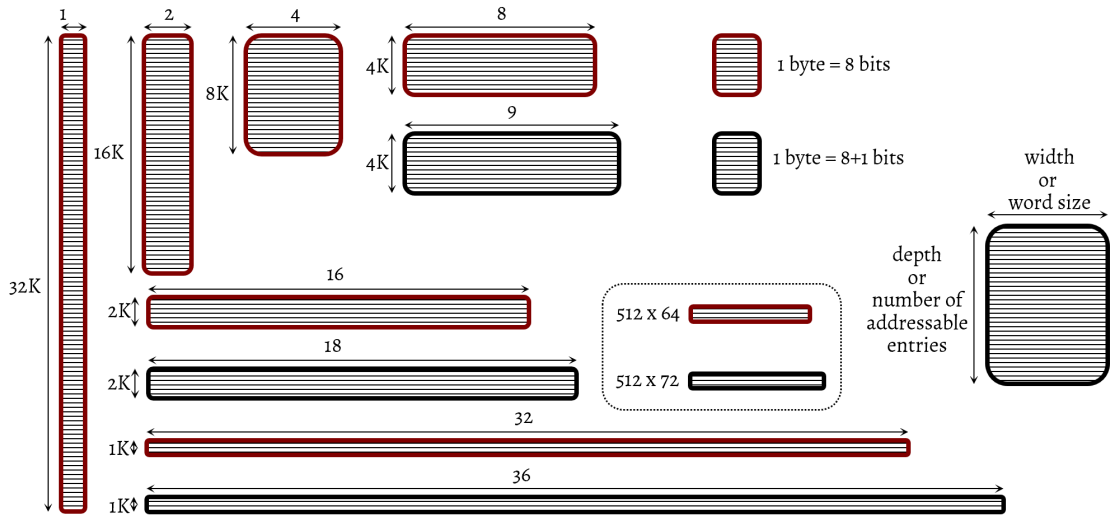

Figure 5: Configuration of a BRAM cell: the word size and the maximum number of addressable entries are defining the aspect ratio

addressable number, and so 1Kx36 (as 2^{10}=1024). Moreover, each Block RAM can be configured to be accessed through a single port or a dual port. This will be examined later, together with the design description.

Read and Write operations on a BRAM cell are issued through a specific interface, made of a data bus, a clock signal and other control signals, enabling or disabling the corresponding operation. The reset pin is optional and is the only signal belonging to this interface which is not synchronous to the clock. Also, these operations will be explained later, applied directly to the case study.

To estimate the amount of BRAM resources in a hardware design, usually, the number of cells is counted, rather than their overall size. From the physical point of view, the 7-Series logic-fabrics have all the available cells arranged in vertical rows across the floor plan, as shown in figure 6. As a result, the designer should try not to abuse such memory resource and minimize the number of required primitives, by instantiating only a few cells per specific application and choosing wisely the aspect ratios. The reasons for this advice have their roots in the physical organization across the fabric. If the design requires more cells than the ones available on one of the vertical lines to build a memory space, the extra cells will be placed on a separate line. That makes it harder for the clock

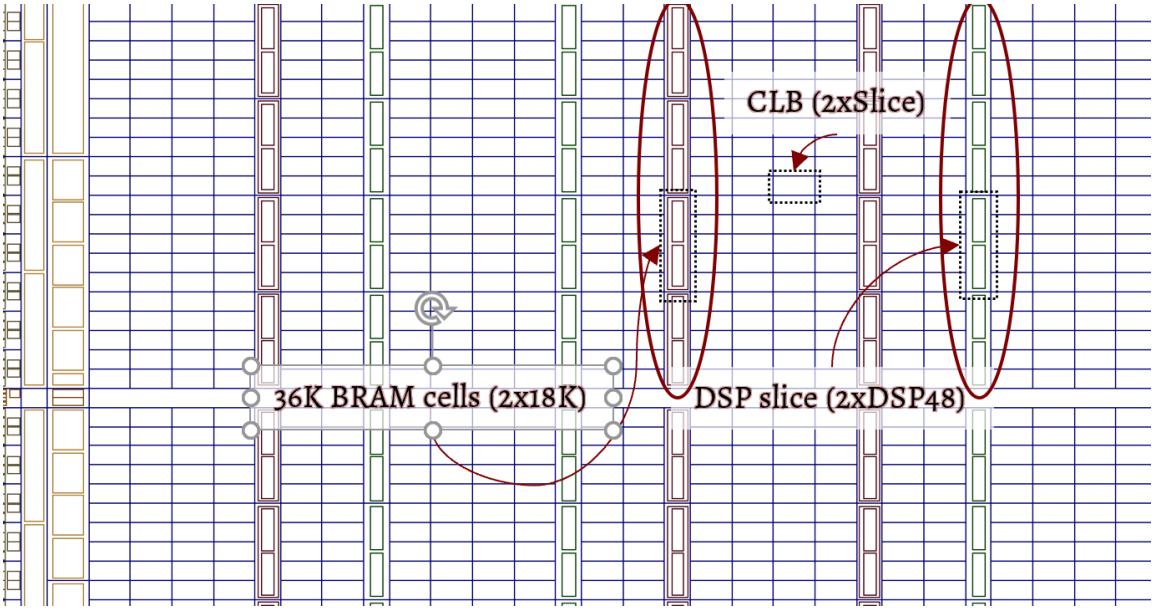

Figure 6: Cropped floorplan of a 7-Series fabric, showing the arrangement of CLBs, BRAM cells and DSP slices

and data bus to reach at the same time all the BRAM cells generating the same memory space, leading possibly to timing violations.

2.1.3 DSP Slices

Digital Signal Processing (DSP) is one of the most practical tasks that FPGAs are capable of performing, due to the high level of flexibility and inclination in running parallel algorithms. A DSP slice implements many binary multipliers and accumulators, grouped in full-custom, low-power logic blocks [12]. As regards the fabric floor plan organization, the DSP blocks are placed similarly to BRAM cells, since they are also aligned over vertical lines. As regards the functionalities, a DSP slice can be configured to perform several arithmetic operations, among which multiplications and comparisons. This comes along with a fast Arithmetic Logic Unit (ALU), that can be programmed to perform additions or subtractions between up to 48-bit inputs. Regarding its implementation and configuration, the topic will be covered in the following chapters, based on the case study.

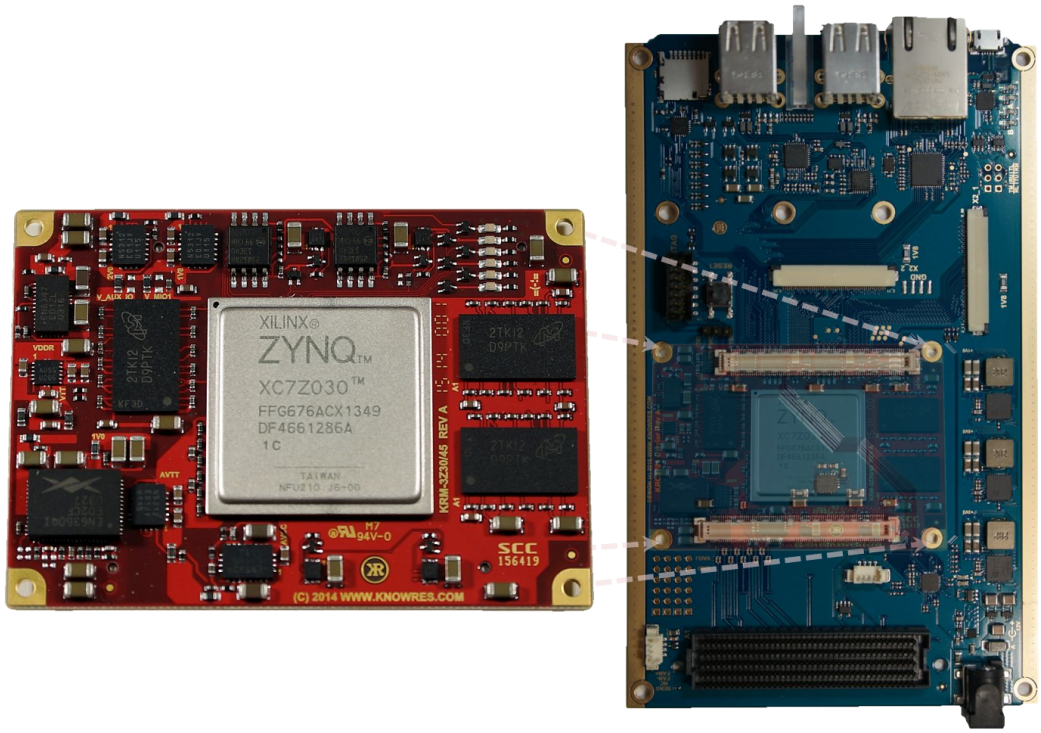

Figure 7: KRM-3Z7030 50mm x 70mm Module (left) mounted on KRC3701 evaluation carrier (right). Pictures taken from the corresponding product pages [13], [14]

2.2 Hardware Resource Under Development: KRM-3Z7030

The device used for developing this Thesis is manufactured by Knowledge Resources GmbH and is referred to as KRM-3Z7030 module[13]. Such module, shown in figure 7, contains a System on a (Programmable) Chip (SoC) belonging to the Zynq-7000 family of devices, by Xilinx, namely the Zynq-7030. The Zynq-7000 family provides an interesting hybrid device, which embeds a dual-core ARM Cortex A9 processor into an FPGA logic-fabric, in this case, a Kintex-7 type. The two joined architectures are still well differentiated inside the device: that is why the region around the processor is called Processing System (PS), and the remaining logic-fabric is called Programmable Logic (PL). The specific part number of this FPGA SoC is XC7Z030-1FFG676C, and its speed grade is equal to 1. According to the datasheet [15], this device contains already 125 thousand logic cells, 400 DSP Slices, 265 36Kb Block RAM cells and 256Kb On-Chip Memory (OCM). Then, thanks to Knowledge Resources customization, the device has four additional Gb of Low Power Double Data Rate D-RAM (LPDDR3) on the PS side, whereas only two ad-

ditional Gb of LPDDR3 Dynamic RAM (DRAM) on the PL side. The speed grade indicates the quality of both fabric interconnections and single components, and usually, it ranges from 1 to 3. A higher speed grade suggests a higher maximum guaranteed operating frequency of the components inside the logic-fabric. The KRM-3Z7030 module comes along with the KRC3701 evaluation carrier kit (see figure 7), still developed by Knowledge Resources GmbH [14], which allows attaching several peripherals to the main module, both on PS and on PL sides. To mention some of them, there are USB ports, an Ethernet port, a micro-SD card reader (that might store the module Firmware) routed towards the PS, and a pair of 50-pin Zero Insertion Force (ZIF) connectors, allowing to connect a wide variety of external modules, routed inside the PL. The beauty of this hybrid approach consists in combining the computational power of the dual-core processor with the versatility of the logic-fabric so that the first might offload many of its tasks to the programmable logic: as a result, one usually refers to this capability as *hardware acceleration*[16].

From here on, the device under development will always be considered as the KRM-3Z7030 module mounted on the KRC3701 evaluation carrier kit. To start the device correctly, both PS and PL must receive a preliminary configuration. However, in the following sections, a general overview will be provided on both the two environments.

2.2.1 PS Configuration

The Processing System of a Zynq device must be properly configured before starting with the hardware design, especially as regards the interfaces with the PL and the external world, through the connectors contained in the carrier kit. First of all, the Dynamic 2Gb memory may be enabled and accessed through the corresponding controller, via a 32-bit bus. Then, as regards the Input/Output (I/O) peripherals, a few more words are required. The PS has 54 programmable pins available for connecting to a specific peripheral. Each peripheral might be assigned one of the different pre-defined sets of pins, allowing a flexible assignment of multiple devices simultaneously[9]. This set of 54 configurable pins is referred to as Multiplexed Input/Output (MIO), and the name already tells what the nature of such an interface is. The job made by the MIO is to multiplex access from

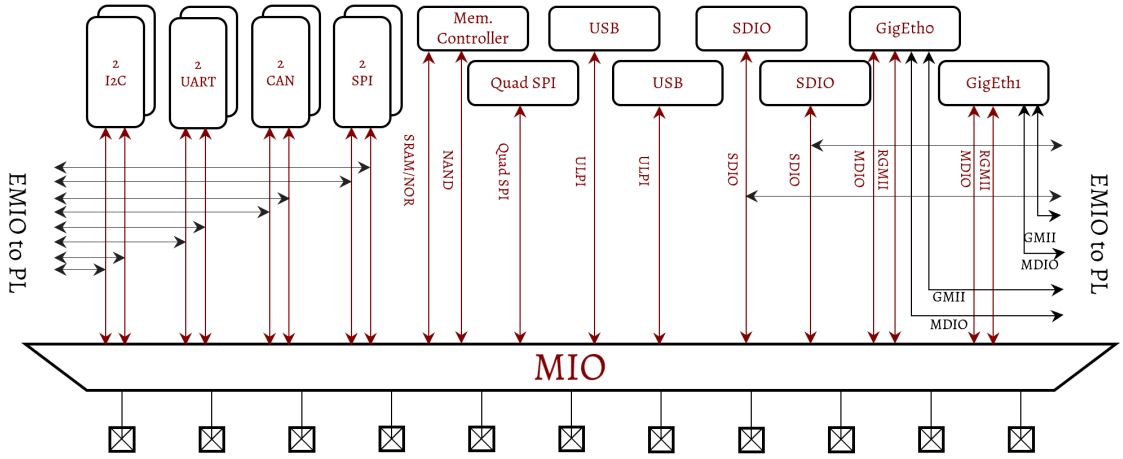

Figure 8: MIO/EMIO routing to PS: Block Design, according to [9]

the PS peripheral to the PS pins as written in the configuration registers. Such configuration registers can be easily set up by the user, through the software interface described in the following section. Figure 8, shows a block diagram of the MIO and all the peripherals that can be mapped. Should extra I/O pins be required, beyond 54, they might be routed through the PL, to the I/O associated with the PL. Such a feature is referred to as Extendable Multiplexed I/O (EMIO) [9].

Among the different options available through the MIO, one will be examined thoroughly in the following chapters: it is the *tri-mode Ethernet MAC*, also referred to as *Gigabit Ethernet MAC Controller* (GigE, or GEM). Such peripheral supports the IEEE 802.3 Standard [17], that is regulating data exchange through a wired Ethernet connection. The name "tri-mode" comes from the three possible configurations, according to the speed of the Ethernet link, namely 10 Mbps, 100 Mbps or 1 Gbps. The corresponding MIO pins are configured to be routed towards the Ethernet RJ45 connector, available on the carrier kit. The second GigE interface, however, is not geared to be routed across the carrier kit via MIO, as there are no additional RJ45 connectors. However, an external Ethernet module can be provided and plugged in the carrier kit. This forces the developer to route this interface via EMIO, as the pins of the external module will be inside the PL.

2.3 Interaction between PS and PL

In the previous section, the reader has already figured out an interface between PL and PS, that is the EMIO, able to route external peripherals to the central core, through the logic fabric. Another possibility is mediated by a set of interfaces that are working with a popular bus type in the embedded system world, the Advanced eXtensible Interface (AXI)[18]. The communication across such interface follows the specifications given by the Advanced Microcontroller Bus Architecture (AMBA®) ARM®AXI Protocol [19]. According to this protocol, there are three types of AXI4 interfaces: AXI4, AXI4-Lite, and AXI4-Stream. The first one is mainly used as an interface for high-performance memory-mapped peripherals, and it is exploited by the PS allow interconnections with the PL. Before explaining the set of available interface, one additional remark must be clarified, even though it may look obvious. An interconnection between two entities, one of which is a processing unit, is usually defined according to the hierarchical level of the two parties involved, namely, *Master* and *Slave*. Therefore, a Master interface allows an entity to send commands, or in general to drive the connection with the other entity, whereas a Slave interface allows the unit to be controlled by an external peripheral. The PS of a Zynq-7000 SoC has two 32-bit AXI master interfaces, and two 32-bit AXI slave often referred to as General Purpose AXI interfaces. Then, it has four 64-bit/32-bit configurable, buffered AXI slave interfaces, which allow direct access to Double Data Rate (DDR) memory and On-Chip Memory (OCM). This is usually referred to as high-performance AXI port. Figure 9 shows the architectural overview of a Zynq-7000 PS. The reader might notice that the previously mentioned AXI interfaces are routed in this architecture through an Interconnect block. Such component supports multiple Master-Slave transactions at the same time and therefore is non-blocking.

Last but not least, the PS can configure and enable up to 4 separate clock output signals, facing the PL side, together with four different reset output signals.

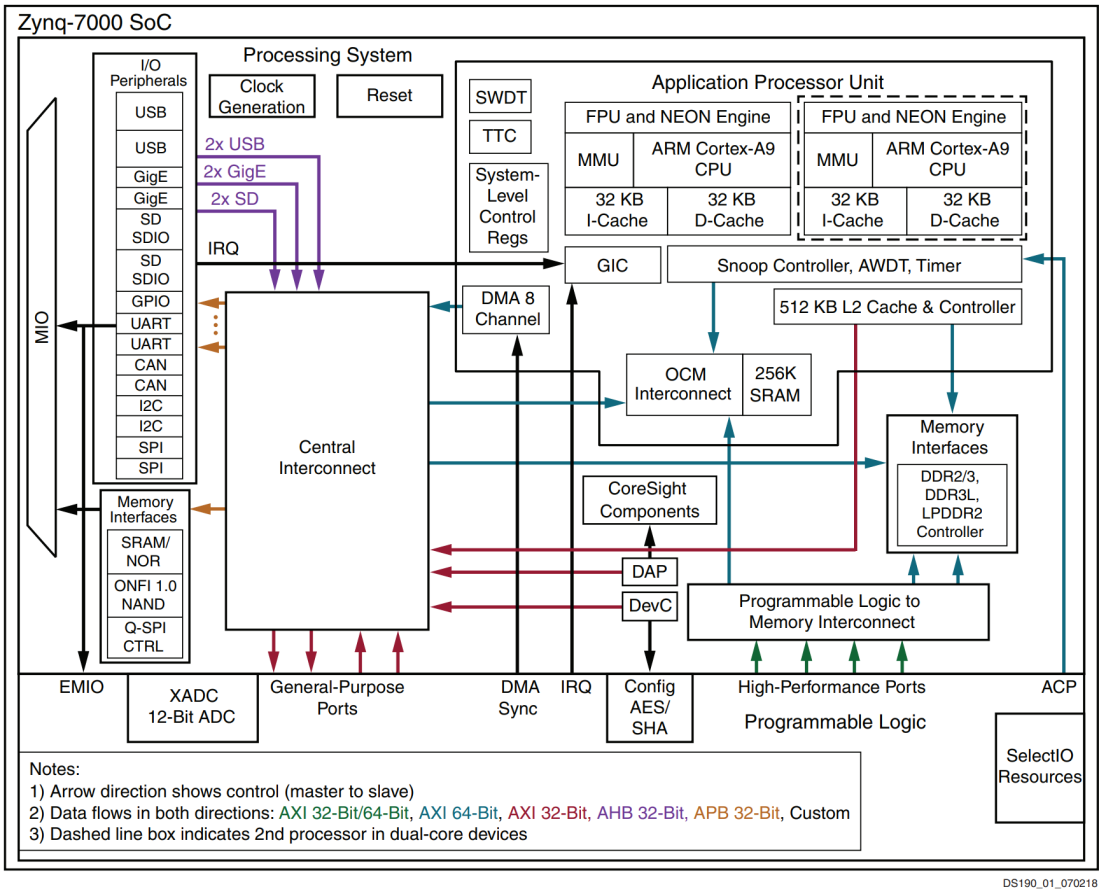

Figure 9: Architecural overview of the Zynq-7000 SoC Processing System. The picture is taken from the official datasheet [9]

2.4 Software

After this short overview of the hardware specifications, it is necessary to understand how to interact and effectively program the device under development. For this purpose, there is a powerful and complete suite, provided by Xilinx, called VIVADO®[1] [20]. This collection of design tools, allows the user to program every Xilinx FPGA. The language required for programming such platforms must be a Hardware Description Language (HDL) like VHDL or Verilog. The developer might choose to write all the lines of codes from scratches, or be helped by drawing and connecting some pre-packaged block-shaped Intellectual Properties (IPs) in a Block Design environment. Such IPs are largely provided by Xilinx and can ease a lot the development for what concerns configurating or inter-

[1]Version 2017.3 as regards this thesis

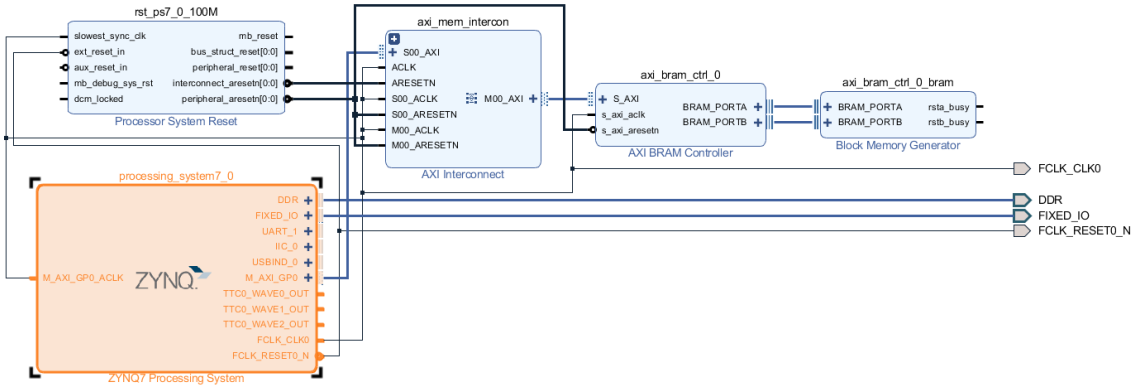

Figure 10: Example of a Block Design inside Vivado®interface

facing parts. For example, all the Zynq®-7000 SoC PS options explained in the previous section are grouped in a configurable IP (Called *ZYNQ7 Processing System* [21]), that can be instantiated and customized in the Block Design. The interface looks like figure 10. Then, as soon as the Block Design is ready, everything can be wrapped in an HDL file. This allows both the developer and the compiler, to connect, possibly, other components to the block design, described by typewritten lines of code.

2.4.1 Programming the PL

In order to make an FPGA execute a piece of code with a specific hardware configuration, it is necessary to program its "firmware". This word must be quoted since in the FPGA jargon one refers to it as *bitstream*. The bitstream file generation is the last step occurring in an hardware design, before testing the device physically under development. Starting from the beginning, the hardware part number must be indicated in the setup. Then, the HDL files together with, possibly, a block design undergo the Register-Transfer-Level elaboration, that converts the written lines of code into a block netlist. Such netlist contains only the FPGA Primitives that are necessary to perform the written operations. A primitive is a component that is native to the target architecture, like LUT, DSP block, Flip-Flop, etc. [22]. At this point, the designer might be interested in simulating the written lines of code. He will be supposed to write a *testbench*, that is an additional HDL file, describing the simulated inputs and boundary conditions to apply, and then to run the *Simulation*. Of course, this is not a mandatory step, but for complicated designs, it is

a must. If the model works as expected, then the *Synthesis* operation can be run. Such process consists in transforming a Register-Transfer-Level(RTL)-specified design into a gate-level representation [23].

The following step is called *implementation* and includes different design transformations, both logical and physical: first, the logical design is optimized to better fit onto the target hardware part (*Opt Design*), then, the single elements of the design are optimized to reduce the amount of power demanded by the target hardware device (*Power Opt Design*). Note that this last step is not mandatory. After that, the design must be placed onto the target hardware floor plan (*Place Design*) and then another non-mandatory step follows, that consists in optimizing the design timing (*Phys Opt Design*). This allows replicating the drivers of high-fanout nets, to distribute the loads across the design. Eventually, the last step consists in routing the design (*Route Design*), that is, drawing the connections between the previously placed components, and ensure, possibly, not to violate any timing.

Especially the last three steps can be iterated many times: moreover, being them iterative processes themselves, their tolerance can be tuned, to spend more or less time to make them converge [24]. As soon as the implementation is done, one can generate the *bitstream* and then load it on board.

2.4.2 Programming the PS

The *bitstream* file previously generated has to be now exported to the Software Development Kit (SDK) folder, together with another file, containing information about the hardware platform specifications. Here, another tool is launched, that is XILINX®SDK, still belonging to VIVADO®suite. It is a C programming environment based on the well-known ECLIPSE®Integrated Development Environment (IDE). This allows the developer to program also the dual-core Processor inside the PS, through a dedicated set of libraries, including even the standard C libraries. As soon as the code is ready, the tool compiles it and the developer can both run or debug the program on the board. The FPGA is first loaded the previously computed *bitstream*, and then the software code is executed. The interface port for programming the FPGA is the so-called Joint Test Action Group (JTAG).

2.5 Anteriority Research

It is always difficult to adequately address research in the hardware design field. Said in different words, there are very few people developing for such platforms as FPGAs, and even fewer that are really aware of what is going on inside the devices, once they are programmed. Probably, as the time goes by, there will be more people and more resources to be consulted. As a result, it is quite challenging to find useful information on the topic, available to everyone. It often happens that the "gurus" of hardware design are working for big and well-known companies, like Xilinx or Intel, where they develop applications under secrecy or contribute to the development of Intellectual Properties. As regards Xilinx, the company had the brilliant idea of creating an online forum where the users can interact, sharing doubts and issues faced during the design of their application. The unusual feature of such a forum is that the employees of the company are replying to the messages from time to time. However, being them so proud of their lines of code, it is not rare to find short and bothered answers when a user demands more clarity. That being said, it was challenging to fetch information regarding applications similar to the goal of this thesis: first, the authors lacked details when describing their work, and then, most of them eventually produced a product to be sold on the market, leaving indeed no trace of the approach they followed.

Nonetheless, some interesting conclusions can be drawn out of the few works found on the web. First of all, the already existing FPGA-based solutions concerning Virtual Private Networks are mainly dealing with the topic of encryption, rather than embedding this function on a broader tool, managing also the link to the network. Considering the works by Cheung and Leong [25], [26], the FPGAs were used to support a central workstation, by performing the encryption steps a VPN is in charge of, to encrypt and transmit the network data. That information is not helping the development of this thesis, at least not at the beginning, as the encryption schemes will be checked out as soon as the FPGA can handle network traffic. Both Mingarelli [27] and Gallego [28] published interesting works on the usage of lwip library and XEmacPs driver. These topics are dealing with

network data management and will be explained in the following chapters. The second was carefully read to grasp more insights about the usage of the DMA engine. The main concern with such research activities, as well as the one by An. et al. [29], is that their final purpose is to build a design that is suitable to be integrated into an operating system. Due to the presence of a classical Von-Neumann architecture, the Zynq-7000 SoC can load an operating system on the PS, and the most common choice is the so-called PETALINUX [30]. Even though this peculiarity is not shown clearly in the description of the goals of this thesis, it was given for granted that the reader would have assumed a /textitstandalone usage of the FPGA, namely, no operating systems are loaded, and the CPU is merely an additional resource.

A remarkable approach to network packet filtering with FPGAs was brought by Whelan [31]. His work shows proper methods to process data, discussing the advantages and drawbacks of different search algorithms. Nevertheless, his application is now obsolete, not to mention its scarcity regarding real-time features. On the other hand, the two most useful papers for this research were from Födisch et al. [32], and del Pino [33]. The first one is more technical and helped in understanding the structure of a network packet, together with the operations performed by the electronics to correctly receive and transmit data through the wires. The second, on the other hand, is focused mainly on the design of a packet processing unit operating at a high level of abstraction, that is different from the outcome of this thesis, operating at a shallow level. In the end, among the browsed papers, there is a lack of investigation on both the cryptographic potential of FPGAs and their high flexibility in processing data in real time. As a result, the motivation to face this activity increased a lot, leading in the end to a successful result.

Chapter 3

Network Protocols and Standards

The purpose of this chapter is to introduce the reader to the main development framework: it is not mandatory to have a vast knowledge on the topic; therefore only a brief of the main protocols and standards regulating network communications will be provided.

3.1 Ethernet Technologies

Using the word *Ethernet*, one refers to a large family of products, used in the Local Area Network (LAN) environment. Such products are covered by the IEEE 802.3 standard [17]. The problem consists in connecting two devices in a Point-to-Point fashion, through a network cable, and letting them exchange data according to such standard. Such wire is a twisted-pair cable, made of four couples of twisted-cables. The endpoint connector is referred to as 8P8C (8 Positions 8 Conductors), and the interface where it is plugged is called Register Jack 45 (RJ-45). The operation over this kind of cables has three data rates definitions: 10 Mbps, i.e., 10Base-T Ethernet, 100 Mbps, i.e., Fast Ethernet and 1000 Mbps, i.e., Gigabit Ethernet.

3.2 Open Systems Interconnection (OSI) Model

The OSI model is a conceptual module, characterizing and standardizing the communication functions inside a computing or telecommunication system without taking into account the internal structure nor the technology [34]. Such model divides a communication system into seven different abstraction layers, often referred to as a stack, which

together carry out all the network functionalities, following a logical-hierarchical model. As far as concerns the design discussed in the following chapters, only layers from 1 (bottom layer) to 3 will be mentioned here.

Each layer defines a communication protocol valid only over that specific layer. Then, to communicate data across the different levels in the stack, say from top to bottom, the whole structure of the message is encapsulated in the *payload* of the following layer message. Such payload is defined as the object of communication, what contains the very true information to be exchanged. This concept can be grasped better by making a comparison with a message in an envelope. The words written on paper are representing a conversation over a specific layer. In order to be shipped, the message is wrapped inside an envelope, that is now serving the communication over the following layer. The letter that is encapsulated in the envelope belongs to the payload of this last layer.

This structure helps with the implementation of different algorithms for routing communications inside the network. As a result, such an arrangement makes the system modular. So, starting from the bottom layer, there are:

1. *Physical Layer*

The goal is to transmit a flux of non-structured data, by setting up correctly the signal voltages and waveforms. It is dealing with mechanical and electronic procedures for establishing, keeping and dismissing a physical link. Here the system understands whether it is allowed to transmit and receive data simultaneously (Full-Duplex mode).

2. *Datalink Layer*

This layer allows reliable data transmission through the physical layer. It sends data frames with proper synchronization and performs an error check to detect transmission errors like signal losses. The *Datalink layer* is the last level of the stack (looking from top to bottom) able to encapsulate the message from the previous layer into a packet containing a header and a tail, used for delivering it correctly through the physical link. The data exchanged through this protocol are referred to as *frames*.

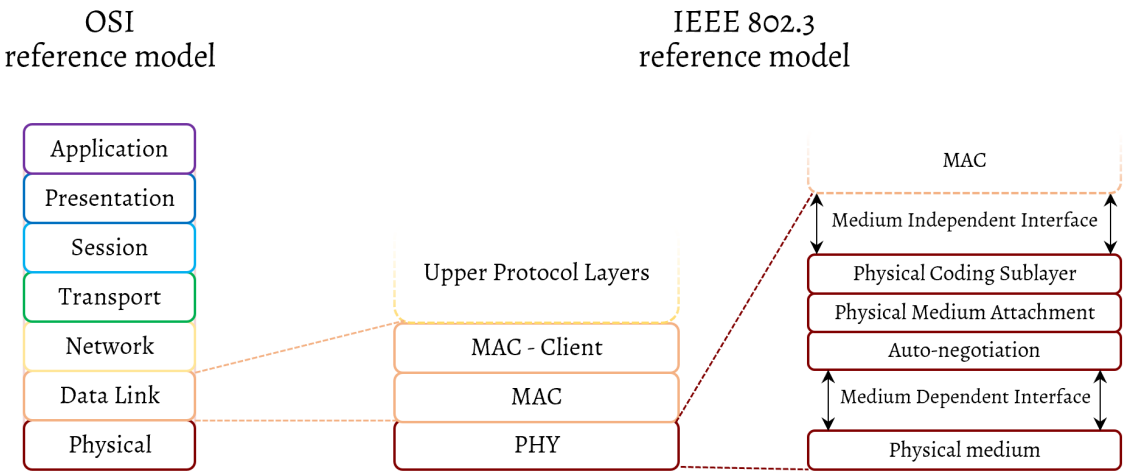

Figure 11: Ethernet Physical Layer Reference Model, according to [35]

3. *Network Layer*

This layer is in charge of routing the message to the correct recipient through the network, across the optimal path. The data exchanged through this protocol are referred to as *packets*.

3.3 Ethernet Physical Layer Reference Model

According to the reference model proposed by the IEEE 802.3 [17], the information exchanged according to the Ethernet standard travels across the OSI stack through a set of different sublayers, as shown in figure 11. Starting from the bottom of the equivalent Physical Layer, the first Sublayer found is the Medium, that, of course, represents the twisted pair cable. Then, the cable connector (here defined as Media Dependent Interface, MDI) allows the device on the other end of the link to exchange information about its capabilities, through the Auto-Negotiation sublayer [35]. Such a set of information includes the maximum link speed supported and the operational mode (Full-Duplex or Half-Duplex, if the transmitter is supposed to wait until the message is received). Following the Auto-Negotiation sublayer, there are the Physical Medium Attachment (PMA) and the Physical Coding Sublayer (PCS). The first contains signal transmitters and receivers together with a dedicated clock recovery logic for the incoming data streams [35]. The second instead

contains the logic necessary to encode, multiplex and synchronize the outgoing data as well as decode and demultiplex the incoming data. All those sublayers belong to the same equivalent OSI Physical Layer, referred to as *PHY*, if considering the Ethernet reference model. The name PHY comes from the electronic component that integrates all the previously described sublayers. The interface allowing the PHY to exchange information with the next Data Link Layer is referred to as Media-Independent Interface. If the link speed is 1 Gbps, such interface becomes a Gigabit Media-Independent Interface (GMII), 8 bits wide. Depending on the PHY capabilities, however, this interface might be only 4 bits wide and thus referred to as Reduced GMII (RGMII).

The following layer in the stack, as mentioned before is the Data Link Layer, however, as regards Ethernet, it is divided into two sublayers, namely, Media Access Control (MAC) and MAC-Client. The first one is responsible for encapsulating data before the transmission occurs, and detecting errors during, or possibly after, the reception phase. As its name suggests, it is also responsible for controlling the access to the telecommunication medium. The MAC-Client instead is usually referred to as Logical Link Control (LLC) and provides an interface-like connection between the MAC and the Network Layer (Number 3, according to OSI model). Eventually, above the MAC layer, the Ethernet equivalent OSI model, is the same as the original OSI model.

3.4 Ethernet Frames

According to the IEEE 802.3 standard, whenever the information travels from one Layer to the following in the stack, from top to bottom, it is encapsulated in the payload of the following data structure. Such structure is often given a name according to the Layer. For what concerns the MAC layer, it is referred to as *Ethernet frame*. As soon as the frame is about to be transmitted, it is encapsulated in the following structure belonging to the PHY layer, i.e., the *Ethernet packet*. As a matter of coherence with the topics discussed in the rest of this Thesis, it is preferable to describe the structure of an Ethernet frame, rather than a packet. Such structure contains seven fields, as shown in Figure12. In the following, a brief description of each field is provided:

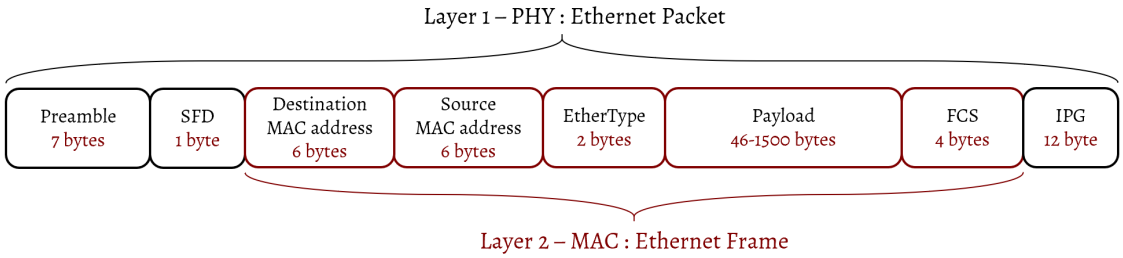

Figure 12: Ethernet frame structure according to [36]. The Preamble, SFD and IPG

♔ Preamble

The Preamble is made of 7 bytes. It is an alternating pattern of 1 and 0 that allows the receiver clock to synchronize with the devices on the network easily [36].

♕ Start Frame Delimiter (SFD)

The SFD is just 1 byte long, and its purpose is to mark the subsequent arrival of the frame encoded information. The value of this byte is chosen to break the previous sequence of 1 and 0; therefore it ends with two consecutive 1-bits.

♘ Destination Address

The Destination Address is a 6-byte field, identifying the station(s) that is supposed to receive the frame. The Least Significant Bit (LSB) of the first byte is referred to as Multicast bit. If equal to 1 it indicates that the address targets a group of stations, whereas if 0 it defines an individual address. The first three bytes are referred to as Organizational Unique Identifier (OUI), and they identify a product manufactured by a specific company. The company has to purchase the desired address directly from IEEE [37].

♘ Source Address

The Source Address is, once again, a 6-byte field, which has the same structure as the Destination Address. This time, however, it identifies the sending station. Moreover, the address indicated in this field is always identifying a single source; therefore its Multicast bit is always 0.

- EtherType

The EtherType field consists of 2 bytes that identify the protocol of the encapsulated data, coming from the upper layer in the OSI stack. In the previous version of Ethernet Standard [38], this field was storing the number of data bytes contained in the data field of the frame. That is the reason why all the EtherType codes are higher than 1500, that is the maximum allowed Payload size.

- Payload

The Payload is a sequence of at most 1500 bytes of any value. According to the Standard [36], if the length of this field is smaller than 46, its size must be extended by inserting a filler, to bring the payload length to a minimum of 46 bytes.

- Frame check sequence (FCS)

This 4-byte field contains a check sequence of bytes, aimed to be inspected by the receiving MAC after each transmission of data, to ensure the absence of any transmission errors. The algorithm used is the 32-bit cyclic redundancy check (CRC32), and it is run by the sending MAC. Such FCS is computed over the whole frame, except Preamble, SFD and indeed FCS field.

- Inter Packet Gap (IPG)

The IPG is not a sequence of bytes containing data. According to [17] there must be a minimum temporal gap between the transmission of two consecutive packets, to allow, possibly, the receiver clock to be synchronized again to receive another one. Such temporal gap must be equal to the time necessary to transmit 12 bytes; therefore it tightly depends on the speed of the link.

Chapter 4

Case Study - Preliminary Approach

The core part of this thesis, as written in its title, is the design of an *Ethernet Packet Processing Unit*, that is an architecture able to read data through an Ethernet channel and make decisions upon their reading. Such decisions involve possibly the modification or even deletion of data in a *transparent way*, i.e., without any trace of delay measurable from the outer world. During the six months of development, three different strategies were followed according to both the availability of additional components for the board and the acquired level of expertise. Such components are nothing but Ethernet modules designed by Knowledge Resources (namely, KR-LAN-A1). These small I/O peripherals are mounting a Marvell 88E1116R PHY[39], connected to an Ethernet RJ45 interface from one side. Then, on the other side, they expose to the outer world an RGMII interface. Still, on the same modules, there are eight LEDs: four of them are red colored, whereas the other four green. Figure 13 summarizes all these pieces of information.

Briefly, the different approaches to develop this thesis can be summarized according to the number of external Ethernet modules connected to the FPGA and the activity of the processor.

- No additional Ethernet modules, Processor playing an active role
- One additional Ethernet module, Processor playing an active role
- Two additional Ethernet modules, Processor not playing an active role

The first two approaches will be discussed in the following chapters, and they show

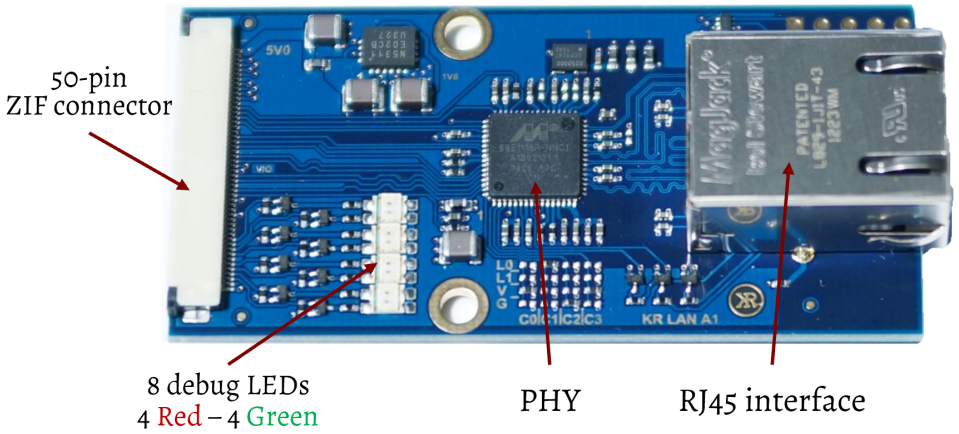

Figure 13: KR-LAN-A1 module. Picture taken from the corresponding product page [40]

many similarities in the design: the second strategy is an extension of the first one because they follow the same principles; however, it introduces interesting insights, useful for the development of the last design. The last element in the list, on the other hand, will be explained in a separate chapter, since it led to the final proposed design and it is based on a different approach. Each design will be explained starting from the physical setup, and then each checkpoint will be described from both software and hardware (PS and PL) point of views.

> ***Warning:*** *In the following sections the numbers identifying memory addresses, or dealing with byte dimensions are often expressed in hexadecimal (HEX) units. Whenever this is the case, the "0x" prefix precedes the HEX number. Therefore, the reader should know that, for example, the number 0x600 is equal to* $6*16^2 + 0*16^1 + 0*16^0 = 1536$

4.1 Initial Setup

As explained in Chapter 2.2 the device chosen for this research is the KRM-3Z7030 board, coming along with KRC3701 carrier kit. In order to build a packet processing unit, first of all, it is necessary to understand how data are traveling across the wires and then try to read them. The module board contains an Ethernet port that is routed to the PS through the MIO pins. In order to access network data, the Cortex A9 processor will play

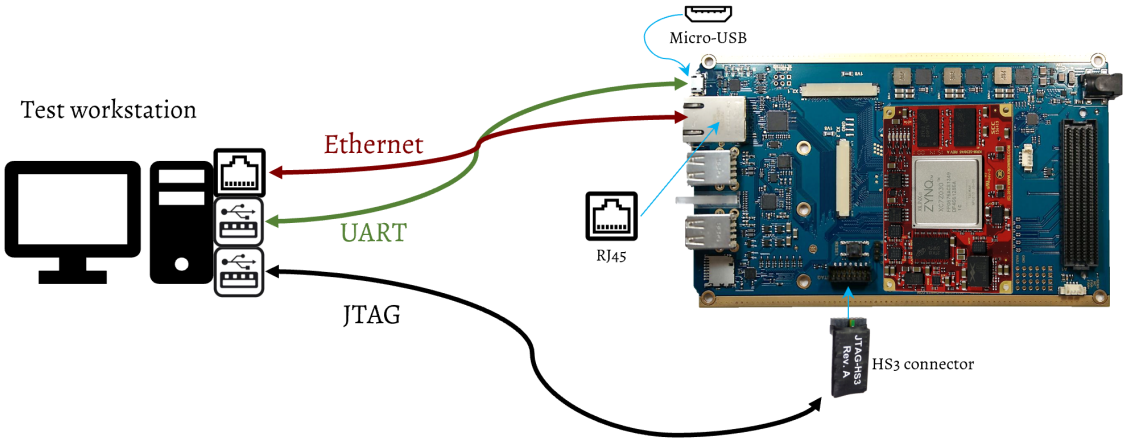

Figure 14: Physical setup showing all the connections between the board and the test workstation

an active role, being it the only processing unit in the PS. The board is connected to a test workstation through three different interfaces, as shown in figure 14.

- Ethernet, via the RJ45 connector
- UART, via micro-USB connector
- JTAG, via HS3 connector

Such test workstation can generate network traffic and push it through its Network Interface Card (NIC). The software used for this purpose is OSTINATO[41]. Moreover, the UART console allows the user to interact with the FPGA through a serial terminal. Eventually, the JTAG port is necessary to program both PS and PL.

4.2 Alternative “Echo” Application

A reasonable approach to start interacting with network traffic is indeed the design of an echo application. The operating principle is straightforward: each and every packet sent to the board is replied back to the sender (thus behaving like an echo). The LWIP library looks like the best candidate for this purpose and, after a quick glance among the proposed examples, it is easy to find one containing a simple echo implementation. The reader should always remember that the final purpose of this thesis is to interact with the internet traffic, possibly manipulating the information transmitted. Therefore, after

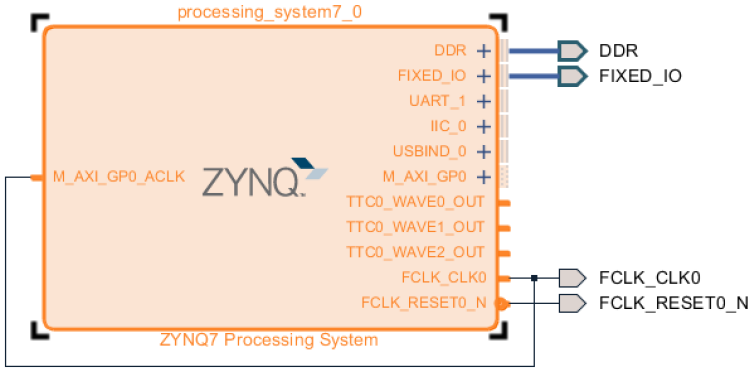

Figure 15: Block design of the first echo application

having understood the working principle of such an application, it seemed reasonable to set as a first goal a simple manipulation of the internet traffic: say, turning upside down the stream of data. That was not a difficult task to be done. First of all, the hardware block diagram does not need any particular configuration, as shown in figure 15. The processor is therefore fine with the PS resources only; thus the PL block design will be empty.

Now, after the inspection of the code proposed by the lwip library, it is not trivial to deduce the correct order of operations upon the reception of a new Ethernet packet. Everything starts with an interrupt request issued by the MAC controller[1], which is notifying the processor. Then, the processor issues a request to memory, asking for available space[2]. If there is enough for a packet, considering as a worst case scenario the maximum allowed size of 1500 bytes, the packet is accepted and stored in memory [3]. That is a standard procedure whenever dealing with data coming from the network. First, there should be enough space in memory to host such data; then they could be stored and eventually edited or re-transmitted. In this case, the re-transmission function is called upon the successful reception of data, since, being a simple echo application, it is re-transmitting the packet without any modification[4].

In order to match the purpose explained before, it is necessary to introduce manipulations to the data before their re-transmission. Right now, there is no reason to edit

[1] [42, line 78]
[2] [43, line 390]
[3] [42, line 63]
[4] [42, line 68]

any other section but the *payload* of the received Ethernet packets. That is preventing any accidental mistake, and after all, there is not even the need of editing other information like sender and receiver addresses, because it is still an echo application. Those two addresses are simply swapped since the sender becomes the receiver and vice versa. Luckily, the Ethernet packets are saved in an ordinate structure; therefore it is easy to retrieve the portion of data of interest. In particular, the useful pieces of information stored in such a structure are the *payload* and its length. The approach followed to handle the data is pretty straightforward, and is well summarized here below:

- Create a string `new_payload` with the same length of the payload
- Copy the original payload to `new_payload`
- Apply a function on `new_payload`
- Replace the original payload with `new_payload`

Since it is pretty easy to define a function operating on a *char* array whose length is known in advance. Three different functions were coded in order to preliminary manipulate the data. The topic of cryptography was already taken into account since two of those functions are using a well known, but at the same time obsolete, algorithm.

> ***Question:*** *What is the purpose of implementing an obsolete algorithm in a security context?*

The reader might wonder this question, and he would be perfectly right to point it out. The answer is again straightforward. The way this whole project was approached looks like the construction of a house, block by block. It is meaningless to start building one room after the other, in perfect shape, because it could be damaged or even redesigned before the skeleton of the house is ready. Therefore, the reader should be aware that before building complicated algorithms to handle data, there is one primary goal, that is to make the transfer of data actually work.

Here in the following are shown three elementary functions, coded on purpose, to be implemented in this preliminary code:

```
void reverse (char * in, char * out)
```

The function above is indeed elementary. Starting from the `in` pointer, that will be represented by the payload, it first scans the array until the end (`'\0'`). Then, it updates the position of such pointer, and it copies each byte of the input on the output array, moving backward until its starting point.

```
void caesar (char * in, char * out, int caesar_shift)
```

The second example is a little different, since it implements the very simple Caesar cipher[44]. The user has to provide the integer variable `caesar_shift`, that is nothing but a constant offset applied to all the characters present in the `in` string. If such offset is, say, equal to 3, the string "FPGA" will be transformed into "ISJD" since the 3rd character after 'F' is 'I' and so on.

```
void vigenere(char * in, char * out, char * vigenere_word)
```

This last function implements the Vigenère cipher[45]. The user has to provide the pointer to a char array `vigenere_word` where is saved the secret. The Vigenère cipher can be considered like a more general Caesar cipher, since the offset applied to each character is not constant but depends on the input `vigenere_word`. Considering again the previous example with the string "FPGA", if the chosen secret is "BIT", the output will be "GXZB". To understand better, it is useful to look at figure 16. First of all, the letters of `vigenere_word` are translated into numbers, from 'A' = 0 to 'Z' = 25. Then, letter by letter, the corresponding offset is applied to the input characters. Whenever the last letter of the `vigenere_word` is reached, the encoding procedure repeats starting from the fist. Thus, 'F' + 'B' = 'G', 'P' + 'I' = 'X', 'G' + 'T' = 'Z' and finally 'A' + 'B' = 'B'.

4.2.1 Results and Conclusions

The board behaves as expected. The CPU entirely handles the code, and loops back the packets with the chosen modifications. After this preliminary approach, it is necessary to make one step further as regards the management of resources. The reader should always remember that the final goal is to actively use the FPGA logic to speed up the process and

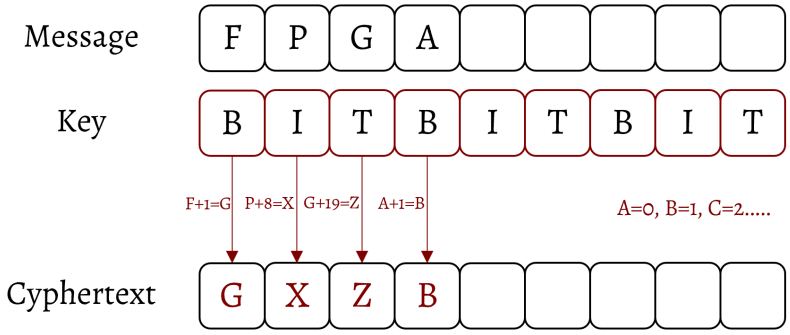

Figure 16: Encryption algorithm according to the Vigenère cipher

eventually program it to offload the most of computational effort to hardware. It might be interesting to edit the data inside the PL, in order to offload part of the processor tasks.

A reasonable starting point is, therefore, memory management. The best option available in the PL is the so-called Block RAM (BRAM), already explained in Chapter 2.1.2. In the previously run application, the incoming packets are temporarily stored in the system memory thanks to the `malloc` functions mentioned before. This implementation is excellent for resource optimization because it stores the data in memory in a dynamical way: to say it better, the physical memory location where data is stored is not necessarily contiguous, because the software tries to fill all the available space. Although there seems to be nothing wrong with it, from the point of view of hardware development it is quite difficult to handle such a chaotic structure. Thinking of a simple operation to be offloaded to hardware logic, if the final storage location for data is not known a priori how can such data be effectively manipulated? The only way to do that would include the instantiation of an additional component, able to keep track of the written physical memory addresses (after the `malloc` request) and to provide this information to a re-configurable hardware component. As a result, this procedure was considered not elegant enough to start the development of a hardware project and therefore abandoned in favor of a different strategy, well described in the following chapter.

4.3 Frame Repeater

The most sensitive point of the previous application was the memory allocation strategy (i.e., `malloc` function), that was not defined by the user but by the lwip library. Such library is very optimized to work at IP level and is available through the Software SDK (also online [46]): all the data from the network are stored in several variables, among which the source and destination IP addresses, the payload length, etc. However, it will be difficult for the PL to come into play, being all the data handled neatly by the PS. Hence, a new idea was born, with the aim of inspecting network packets from a lower level of abstraction (also lower in the OSI stack), that is to build a *Frame repeater*. From a very high-level perspective, the working principle of this new application is the same as the previously mentioned one: as seen in chapter 4.2, the goal is to reply back to the sender a packet received over the network, possibly including some modifications on the data. However, there are a couple of differences that play a significant role in the design of such an application and must be taken into account.

1. *The lwip library is dismissed, in favour of XEmacPs driver*

That is probably the most significant change. First of all, it means that there is no more interest in all the pieces of information encapsulated in a network packet, but only the raw data extracted by the Gigabit Ethernet MAC Controller (GEM) upon each transaction (both Transmission, TX, or Reception, RX). Such data are nothing but Ethernet frames without FCS (recall Chapter 3.4). Moreover, the XEmacPs driver allows the user to operate on the GEM largely, including memory transfers. Indeed, the lack of a pre-made library implies that even the simplest functions, like data storage, must be manually coded. In this specific case, the idea of using the processor alone to store the incoming data in memory is unfeasible, especially when the traffic rate is very high. The processor speed (1 GHz) is not enough to guarantee a fast transfer from the MAC controller to the memory, and, if it were ten times faster, it would be still not enough. Due to the nature of this architecture, namely, a Von-Neumann type as recalled in Chapter 1, the processor is a clear bottleneck

in memory transfers. However, luckily, the GEM can exploit a fascinating feature which can fix this issue.

2. *Direct Memory Access (DMA) and memory transfers to BRAM*

As said before, the main issue is related to memory transfers. That happens because for each byte of information the processor has to initiate the communication with the memory, wait to receive an interrupt from memory as soon as the operation is completed and start again with the following byte until the transaction is done. That makes the processor busy for the whole duration of the transfer. However, if peripherals are supporting DMA, it is a completely different story. This feature allows such peripherals to access main system memory without the processor intervention. The CPU, therefore, is first initiating the transfer, then it can continue with the other operations while such transfer is going on. Eventually, it will receive an interrupt from the DMA controller, informing that the transaction is done. Indeed, this feature is helping a lot the processor to keep a good pace despite the high traffic rate, however, there are some additional steps to be taken into account, to configure the DMA engine properly. A good explanation will be provided in the following section. The reader should now question what is the relationship with BRAM because it is clearly highlighted in the header of this paragraph. The reason is simple to be understood. Being the DMA engine helping the CPU in memory transfers, of course, this unit can write both to the On-Chip system memory or to the BRAM; it only needs the peripheral memory-mapped address. Conversely to the previous case in 4.2, the data are now compact since there are no more predefined structures nor allocation strategies. It is sufficient to store the whole stream of data in the BRAM and subsequently work on it. In this way, the PL gets involved, allowing the development of a processing unit in the FPGA logic-fabric.

4.3.1 Gigabit Ethernet MAC Building Blocks

The Gigabit Ethernet MAC controller is made of three building blocks (three for each side, namely RX and TX):

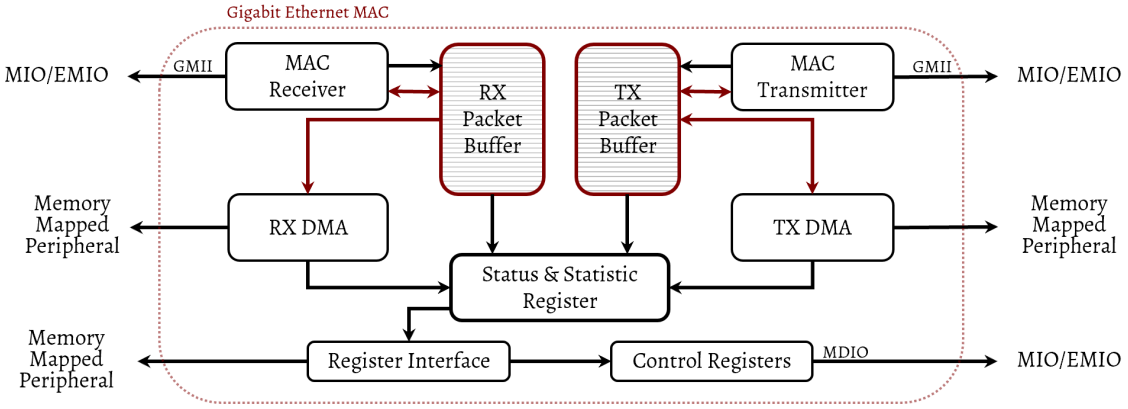

Figure 17: GEM architecture. The picture was drawn on the basis of the user manual contents [47]

- MAC controller
- Packet Buffer
- Ethernet DMA controller

To understand better how the different blocks depicted in figure 17 are interacting with each other, it is useful to introduce a simple example. Suppose that the FPGA is receiving a network packet. First of all, the PHY sends a notification signal to the MAC controller as soon as the first byte of data is received from the network. Such a signal will be immediately followed by the stream of the whole packet, through the RGMII interface. In order to properly handle those data, the MAC controller has to place them in the Packet Buffer temporarily, that is nothing but a FIFO[5]. Such FIFO must be emptied soon, to avoid the risk of overfilling in case of high traffic rate. The DMA engine will start playing soon; however, it must be instructed on where to move the buffered data. Solving this problem is not straightforward, requiring a high level of robustness for high data rate transactions. The DMA needs two new spaces in memory: a Buffer space, where to move the packets from the FIFO, and a Buffer Descriptor list. Such a list contains the starting memory address of the buffer space where the packet will be moved and its length. There are also other pieces of information stored in the Buffer Descriptor List; nevertheless, further explanations will be given in paragraph 4.3.2. The GEM implements this architecture both

[5]First In First Out (FIFO): it is a way to represent the storing process inside a buffer memory, in which the first object entering the buffer is also the first leaving. It might be compared to a water pipe.

for RX and TX side, so in conclusion, there will be two Buffers and two Buffer Descriptor Lists; two MAC controllers (Transmitter and Receiver) and two Packet Buffers (FIFO, TX and RX).

The operational flow described until now can be summarized in this way:

1. A network packet is about to be received
2. MAC controller (Receiver) enables the Packet Buffer (FIFO, RX) to be filled
3. MAC controller (Receiver) triggers the DMA upon reception of the whole packet
4. DMA reads the first available address in memory from the RX Buffer Descriptor List
5. The data is moved to the RX Buffer
6. The corresponding status registers in the RX Buffer Descriptor List are updated
7. DMA raises an interrupt request to notify the CPU that the transfer is done

It is necessary to explain better now what happens after point 4. The RX Buffer is divided in chunks whose size is 0x600 bytes (i.e., 1536 bytes). The Buffer Descriptor list must be seen as a collection of all the starting addresses in memory of such chunks. However, this list must be programmed by the user in advance, before the operations start. There are no restrictions on the number of elements that such list should contain, it all depends on the amount of memory reserved for this purpose.

As regards the opposite flow, from the memory to the Ethernet cable, it is more or less the same, but reversed, that is:

1. A network packet in the TX Buffer is about to be transmitted
2. The pointer to its address in memory is written to the TX Buffer Descriptor List
3. DMA is triggered to start the transaction
4. DMA moves the stored packet from the TX Buffer to the Packet Buffer (FIFO, TX)
5. The corresponding status registers in the TX Buffer Descriptor List are updated

6. MAC controller (Transmitter) enables the Packet Buffer (FIFO, TX) to be emptied

7. DMA raises an interrupt request to notify the CPU that the transfer is done

4.3.2 DMA Transactions

As explained in the previous chapter, the RX (and TX) buffers are made of chunks, 1536 bytes long. That is not a chance, since the maximum size of an Ethernet frame, without considering preamble or FCS is in general 1514 bytes[6]. From this specification, it is possible to instantiate the right amount of memory needed by the application to run properly. Suppose the Buffer Descriptor list has five entries: the corresponding Buffer, therefore, will be 5*1536 = 7680 bytes long, that can easily fit in 2 36 Kb Block RAM units (equivalent to 8192 bytes).

Now, the focus should be moved on the Buffer Descriptor List. First of all, each entry is written over 64 bits (referred to as Buffer Descriptor). The first 32-bit word is mostly containing the pointer to the initial address of one among the previously mentioned chunks, whereas the latter 32-bit word is containing different pieces of information, depending on the nature of the Buffer Descriptor List (RX or TX). In figure 18 those two words are referred to as Buffer_Descriptor_#[1] or Buffer_Descriptor_#[0]. The DMA accesses the Buffer Descriptor List in order to fetch the correct instructions; however, this will not be possible without the *Queue Pointer Register*, belonging to the GEM registers. This register plays a simple but important role, because it shows the DMA the next pointer to be processed, and, indeed, there are two of them: one for RX and one for TX side. After each transaction is concluded, such pointer is updated with the following Buffer Descriptor. Among the registers stored in the Buffer Descriptor List, there is one indicating the last entry in the list. Such register, called *wrap*, makes the Queue Pointer Register start over from the first element (Base Address), as soon as the "last" transaction is done[7].

[6] According to IEEE 802.3

[7] Said better, it is the transaction involving the last element in Buffer Descriptor List

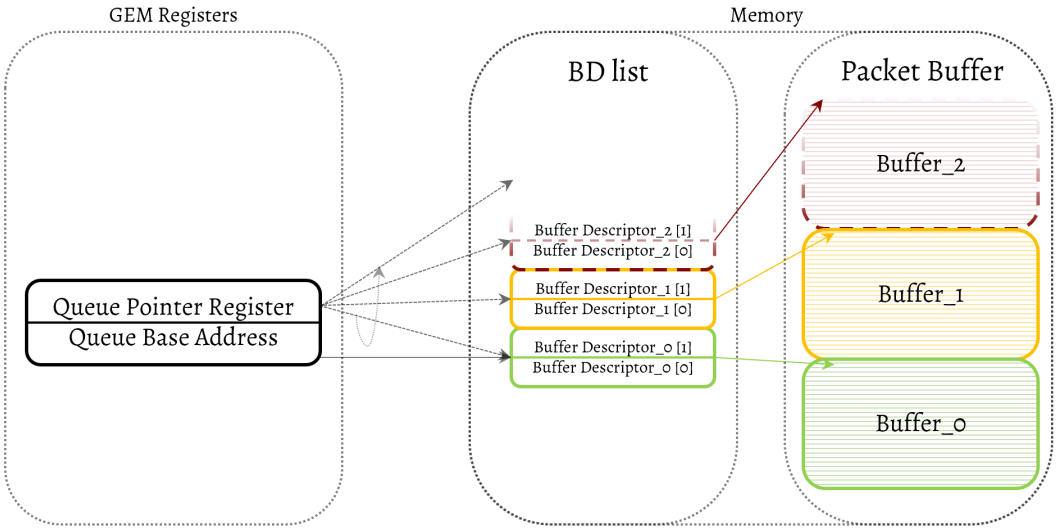

Figure 18: Ethernet DMA controller under the scope: building blocks

1. *Buffer Descriptors - Remarkable Entries*

As said in the preiovus paragraph, the Buffer Descriptors will show slight differences depending on their RX or TX nature. However, it is sufficient for the reader to understand the things they have in common. First of all, they have a *length* field, that is telling the DMA how many bytes are supposed to be moved from the pointed address onwards. Then, there is the *wrap* bit, already explained before, that is telling the DMA to start over with the Base Address for the following transaction.

An interesting bit is the *ownership* (belonging to RX Buffer Descriptor list) or *used* (belonging to TX Buffer Descriptor list). These two elements are playing same jobs although their name is different. Whenever new data are moved to the Buffer pointed by one Buffer Descriptor in the RX list, the DMA writes '1' to the *ownership* bit. That is equivalent to indicate that such Buffer has been used, and thus given to software, which is now "owning" it. This means that it is not anymore available for hardware. The reader should understand that the two words hardware and software, as regards this paragraph only, are used to indicate what is belonging and is controllable by respectively the DMA and the CPU. Analogously, regarding the TX side, the *used* bit is set to '1' as soon as the DMA successfully transmits data. Both for RX and TX side, this bit must be cleared by software; otherwise, the DMA will not be allowed to receive or transmit data on that specific buffer, being it

already marked as "used"[48].

This paragraph is sufficient to understand what is the necessary information provided to the DMA engine to work correctly, however, its working principle is still not explained well. The CPU is not needed to make the DMA work in hardware; however, it is necessary to ensure the right sequence of operations. The reader should always remember the goal of this preliminary approach, i.e., to build an application able to receive a stream of packets from the network and loop them back to the sender, possibly modifying their payload. It is clear that the DMA and their components must always be ready to receive, store and send back; therefore, the whole engine must be able to work without ever reaching an end. Thanks to the XEmacPs driver, the CPU is allowed to interact with the DMA engine, by setting the status of the Buffer Descriptors in such a way that they can cycle forever.

2. *Buffer Descriptors - State Machine*

At any moment, Buffer Descriptors can be in one out of four different states. A specific function belonging to XEmacPs driver mediates the transition from one state to another, and this is briefly summarized in figure 19. Being the whole process cyclic, the reader should think at the Buffer Descriptor List as a merry-go-round, with a finite set of seats (representing the Buffer Descriptors), however, in literature this is usually referred to as *Buffer Descriptor Ring* (BD Ring)[47].

The function used to create a BD Ring is `XEmacPs_BdRingCreate()` and the most relevant inputs taken by such function are the starting point of the Buffer Descriptor List and the desired number of entries (i.e., Buffer Descriptors). This function must be called once per side (RX and TX), thus allowing the creation of two different and independent BD rings. This function generates the number of Buffer Descriptors according to the corresponding passed argument, and leaves them in a state referred to as *Free*. This indicates that the BD is not controlled by the user application and it is not yet available for any DMA transaction. Thanks to the function `XEmacPs_BdRingAlloc()`, the user application can gain control over the Buffer Descriptors and thus be able to program them for a specific DMA transaction. This is also making the state switch from *Free* to *Pre-Process*. In

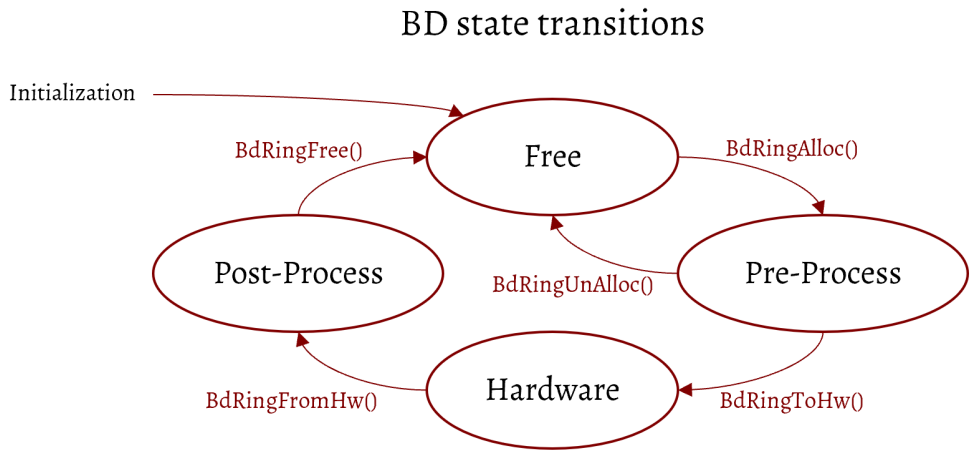

Figure 19: Buffer Descriptor state transitions: the only inputs able to change state are given by software through XEmacPs driver

this state it is possible to specify the address of RX or TX Buffer that the Buffer Descriptor should point to, respectively, thanks to the function `XEmacPs_BdSetAddressRx()` or `XEmacPs_BdSetAddressTx()`. Now the BDs are ready to be committed to hardware, and this is made possible thanks to the function `XEmacPs_BdRingToHw()`, that also allows the state transition from *Pre-Process* to *Hardware* state. In this state, the hardware controls the Buffer Descriptor, until the DMA transaction is concluded.

Then, the only way the user application can claim back the BD is through the function `XEmacPs_BdRingFromHwRx()` (or `XEmacPs_BdRingFromHwTx()` depending on the nature of Buffer Descriptors, RX or TX), allowing the transition from *Hardware* to *Post-Process* state. Here, the user application checks the information encoded in the Buffer Descriptor in order to understand whether the transaction was successful or not. In the end, after the Buffer Descriptors are post-processed, they should be made ready to be used again and therefore should be returned to their initial state: the function `XEmacPs_BdRingFree()` is exactly designed for this purpose.

There is still one function, `XEmacPs_BdRingUnAlloc()`, showed in figure 19 but not mentioned before: it is useful whenever there are more BDs in Pre-Process state with respect to the desired amount. Such a function makes merely the BDs again Free.

3. *DMA - Interrupt Management*

The main exchange of information in hardware between the DMA engine and the CPU occurs employing interrupts, i.e., signals that require high-priority from the processor, making it stop its operational flow to first execute a function called *interrupt handler* (also interrupt service routine, ISR). After the interrupt handler is done, the processor can resume normal activities[49]. Whenever a DMA transaction is concluded, an interrupt is asserted, but up to now, the processor is yet not able to understand whether such interrupt is reporting a successful transaction or an error. Such interrupt is propagated to a block called Generic Interrupt Controller (GIC), belonging to the PS. The GIC is a fundamental component, receiving all the interrupt signals from the I/O Peripherals within the PS; moreover, it has to manage the priority of such interrupts correctly, before feeding them to the CPU. The GIC is hence the last element in the chain that interrupts the CPU, and also indicates the interrupt. In this way, the appropriate ISR will be called. As regards the Ethernet DMA, the interrupt conditions that can be raised are: *received*, *send* and *error*. Whenever there is an occurrence of one among such interrupts, the GEM asserts a single interrupt signal (namely, IRQ), and the application calls a single ISR. This is proved by the function included in one of the examples of XEmacPs driver[8], which is also defining the "general" interrupt handler, called `XEmacPs_IntrHandler`. Such handler can infer the appropriate interrupt condition, according to the GEM registers. As soon as the condition is known (received, send or error), the right callback handler is called (respectively, `RecvHandler`, `SendHandler`, `ErrorHandler`).

After this extensive overview over the theoretical background lying behind the XEmacPs driver, the reader will go through the design of the frame repeater, starting from a hardware description, and eventually the software description.

[8] `xemacps_example_intr_dma.c`, line 1089

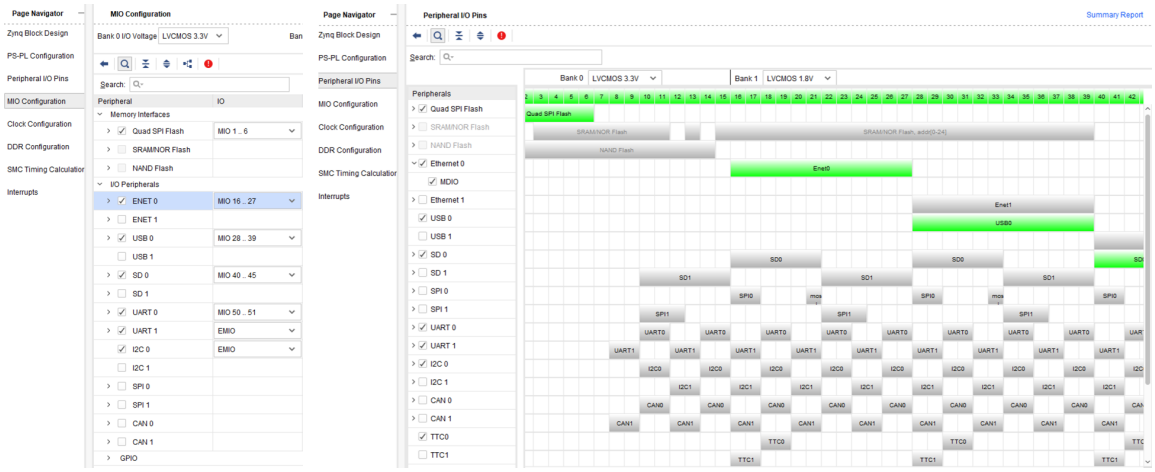

Figure 20: PS configuration for the preliminary hardware design. The reader should notice that the Ethernet interface (ENET0) is enabled and routed through the MIO pins

4.3.3 Hardware Design

The preliminary design described in the following is quite simple; however, the explanation will not be concise, as the reader should clearly understand why the components are connected in that specific fashion. The main design blocks under the scope are:

- ZYNQ7 Processing System
- AXI BRAM Controller
- Block Memory Generator

The first block is the director of the orchestra since it sets the initial configuration for PS. It can be seen as a wrapper for the whole Processing System, able to properly enable all the Input/Outputs, clocks, and interfaces and eventually make them available for the PL (see figure 20). This includes enabling the Gigabit Ethernet MAC and configuring it to be routed through MIO pins, exploiting the mounted PHY and RJ45 connector on the carrier board. As shown in figure 21, only the two Master General Purpose AXI interfaces are enabled, interfacing PS with PL. There are two AXI BRAM Controllers connected to such interfaces via an *AXI Interconnect* (one per each interface). The purpose of an AXI Interconnect is, as the name suggests, to interconnect a set of master peripherals to a set of slaves, that are communicating through the AXI4 standard [19]. Each of the slave

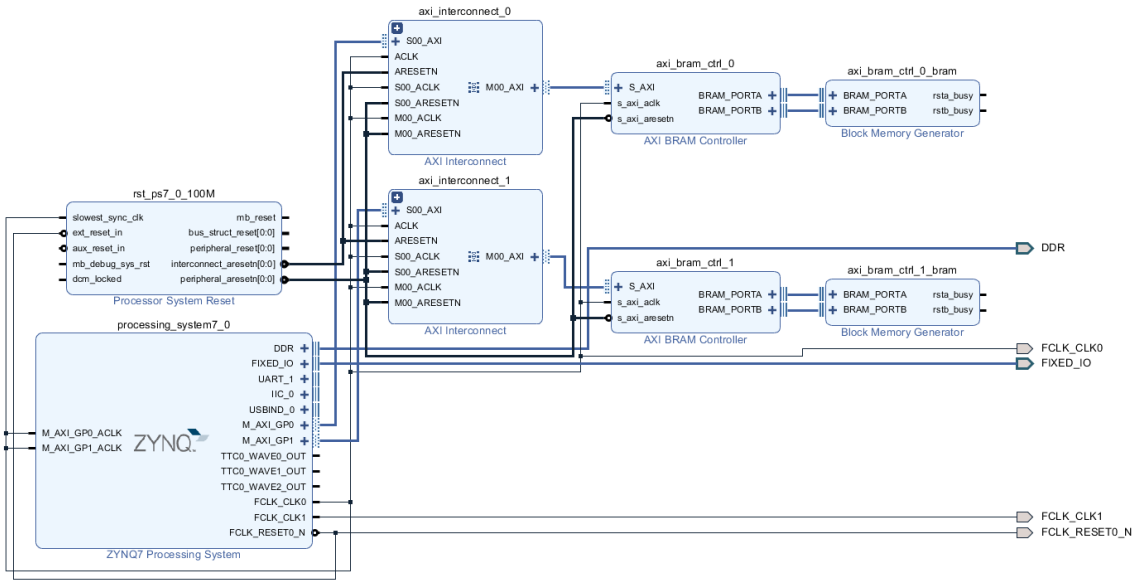

Figure 21: Block Design of the Frame Repeater, in its first version, for the preliminary approach

peripherals is assigned an address offset and a range, allowing the master to interact with them without any conflict. Indeed, an overlap in the address ranges of peripherals is not allowed. In this application, the PS represents the master, whereas the BRAM Controller represents the slave. The PS needs such BRAM controller to read and write data from/to the Block RAM, that is instantiated by the Block Memory Generator. Hence, one can say that such a controller is also a master from the Block Memory Generator point of view.

The overall amount of BRAM for this design was chosen to be relatively small (8 KB per block). It is always reasonable not to waste resources across the fabric, but especially when dealing with such kind of memory, it is not easy at all to handle all the interconnections correctly, guaranteeing the correct time propagation for all the instantiated elements. Each elementary cell size is only 36 Kb, so in this case, only four elementary cells were instantiated (two per block). That being said, the primary purpose justifying the memory instantiation is the need of storage for both the RX and TX buffer and their relative Buffer Descriptor Lists. However, since in this case the goal is to reply a packet back to the sender, it is reasonable to share the same memory location both for RX Buffer and TX Buffer. Of course, the Buffer Descriptors Lists will have distinct memory locations. Eventually, the two different blocks of BRAM instantiated in this design are used respectively to store the two Buffer Descriptor Lists (in two separate but contiguous regions) and a

"generic" Packet Buffer (both RX and TX). The offset address for these two peripherals is chosen to be respectively 0x80000000 and 0x40000000.

Before continuing to the software definitions, the reader cannot neglect the clocks and reset configuration. As regards the reset, there is nothing special to remark: when the system is turned on, the PS sends an asynchronous reset signal through the block called *Processor System Reset*. This signal is made synchronous to the chosen clock (in this case, all the components are run by the same clock) and propagated to all the blocks instantiated in the PL for a correct initialization. As regards the clock, it is set to run at 100 MHz. Although it seems to be quite slow, the choice is justified by the preliminary hardware design, that does not yet include optimization, but rather a rough working draft. The reader should always remember that the higher the clock speed in an FPGA, the harder is the effort to place and route all the connections between hardware elements without violating the timing constraints.

4.3.4 Software Design

The starting point for designing the software, and also, the best resource for understanding how to use the XEmacPs driver functions is one of the examples that comes along with such driver, that is `xemacps_example_intr_dma.c`. This paragraph will be divided into two parts, explaining first how to set up correctly the environment and then how to run the main program.

1. *Setup*

The first task to run as soon as the system starts up is erasing the BRAM memory and set up the environment for capturing network packets. This includes the following steps:

- Find out the GEM memory-mapped address
- Initialize a `XEmacPs` structure, containing the GEM configurations and its address
- Set the Ethernet TX Clock to the maximum speed of 125 MHz

As regards the first point, it is similar to the previous problem with the BRAM Controller: the CPU operates on the configuration registers, provided it is given the right address offset, i.e., the peripheral address. In a Zynq®PS, usually, the GEM0 is mapped on the address 0xE000B000. For what concerns the clock speed, since 8 bits are transferred through the GMII interface per clock cycle, the overall speed will be $8\,\text{b} * 125\,\text{MHz} = 1\,\text{Gbps}$, that is the maximum line rate allowed by the PHY. To configure the Ethernet TX Clock, the user has to operate on the System-Level Control Registers (SLCRs). In the example mentioned before, there is already a function called `XEmacPsClkSetup` that contains all the necessary steps to accomplish this goal.

Now, it follows an interesting design choice, that is to configure the MAC *promiscuous mode* option. Usually, the board should run with a specific MAC address (that is set by the function `XEmacPs_SetMacAddress`), so that the other elements inside the network can address it. The reader should recall that the MAC controller does not accept the packets that have a destination address not equal to the board's address. However, being this application meant to be the first step towards a Packet Processing Unit, it should be able to monitor the whole traffic stream without exceptions. That is why the MAC *promiscuous mode* exists, to prevent the MAC from excluding the packets that are not targeting the board, and making it accept them all. The function below is used to configure the MAC for multiple purposes, like enabling/disabling the computation of FCS, or enabling/disabling the reception of packets longer than 1516 bytes, etc. In this case, however, it is straightforward to understand the configuration by looking at the second argument of such function. The first instead, is the pointer to the GEM address.

```
XEmacPs_SetOptions(EmacPsInstancePtr, XEMACPS_PROMISC_OPTION);
```

The software then configures the GIC, as explained in Paragraph 4.3.2, by means of the function `XScuGic_Connect`. The reader should remind that it will connect a device driver handler, that is going to be called whenever an interrupt for the device occurs. Such device driver handler is able to perform the specific interrupt processing for the device. Therefore, by using the function `XEmacPs_SetHandler`, it is possible to assign a different routine (in this case, `XEmacPsSendHandler`, `XEmacPsRecvHandler` and

XEmacPsErrorHandler) per each different interrupt condition. Again, there are already a few lines of code in the provided example xemacps_example_intr_dma.c, doing this job.

```
XScuGic_Connect(IntcInstancePtr, EmacPsIntrId,
        (Xil_InterruptHandler) XEmacPs_IntrHandler,
        (void *) EmacPsInstancePtr);

XEmacPs_SetHandler(EmacPsInstancePtr,
            XEMACPS_HANDLER_DMASEND,
            (void *) XEmacPsSendHandler,
            EmacPsInstancePtr);
XEmacPs_SetHandler(EmacPsInstancePtr,
            XEMACPS_HANDLER_DMARECV,
            (void *) XEmacPsRecvHandler,
            EmacPsInstancePtr);
XEmacPs_SetHandler(EmacPsInstancePtr,
               XEMACPS_HANDLER_ERROR,
            (void *) XEmacPsErrorHandler,
            EmacPsInstancePtr);
```

There is one more component to be configured by software before launching the main program and probably one of the most important: the PHY. The registers of such component are configured through the Management Data Input/Output (MDIO) interface, which runs however with a specific clock frequency. Such frequency must not exceed 2.5 MHz as defined by the IEEE802.3[47, Pag.516] standard. Once again, one of the functions proposed by the example is exploited, to generate the appropriate clock, by dividing the frequency of the main PS clock.

```
XEmacPs_SetMdioDivisor(EmacPsInstancePtr, MDC_DIV_224);
```

In order to configure or view a register belonging to the PHY through the MDIO bus, there are two functions available: XEmacPs_PhyRead and XEmacPs_PhyWrite. They take the same arguments: the address of the target PHY (there could be multiple PHYs, as in the following chapters), the number of the register to be read or written and eventually a pointer to a variable where the read data will be stored or the data to be written is located. The last steps before concluding this setup consist of:

- Setting the maximum speed (1 Gbps) for the connection
- Setting any additional TX or RX delays

- Resetting the PHY
- Triggering the Auto-negotiation
- Waiting for the link to be up

The maximum speed of 1 Gbps is achieved by writing 0x0140 in Register 0. As regards the second point, it will be discussed better in the following chapters; right now there is not anything relevant to comment. The PHY can add a skew of 2 ns to the data with respect to their clock, both on RX and TX path by writing 0x0070 in Register 21. After the reset event (write '1' on the MSB of Register 0), the PHY starts an Auto-negotiation[9] procedure with the link partner, sitting at the other side of the cable. This process is a standardized handshake through which the PHY finds out the capabilities of the partner and negotiates how to exchange information over the link, by choosing the speed and duplexing mode[47, Chapter §16-16.3.4]. In order to enable this procedure, one has to set the 13th bit of Register 0 to '1'. As soon as the handshaking is done, the 6th bit of Register 1 is asserted to '1'; hence, it is sufficient to poll such a bit before going on. The same principle holds for understanding if the link is up: one should simply poll the 3rd bit of Register 1 and wait until it is equal to '1'.

2. *Main program*

The set of instructions to be executed after the configuration setup can be divided into two distinct parts. First of all, the Buffer Descriptor Lists and the RX/TX Buffer must be initialized; then the program will be able to run forever, cycling through the ring buffers. The majority of functions used in this part come from the XEmacPs driver.

The first task consists in creating the Buffer Descriptor Lists by means of the function `XEmacPs_BdRingCreate`: such function takes as arguments the pointer to the corresponding GEM controller, the pointer to the structure of the list (RX or TX), the pointer to the address in memory where the user would like to initialize such list, and eventually the number of elements it should contain. As said in section 4.3.3 there are two available

[9] Recall section 3.3

locations in BRAM to store the BD Lists; hence, 0x40000000 is chosen to be the starting point for the RX list. The TX one belongs to the same BRAM; however, the starting point is placed at 0x40001000, that is at half of its width. In order to avoid unintended transmissions on behalf of the DMA, it is useful to set the *used* bit of all the Buffer Descriptors in the TX list.

That being said, the system is almost ready to enter the infinite loop. However, the boundary conditions are still not set. The reader should remember the relationship between the *Queue Pointer Register* and the *wrap* bit from section 4.3.2. These two elements must be respectively initialized and set up correctly before entering the infinite loop since they will be responsible for such an endless cycle. In particular, the two Queue Pointer Registers (RX and TX) are assigned the beginning of their respective lists (RX and TX), so that the first transaction will involve the first element in the list (whose address is referred to as `BaseBdAddr`). The wrap bit instead, must be set only on the last element of each list (whose address is referred to as `HighBdAddr`).

During the development of this software application there have been two main approaches depending on the number of buffer descriptors in the relative list: only one or a generic amount N. For simplicity, it is better to start describing the most straightforward application, with a single Buffer Descriptor per list, and then deal with the improvements in the next chapter. The reader should make a step back to the function `XEmacPs_BdRingCreate`, and remember to set the number of buffer descriptors to 1, both for RX and TX sides. This will also imply that the Buffer Descriptor List will have their `BaseBdAddr` coincident with their `HighBdAddr`.

In order to start the loop of operations it is first necessary to Pre-Process the RX BD: using the function `XEmacPs_BdRingAlloc`, the RX BD is ready to be committed to hardware; however, it is still not assigned any RX Buffer address. Thanks to the function `XEmacPs_BdSetAddressRx`, the starting point of such space in memory is given to the relative Buffer Descriptor, that is now ready to be committed to hardware via the function `XEmacPs_BdRingToHw`. The developer is in charge to set up the right RX Buffer address, even though in this case it is straightforward, being it coincident with the start-

ing point of the RX Buffer reserved space in memory. As mentioned before, the memory block starting at 0x40000000 is already assigned to the Buffer Descriptor Lists; therefore the RX/TX Buffer list is given the address 0x80000000. The system is now ready to receive packets, and it will actually do it, as soon as the interrupts on the RX path are ensured to be armed and the GEM is started.

```
XEmacPs_IntEnable(EmacPsInstancePtr, XEMACPS_IXR_FRAMERX_MASK);
XEmacPs_IntEnable(EmacPsInstancePtr, XEMACPS_IXR_RX_ERR_MASK);
XEmacPs_Start(EmacPsInstancePtr);
```

The flow diagram that summarizes the whole loop procedure explained in the following is depicted in figure 22, making it easier for the reader to follow. As soon as the link partner sends a packet, the MAC will trigger the DMA, which will move the data from the Packet Buffer (FIFO) to the RX Buffer pointed by the first (and only) available Buffer Descriptor. After that, the DMA will write the packet size, and the *ownership bit* in the relative Buffer Descriptor registers, as explained in section 4.3.1. If the transaction is successful, the DMA will trigger the *FRAMERX* interrupt (that will eventually run the `XEmacPsRecvHandler` callback), otherwise it will trigger the *RX_ERR* interrupt (eventually running the `XEmacPsErrorHandler` callback).

The software needs a reference system to keep track of the time evolution. Said differently, the CPU must know how long to "wait" for DMA before continuing the execution of the code: therefore, setting a flag in the `XEmacPsRecvHandler` callback seems a reasonable idea. The software will constantly poll for such flag, after having turned on the GEM, in order to be sure to resume the operations as soon as the DMA is done.

Remembering once again that the primary goal of this application is to send back the received Ethernet frame, it is now time for the TX path to be set. First of all, the RX Buffer Descriptor is revoked from hardware control, thanks to `XEmacPs_BdRingFromHwRx` function. Then, the TX Buffer Descriptor is Pre-Processed (`XEmacPs_BdRingAlloc`) and assigned the same address as the received packet (`XEmacPs_BdSetAddressTx`). There is still one missing piece of information, that is the size of the packet to be transmitted. This can be extracted from the RX Buffer Descriptor (`XEmacPs_BdGetLength`) and can be encoded in the TX Buffer Descriptor (`XEmacPs_BdSetLength`). In the end,

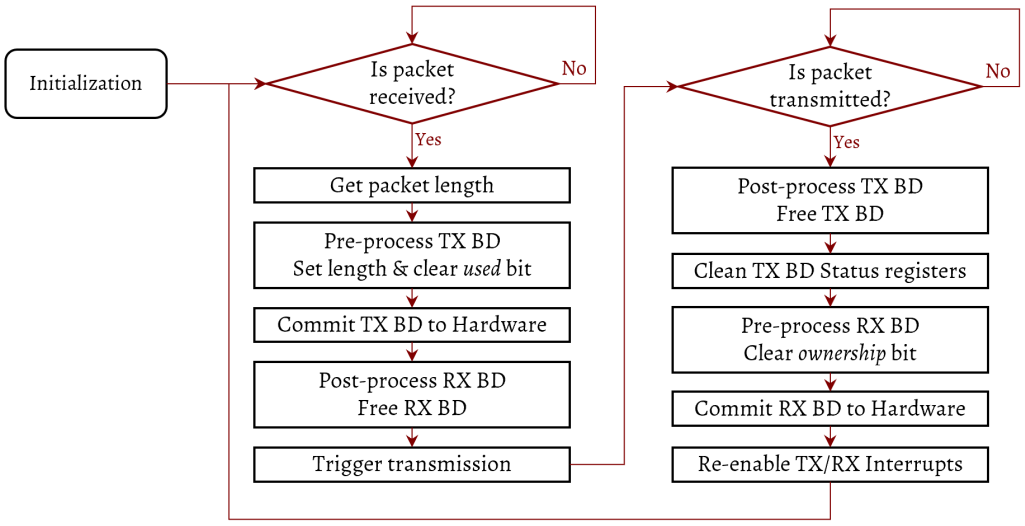

Figure 22: Flow chart representing the software decision steps at runtime

the previously enabled *used* bit must be cleared now, in order to allow the transmission (`XEmacPs_BdClearTxUsed`). The TX Buffer Descriptor is now ready to be committed to hardware (`XEmacPs_BdRingToHw`). The system is now ready to transmit a packet, and it will actually do it, provided the interrupts on the TX path are ensured to be armed, and the GEM transmission is triggered.

```
XEmacPs_IntEnable(EmacPsInstancePtr, XEMACPS_IXR_TXCOMPL_MASK);
XEmacPs_IntEnable(EmacPsInstancePtr, XEMACPS_IXR_TX_ERR_MASK);
XEmacPs_Transmit(EmacPsInstancePtr);
```

Analogously, if the transaction is successful, the DMA will trigger the *TXCOMPL* interrupt (that will eventually run the `XEmacPsSendHandler` callback), otherwise it will trigger the *TX_ERR* interrupt (eventually running the `XEmacPsErrorHandler` callback).

After the transaction is done, the system must be cleared and made again ready to receive a new packet. Therefore, the following steps are executed: first the TX Buffer Descriptor is Post-Processed (`XEmacPs_BdRingFromHwTx`), then both RX and TX BDs are returned to *Free* state (`XEmacPs_BdRingFree`). Eventually, the RX Buffer Descriptor is again Pre-Processed, its *ownership* bit is cleared (`XEmacPs_BdClearRxNew`) to allow a new packet to be saved and it is again committed to hardware.

This set of operations leads to an infinite loop, turning the board into a mirror, rather

than an "echo" server, able to reflect the incoming packets back. In order to include modifications to the payload, a couple of functions are defined, to be run between the successful reception of a packet and its re-transmission. They are operating directly on memory, and the simplest one is merely changing the *EtherType* field of the Ethernet frame. This is useful to read out the replied packets easily. The reader should remember that in the physical setup, the board is connected via Ethernet to a workstation, that is also able to generate network packets and monitor the traffic on the line. Theoretically speaking, this application should run correctly; nevertheless, the developer should analyze the resulting problems, and take into account alternative design choices to make the application run smoothly.

4.3.5 Troubleshooting: Problems, Solutions and Improvements

In the beginning, it is not easy to understand properly how the software works and it is difficult to understand the reasons why it is not performing correctly. This happens because some tasks cannot be debugged (like the DMA operations); hence, one should blindly trust the implemented drivers or libraries. Unfortunately, since they are still under development, or maybe due to limited usage by the community, there are still some known unfixed issues[47, Pag.537-538]. Such issues might generate unexpected behavior by the software and thus require the design of a patch, or a workaround.

One of the first issues found is concerning the double transmission of the same packet on behalf of the DMA, also documented in the user manual; however, the implemented workaround is a bit different from the recommended one. Whenever the FPGA receives a packet, two identical packets are leaving the board, instead of one. The code is slightly changed to fix the problem, as written in the manual, by adding one buffer descriptor to the TX list, and setting its *used* bit to '1'. No other changes are necessary.

Another issue is related with the incoming data rate. Whenever it is reasonably high, the processor is not able to empty and restore the lonely RX Buffer Descriptor in time. Therefore, the FIFO which is close to the MAC gets full quickly, causing soon a "*Receive Buffer not available*" error. There are two ways for solving this issue: the *cheating* way and

the *responsible* way. The first is not providing a way to solve the problem, but rather to completely circumvent it. The reader should have understood that the errors are showing up because of the `XEmacPsErrorHandler` callback. Such a function is called by the DMA that is in troubles while moving data between the buffers. Since there is not enough room to host the data buffered by the FIFO, the DMA raises up the correct identifier for such error, producing the corresponding interrupt. What happens if the DMA is by chance not able to launch any interrupts concerning errors in the reception step? Thanks to the function `XEmacPs_IntDisable`, it is possible to disable the DMA interrupt processing features temporarily. Such function is run as soon as the `XEmacPsRecvHandler` callback is executed.

```
XEmacPs_IntDisable(EmacPsInstancePtr, XEMACPS_IXR_FRAMERX_MASK);
XEmacPs_IntDisable(EmacPsInstancePtr, XEMACPS_IXR_RX_ERR_MASK);
```

Of course, this small cheat works, provided the user re-enable the interrupts before starting the loop again. As said before, this is not a natural solution: in case of high data rate, it is true that the system will not crash, but it will suffer from data loss.

As regards the so-called *responsible way*, the story is completely different. The most important improvement comes from the increase in the number of Buffer Descriptors in the BD List. In this specific application, due to the small amount of BRAM implemented, such number is set to be 5. Recalling that the allocated space in memory for each Buffer is 1536 bytes, it is trivial to say that in 8 kB there is not enough room for more. That being said, it is time now to change a little bit the code, but not from the conceptual point of view, which will be almost all the same as depicted in the flowchart of figure 22. The setup will be identical, and the function for creating the BD ring will take into account the new number of Buffer Descriptors, this time equal to five. After having set the *Queue Pointer Registers* and the *wrap* bits, it will follow the Pre-Processing, this time involving all the five elements in the list. Each of them will be assigned a different address in memory, all equally spaced by 0x600 bytes, i.e. 0x80000000, 0x80000600, 0x80000C00, 0x80001200, 0x80001800. Then they will be committed to hardware. Now, whenever a packet is received, the software can Post-Process it without wasting the following packets in line. This time there is

more room, so the board can store a bunch of five packets before giving up. Therefore, if the processor is able to post process them all in due time, the whole flow of data will be kept intact. One can think that it should be easy to implement this new architecture with the XEmacPs driver functions, however, this was not the case during the development of this application.

```
u32 XEmacPs_BdRingFromHwRx
    (XEmacPs_BdRing * RingPtr, u32 BdLimit, XEmacPs_Bd ** BdSetPtr)
```

The above-mentioned function, which is responsible for Post-Processing Buffer Descriptors, was found to be the cause of malfunctioning: a detailed description is provided in the following lines. The first argument of the function is the pointer to the RX Buffer Descriptor Ring; the second indicates the maximum number of Buffer Descriptors that the software is willing to Post-Process, the last argument is pointing to a variable where it will be saved the address of the first Post-Processed Buffer Descriptor. By merely reading those lines and the official documentation[10] it seems that there is nothing wrong in setting the *BdLimit* to be equal to the maximum number of Buffer Descriptors. In this way, the hardware will be able to Post-Process a subset (or even all) of them, no matter how many packets are received when `XEmacPs_BdRingFromHwRx` is called. The reader should remind that a necessary condition for an RX Buffer Descriptor to be Post-Processed is to have its *ownership* bit set to '1', indicating successful reception of a packet.

Unfortunately, the function seemed to work only when all the *BdLimit* Buffer Descriptors had their *ownership* bit set to '1'. This is causing a considerable limitation since the software hangs until the fulfillment of this condition. Indeed, it is not acceptable, since the application is intended to be real-time processing. So, in order to avoid such problem, an interesting workaround was coded, starting from the definition of the function mentioned above and involving `XEmacPs_BdRingFree` as well, since they work in a couple.

```
1 /* Equivalent to  XEmacPs_BdRingFromHwRx(&(XEmacPs_GetRxRing(
     EmacPsInstancePtr)),1, &NewBD) */
2 EmacPsInstancePtr->RxBdRing.HwCnt -= 1;
3 EmacPsInstancePtr->RxBdRing.PostCnt += 1;
```

[10] Even the comments in the file `xemacps_bdring.c`[50]

```
4
5  NewBD = EmacPsInstancePtr->RxBdRing.HwHead + EmacPsInstancePtr->
       RxBdRing.Separation;
6  EmacPsInstancePtr->RxBdRing.HwHead = NewBD;
7  if (NewBD >(XEmacPs_Bd *) EmacPsInstancePtr->RxBdRing.HighBdAddr)
       EmacPsInstancePtr->RxBdRing.HwHead -= EmacPsInstancePtr->RxBdRing.
       Length;
8
9  /* Equivalent to  XEmacPs_BdRingFree(&(XEmacPs_GetRxRing(
       EmacPsInstancePtr)),1, NewBD) */
10 EmacPsInstancePtr->RxBdRing.FreeCnt += 1;
11 EmacPsInstancePtr->RxBdRing.PostCnt -= 1;
12
13 NewBD = EmacPsInstancePtr->RxBdRing.PostHead + EmacPsInstancePtr->
       RxBdRing.Separation;
14 EmacPsInstancePtr->RxBdRing.PostHead = NewBD;
15 if (NewBD >(XEmacPs_Bd *) EmacPsInstancePtr->RxBdRing.HighBdAddr)
        EmacPsInstancePtr->RxBdRing.PostHead -= EmacPsInstancePtr->
       RxBdRing.Length;
```

Lines 2 and 3 are directly interacting with the Buffer Descriptor Ring structure in hardware, changing the parameters read by the DMA engine. In particular, the count of elements committed to hardware is decreased by one unit, whereas the number of post-processed elements is increased by one unit. Line 5 sets the output to be the following BD with respect to the first committed to hardware and in Line 6, it also becomes the *new* first BD committed to hardware. Line 7 eventually is just a safety check. If the last BD is going to be processed, the count should start over from the first one. The same approach is used to Free the same Buffer Descriptor, in lines 10-15. This example is provided to show how to manage one Buffer Descriptor per time, and it is implemented in the final version of the Frame Repeater, that is using 5 RX Buffer Descriptors and only 1 TX Buffer Descriptor. Whenever the processor is about to start looping back the received packets, it is actually waiting for at least one (out of five) of the *ownership* bits to become '1'. When this event occurs, the processor starts processing one by one the BDs, following the usual order: each BD is first Post-Processed and Freed, then, the TX BD is Pre-Processed, Committed, Transmitted, Post-Processed and Freed. Finally, the previously Freed RX Buffer Descriptor is again Pre-Processed and Committed again to hardware. This set of tasks is stopped only when there are no more active *ownership* bits, that is, the system is temporarily not receiving packets from the network. This new version works much better than the first one, due to the increased buffer size, that could be even further expanded. The only prob-

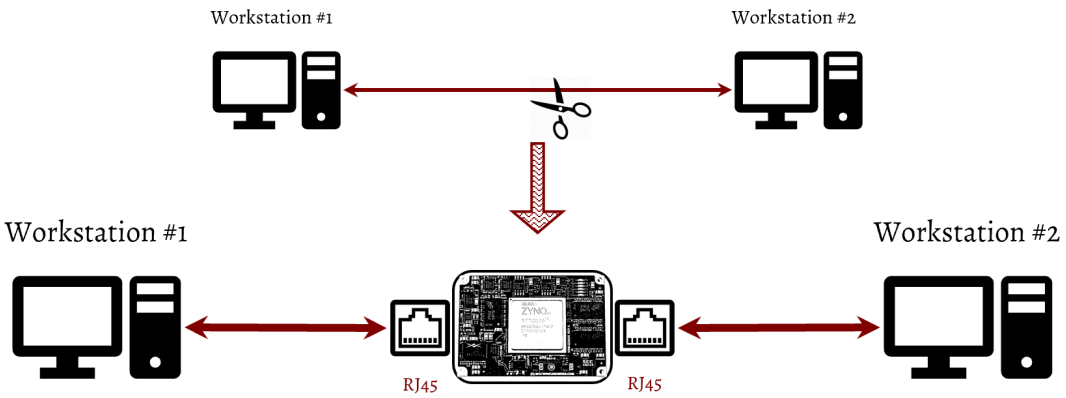

Figure 23: Practical implementation of a real time packet processing device: two separate Ethernet interfaces are the minimum requirement

lem is that for high data rate, it is still not performing well. One of the leading causes of speed/latency problems was found only in the very last days of work; therefore there was no time left to check the improvements after the patch. Such a problem was related to the clock frequency which drives the BRAM units, that was not increased to its maximum and was, therefore, bottlenecking the processor tasks.

In conclusion, the Frame Repeater is a useful way to get acquainted with network packet management in FPGA; however, it is still quite far from the primary goal, that is to monitor and process the data in real time. The reader should reasonably point out that in a normal case-scenario, two entities are communicating through a network cable. Therefore, there is no way to eavesdrop on the data employing only one port. Thanks to an additional port it would be possible to "cut" such cable, letting the stream of data enter one side of the FPGA and exit from the other one (see figure 23). A new external Ethernet peripheral is hence attached to the FPGA, but this is a different story, to be told in the next chapter.

Chapter 5

Case Study - Design Improvements

The previous chapter provided a broad introduction on the tools that can be exploited with the aim of handling data coming from the network. In this chapter is proposed a significant improvement to the previously made design, that can make the FPGA work as a junction between two distinct entities. Thanks to the additional KR-LAN-A1 module, it is possible to enable a secondary Ethernet interface and route it to the PS, through the PL. As shown in section 2.2.1, the Zynq PS has two separate Gigabit Ethernet MACs. Therefore, each of them will be in charge of handling a separate stream of data.

5.1 Physical Setup

The new Ethernet module needs to be connected to the FPGA. The only available port allowing it to reach the board is a 50-pin Zero Insertion Force (ZIF) electrical connector, that should be plugged in one of the two available sockets belonging to the KRC3701 Carrier Kit. The board is then connected to two separate Network Interface Cards belonging to the test workstation, one per each Ethernet port. Each of the four different Ethernet connectors present in this design is labeled with a different identifier so that it is easier for the reader (not only him) to understand the cabling. Everything is well depicted in figure 24. Then, the remaining physical setup is identical to the previous design, explained in section 4.1.

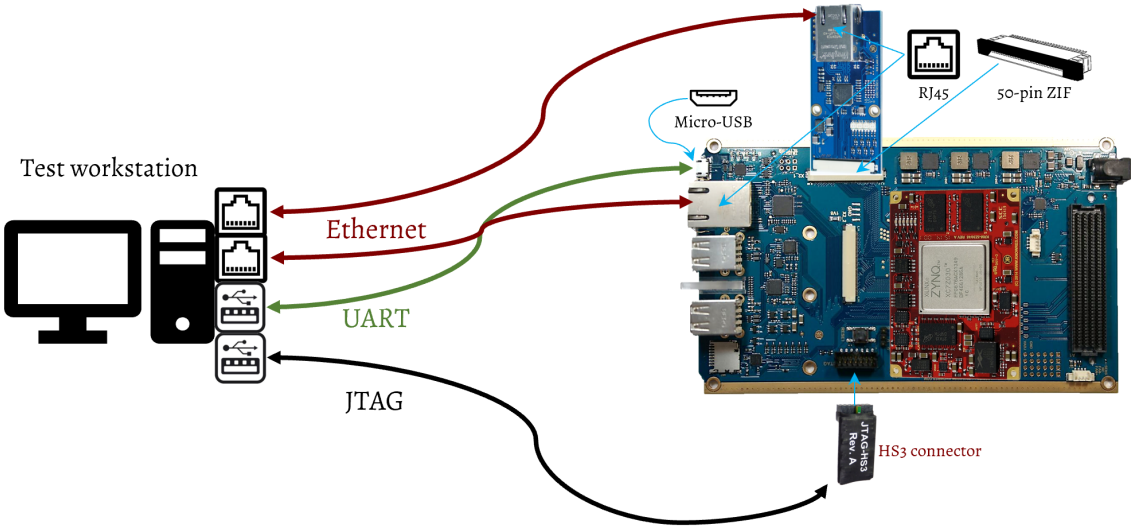

Figure 24: Physical Setup of the improved design, including the additional Ethernet module

5.2 Hardware Design

The design of the new hardware starts from the previous hardware design, discussed in section 4.3.3. The first modification consists in making the FPGA aware of the new Ethernet interface. The ZIF socket is located in the PL side; therefore, the corresponding pins of the FPGA must be enabled. The new Ethernet interface has to be routed to its Gigabit Ethernet MAC, however, being such pins already in the PL, the best option is to route them through the Extended Multiplexed I/O (EMIO). This setting must be included in the ZYNQ7 Processing System configuration in the block design. Such interface allows the PS to directly communicate with an I/O peripheral through the PL, and this is excellent news because the resources available in the logic-fabric could be easily exploited to interact with the exchanged data.

The ZIF socket of the Ethernet module is exposing mostly the PHY pins to the external world. They can be grouped in different buses, listed below:

- Management Data Input Output (MDIO)
- Reduced Gigabit Media-Independent Interface (RGMII)

The Reset signal of this peripheral must not be neglected, even though it does not belong to the previously mentioned buses. Such signal is routed through the block design because

the Zynq PS upon initialization will trigger it. Still not belonging to the previously mentioned buses, there are eight debug LEDs, four red, four green, that will be used mostly in the final version.

As regards the MDIO, it is a very simple interface made of one clock wire and one input/output bus, that is routed to the GEM through EMIO as well. This interface is used to program and interact with the PHY registers, therefore setting the speed of the link.

The RGMII interface is used by the PHY to transfer data towards the GEM, in a Double Data Rate (DDR) fashion: this means that the data are transmitted in correspondence of both rising-edge and falling-edge clock events. Such interface contains a 4-bit data bus, a clock and a control wire per each data path, both RX and TX. Unfortunately, the GEM is not exposing an RGMII interface, but instead the similar GMII. The difference between the two is mainly in the data transfer, that is Single Data Rate (SDR) in the latter case. Moreover, there are a couple of additional signals, which will be discussed later. Now, there is one more serious problem to be solved, because the two components must be connected even though they interact with different interfaces. Looking at the FPGA primitives, one can find that *IDDR* and *ODDR* should be instantiated to bring the data from a DDR source into the logic-fabric. Luckily, there is already one Intellectual Property (IP) module by Xilinx; such module is ready to be instantiated in the block design and can handle the conversion from one interface to the other, even taking care of instantiating the right Input DDR (IDDR) and Output DDR (ODDR) primitives. Such IP is referred to as *GMII to RGMII*[51] and will be explained in detail in the following chapter.

5.2.1 *GMII to RGMII* IP

This module plays a crucial role in the whole design because not only it enables the FPGA to communicate through the external Ethernet module but also it introduces some challenges to meet the timing constraints in the whole design. The interfaces with the external world are depicted in figure 25. Starting from the MDIO, there are two different buses: one (slave) is supposed to be connected to the Zynq PS, the other (master) to the external module's MDIO. The reason why the *GMII to RGMII* is in the middle of this bus, from

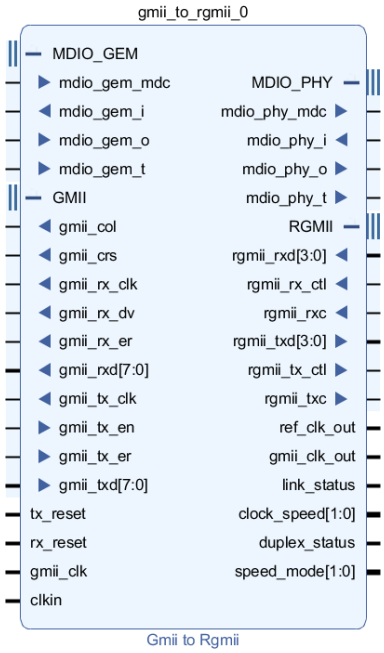

Figure 25: *GMII to RGMII* IP module, inputs and outputs

the Zynq to the external world, is because of the better flexibility with PHY management. Said with different words, whenever there are multiple PHYs in the same design, the PS must know how to communicate with each of them correctly. The PHY has a configurable address; usually, a 5-bit address is hardcoded in the physical module through some pull-up or pull-down resistors, wired in a specific fashion. This means that, depending on the manufacturer, there can be modules mounting a PHY that is already configured on a different address. Such an address can be edited by soldering/unsoldering the corresponding electronic components; nevertheless, it is not usually the most convenient solution. That is why the *GMII to RGMII* has a configurable MDIO address, that is nothing but a virtual PHY, masking the real address of the external PHY, allowing the PS to reach the external module easily. Another reason why the *GMII to RGMII* stays in the middle of this bus is to provide the correct link speed setting, but it will be discussed later.

As regards the GMII interface, there are a couple of signals differing from the corresponding RGMII. Indeed, the data bus is doubled in size, from 4 to 8 bits, due to the conversion from DDR to SDR. Then, there are two couples of status signals that are output

of the GMII interface: *valid*[1] and *error*[2]. Such signals are informing the GEM whether it should accept or refuse the data depending on the network being currently idle, or transmission errors occurrence.

For what concerns the clock domains, the GMII to RGMII IP needs in principle just one clock signal of 200 MHz, referred to as *clkin*. From such clock, the module is generating the other necessary clocks to control the TX datapath, responsible for transmitting the network packets at the same speed of the line: according to the given specifications[51] 125 MHz are needed by a 1 Gbps link, 25 MHz are needed by a 100 Mbps link, 2.5 MHz are needed by a 10 Mbps link. Optionally, an external clock can be provided to drive such TX data path.

Among the outputs, there are three very useful for debugging but also for future development. They are the *link_status*, *clock_speed* and *speed_mode*. The first one is quite intuitive and indicates whether the PHY is ready to exchange data over the Ethernet cable with another entity. The reader should remember that before the Auto-negotiation sequence, even though a cable is physically connecting one entity to another, it is in general not guaranteed that a data transfer is allowed because the two parties have to agree on the link speed and duplex mode[3]. As regards the other two signals, although their names can be easily confused, they indicate the same quantity, that is the speed of the link. However, the *clock_speed* represents the clock speed decoded from the RGMII interface (and thus from the PHY), whereas the *speed_mode* indicates the clock speed that the GMII to RGMII IP should choose for transmitting data. In order to operate properly, this module needs to know the link speed that was set up by the PHY during the auto-negotiation sequence. Such a piece of information is communicated through the MDIO interface. As said before, the GMII to RGMII IP is configured to be a "virtual" PHY, programmable through the MDIO. Therefore, after the auto-negotiation is done, it is sufficient to read the link speed from the actual PHY and write this information to the corresponding register (0x10) in the "virtual" PHY. The *speed_mode* output will show the value written in such

[1] `gmii_rx_dv` and `gmii_tx_en` for respectively RX and TX datapath
[2] `gmii_rx_er` and `gmii_tx_er` for respectively RX and TX datapath
[3] As explained in the previous section 3.3

register; therefore, for correct operation of the interface, such output must be equal to the *clock_speed* one.

5.2.2 Additional BRAM Cells

The title of this paragraph is suggesting to the reader that the previously placed BRAM modules (two) are not anymore sufficient. The reason is not straightforward. In the previous section 4.3.3, it was explained why two separate BRAM modules were needed, that is to place the Buffer Descriptors in one and the RX/TX Buffer in the other. The application described in the current chapter is showing a setup where two distinct Ethernet connections are wired. From a different point of view, one can point out that it is just one Ethernet connection, since the FPGA is in the middle, forwarding the packets from one entity to the other. There is nothing wrong in this conclusion; nevertheless, the reader has to make a further step, already looking at the software implementation and in particular at the sequence of tasks performed whenever a packet transits through the FPGA. This time there are two different GEMs (corresponding to the two different interfaces) and each of them must be assigned a Buffer Descriptor List and a Packet Buffer (both RX and TX). The two GEMs can not share these two structures for apparent reasons. The lack of a suitable protocol prevents them from exchanging information on the number of packets processed in real time. Therefore, there is no way for one of the two GEMs to control in an ordinate way the states of the Buffer Descriptors or the Queue Pointer Register when the other GEM is operating on them.

That being said, one can think to increase the memory size of the available BRAM and place the two additional structures in a contiguous space, so that they can work separately without interfering with one each other. This solution was considered to be implemented, but in the end, it was discarded. Consider a scenario in which two packets are processed at the same time by the two different GEMs. Say, they are trying to write at the same time on the same memory block two different pieces of information (suppose they are both setting the *ownership* bit after a successful packet reception). This is not a problem since the BRAM is connected to its controller through a dual port interface, which enables

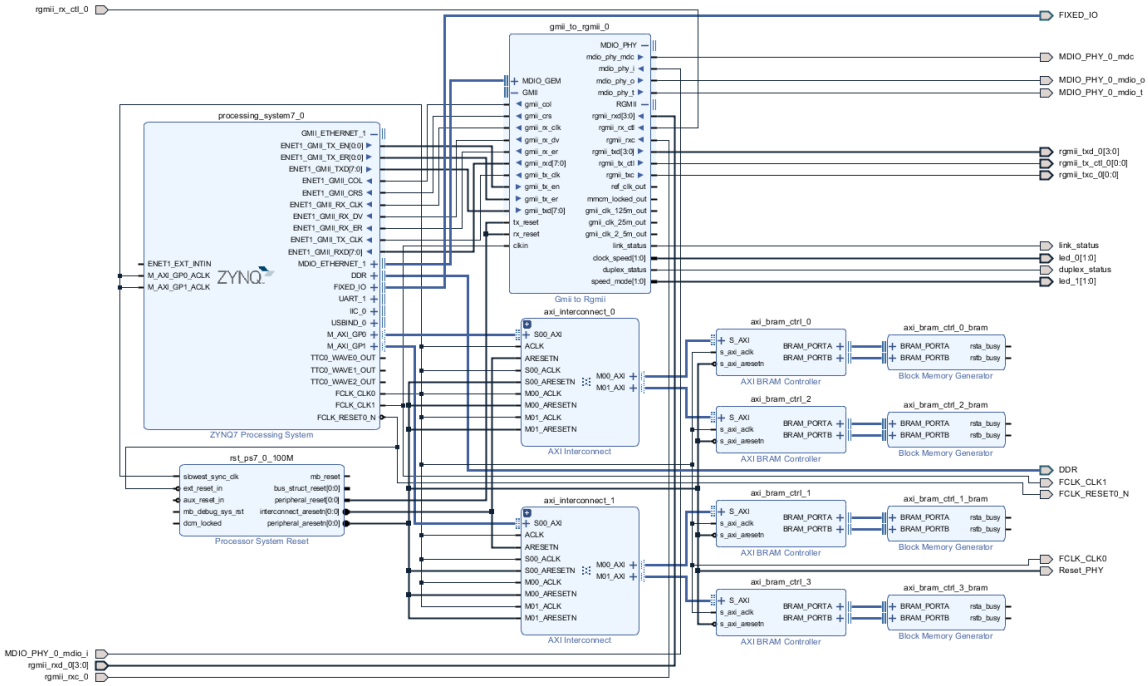

Figure 26: Block design of the improved Frame Repeater

two distinct operations to be executed at the same time on different memory addresses. However, what happens, if, at the same time, a new packet is received by one of the two interfaces? The DMA will move it to the corresponding memory location, belonging to the other BRAM module; however, the related Buffer Descriptor cannot be immediately updated since the two ports of its memory block are already busy. To avoid the occurrence of any conflict in memory, each of the GEMs was assigned a distinct pair of BRAM modules, hosting the Buffer Descriptor List and the RX/TX buffer respectively. Therefore, in the final design shown in figure 26 there are two pairs of BRAM cells, each connected to its Controller, multiplexed by a pair of *AXI_Interconnect*, which are connecting each couple to the PS, through a different General Purpose AXI Interface.

5.3 Constraining the Design

Whenever a hardware design is made of different components with a high number of interconnections and, possibly, different clock domains, the compiler will struggle with timing constraints. Such constraints are a set of rules defining the limits for propagating a signal in a wire. In a sequential design, for example, the ruling component is the

Flip Flop. There are mainly two constraints that must be met for guaranteeing the right functioning of a flip-flop: the *setup time* and the *hold time*. The *setup time* is defined as the minimum amount of time during which the data must be stable *before* the clock active edge. The *hold time* has the same definition, but with the word *after* instead of *before* [52]. This means that in a synchronous design, all the signals must end their journey through the components and reach the next flip-flop input before the setup time. The amount of time needed by a signal to travel across components and wires depends on the wire length but also on the "internal" propagation time, think for example to the time required by a carry signal to travel across a full adder. The compiler task consists of finding the optimal displacement for all the components in the design.

In this specific case, the design still lacks additional modules written by the developer, and only connects pre-defined Xilinx IPs (like the *GMII to RGMII* and BRAM Controllers). Since they are reviewed and maintained by a community of skilled experts, there are in general no problems related to the propagation of signals *inside* these modules. The problem is always at the boundaries, and, in this specific case, the reader should remember that the *GMII to RGMII* has to convert a DDR signal into an SDR signal.

In order to help the board sampling the data coming from the RGMII interface, the PHY has the useful feature of delaying by 2 ns the data with respect to the clock. This will be explained once again in the software section when the code to be executed by the CPU will be checked out. As regards the interface between the FPGA and the external PHY, the *GMII to RGMII* is instantiating IDDR primitives in order to sample the incoming data. Such IDDR primitive can be seen as a couple of Flip Flops, one of which clocked on the rising edge of the clock, and the other on the falling edge. Such clock is nothing but the incoming RGMII RX clock. The right set of timing constraints to be applied for this scenario (i.e., sampling a center-aligned DDR input) is already included in the *GMII to RGMII*, however, it is not automatically recognized by the compiler during the implementation. The reason is a mismatch in the clock names into play: the actual incoming clock, and the virtual clock used as a reference to set the timing constraints. To solve this problem, a virtual clock bounded to the *rgmii_clk* must be declared in a constraint file. Then, such a file must be

loaded during the implementation step, before the *GMII to RGMII* constraint files.

5.4 Software Design

The software that makes this application run is very similar to the one designed for the previous application. The more evident difference consists in the initialization and configuration of the second GEM, whose *BaseAddress* is 0xE000C000. The set of operations performed is analogous to the ones described in section 4.3.4 and the new BRAM modules are mapped to addresses 0x4200000 and 0x82000000. The reader should now pay attention to the flow of data to understand how to write the rest of the code. From figure 24 one can deduce the existence of four separate streams: from A to B and from B to A, from C to D and from D to C. However, since the purpose of this application is to let the data flow through the fabric, the whole architecture will show just two distinct streams: from A to D and from D to A. To achieve this result, the software must guarantee continuity between the B and C interface. In particular, whenever the B port receives a packet, it must be pushed through the C port, and vice versa. Recalling the previous application, described in section 4.3.4, the reader should remember the moment when the software is ready and waits for a packet. In this case, the software still waits for a packet but has to take into account also the secondary port. Hence, a function was designed for polling the Buffer Descriptors and understanding who is *owned* first.

```
1 int IsRxHalf(XEmacPs *Port1, XEmacPs *Port2, int start1, int start2){
2
3 XEmacPs_Bd * Bd1 = (XEmacPs_Bd *) XEMACPS_INDEX_TO_BD(&(
    XEmacPs_GetRxRing(Port1)), start1);
4 XEmacPs_Bd * Bd2 = (XEmacPs_Bd *) XEMACPS_INDEX_TO_BD(&(
    XEmacPs_GetRxRing(Port2)), start2);
5 if ((XEmacPs_BdRead(Bd1, XEMACPS_BD_ADDR_OFFSET) &
    XEMACPS_RXBUF_NEW_MASK) == 1) return 1;
6 else if ((XEmacPs_BdRead(Bd2, XEMACPS_BD_ADDR_OFFSET) &
    XEMACPS_RXBUF_NEW_MASK) == 1) return 2;
7 else return 0;
8 }
```

The first two arguments are identifying the different Ethernet ports (say, Port1 is corresponding to port B, and Port2 is corresponding to port C). The remaining arguments instead are a kind of index used by software to follow the first available Buffer Descrip-

tor per each RX Buffer Descriptor list. Whenever the CPU is waiting for a packet to be received, it will poll the `IsRxHalf()` function. Such function will return 0 if there is no evidence of a received packet; otherwise, it will return the number of the corresponding port (1 for port B, 2 for port C). The macro `XEMACPS_INDEX_TO_BD` is defined to easily convert the previously defined index into a regular address in memory. Such address belongs to the Buffer Descriptor with the corresponding position in the list.

```
#define XEMACPS_INDEX_TO_BD(ringptr, Index)
((UINTPTR)(ringptr)->BaseBdAddr + ((u32) Index%RXBD_CNT * (u32) ((
    ringptr)->Separation)))
```

This piece of information is sufficient to setup correctly all the following functions, from Post-Processing the last *owned* Buffer Descriptor to Committing it again to hardware. The whole routine is placed into another function:

```
int SendBack(XEmacPs * RxPort, XEmacPs * TxPort, XEmacPs_Bd *
    BdToBeSentBack, int Direction)
```

where `BdToBeSentBack` is the previously identified RX Buffer Descriptor and instead `Direction` indicates whether the stream of data flows from B to C or from C to B.

It is important to mention that immediately before running the `SendBack` function, the index corresponding to the receiving port (either `start1` or `start2`) is incremented by one unit or zeroed if the last element was reached; therefore, the `IsRxHalf()` function will be constantly following the first available Buffer Descriptor.

5.5 Results and Conclusions

The improved design of the Frame Repeater is working, although there are two major issues. The first one is the same as the previous application: the BRAM clock frequency was not raised up to its maximum; therefore, the CPU is slowed down by all the operations that are involving memory access. In case of high traffic rate, the FPGA is simply losing some of the incoming packets when it is busy in rerouting them to the other interface.

On the other hand, there is another problem, involving the two separate Ethernet streams. If the FPGA is busy, calling the DMA functions to move the packets from one port to the other, say from B to C, it will be impossible for the processor to bring a packet

from C to B at the same time. The DMA, of course, will do its job, placing the packet in its relative Buffer Descriptor, however, sooner or later they have to be emptied. In the worst case scenario, in which both the two ports are overloaded with network packets, the processor will struggle to empty both the two Buffers and thus will end up losing some packets. One way to solve this issue is to build two distinct code scripts, to be loaded in the two cores of the CPU. As said in section 2.2, the Zynq7030 has a dual-core Cortex A9 processor. This means that it can run in parallel two separate sets of instructions. However, programming the processor in this way is not straightforward, and this design was eventually abandoned, in favor of the last and final version of this application.

5.6 Towards the Final Design

Before moving on to the final design, it is useful to explain the main reasons why the previous attempts failed. As said multiple times, the primary goal of this application is to provide a real-time packet processing device. This means that the application must be surely able to forward the packets, from one port to the other, but in the end, it must also be able to provide some modifications to the data. Since it is quite evident that memory access is limiting the processor, there should be another way to edit the packets on the fly. The most potent feature of FPGA fabric is that with the right logic description, and the right pipelining, it is possible to drive the data through a series of registers before pushing them to the next interface. The main advantage consists in the possibility of modifying the data while they are moving from one register to the following; hence there will be no operational delays (the registers govern time), provided all the timing constraints be met. A good approach to implement this solution consists in intercepting the data coming from the PHY before they reach the GEM. Unfortunately, there is no way to intercept from the PL side the data exchanged between the carrier evaluation board Ethernet port and the MAC controller: the RJ45 connector is in facts directly linked to the MIO pins. Therefore, since such MIO is a big multiplexer whose inputs are inside the PS, there is no way to read the data prior with respect to the MAC controller. The second design opens a beautiful door to the PL, thanks to the secondary Ethernet port. This could be exploited, by

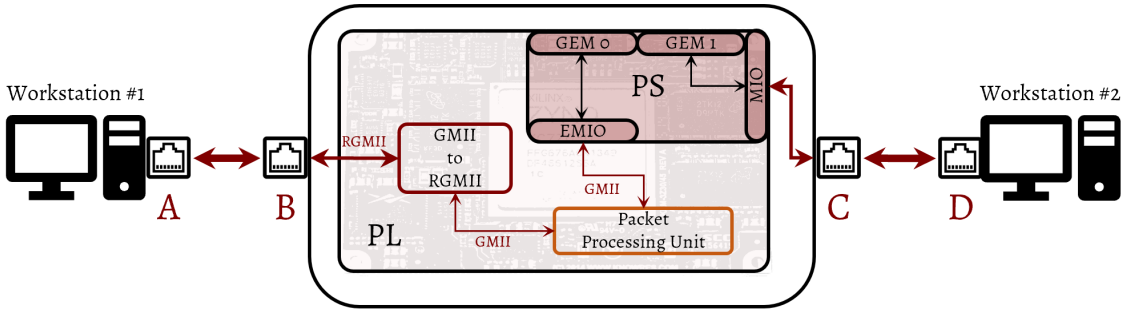

Figure 27: Concept design of a packet processing unit in between the external Ethernet module and the rest of the board

inserting a module between the PHY output and the EMIO interface, as depicted in figure 27. However, there is still no way to intercept data from the other Ethernet port because it is still routed through MIO pins. One can say that there is no reason to complain: in order to monitor the data flowing from B to C a packet processing unit could be placed on the RX line of the external PHY, whereas to monitor the opposite flow (from C to B) it would be sufficient to put another packet processing unit on the TX line. This is true; however, there is a non-negligible difference between the two sets of data. Such difference consists in the GEM and its role in the packets management: the data on the RX data path is supposed to flow into the MAC controller of the other interface. Therefore, if they have a wrong checksum, or, in general, if they do not comply with the MAC preferences, they will be discarded. This is not holding for the TX data that instead already transited through the MAC controller; however, if they are modified before leaving the board, also the final checksum must be recomputed. Although it is possible to develop a more complicated solution, differentiating between the two paths, the application working principle will still be bounded to memory management, DMA and CPU.

Therefore, as soon as the company provided an additional Ethernet module, the final design was started: the MAC controller is not anymore a relevant element in the design since it can be bypassed, by working only with data at Layer 1 in the Ethernet reference stack [4].

[4] See section 3.3

Chapter 6

Case Study - Proof of Concept

The final design here described manages to implement the functions described in the introduction successfully and to achieve all the goals. Indeed, there would be still much work to be done for improving the design, that is why the title of the chapter is "Proof of Concept" rather than "Final Design". Unfortunately, time runs fast, and it is not infinite.

The structure of this chapter will follow a more articulate structure with respect to the previous ones. Here below is proposed the outline:

- Introduction - A new approach
- Physical Setup
- Hardware Design - Preliminary Block Design
- Hardware Design - Packet Processing Unit
- Software Design

6.1 Introduction - A New Approach

The addition of a secondary Ethernet module, yet another *KR-LAN-A1*, makes the whole design change a lot. First of all, the reader should recall that the data coming out from the PHY of such module is pushed to the FPGA through an RGMII interface. Since the primary goal of all the analyzed designs up to now is to drive the received data to another port, there are two possibilities: enter the PS and exit from the MIO or stay in the PL and leave from the other Ethernet module. Indeed, the former option was already faced in the

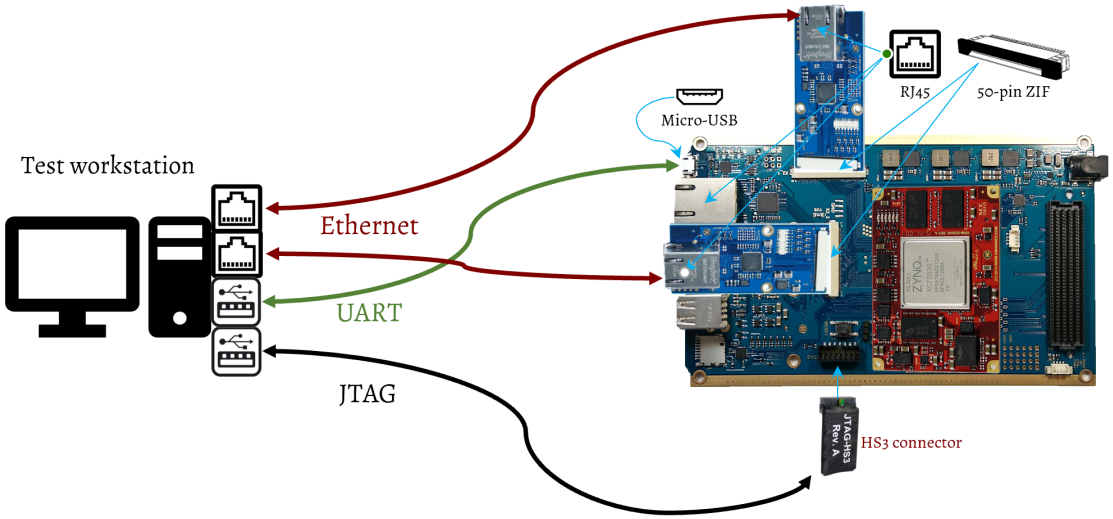

Figure 28: Physical Setup of the final design

previous section 5; therefore it will follow an in-depth analysis of the latter. Since almost everything is routed through the PL, the PS will play a very marginal role, because it will be only necessary to configure the network elements such as the external PHYs and the other building blocks.

6.2 Physical Setup

Again, a new Ethernet module has to be connected to the FPGA. Luckily, the KRC3701 Carrier Kit has two available sockets for a 50-pin ZIF connector; therefore, there is enough room for both the two external Ethernet modules. These modules are connected respectively to the two Network Interface Cards of the test workstation. This setup, shown in figure 28, is almost identical with respect to the one described in section 5.1, differing only by one port, that is not anymore the one embedded in the Carrier Kit, but is the one coming along with the external Ethernet module.

6.3 Hardware Design - Preliminary Block Design

At first glance it seems that is possible to directly connect the pins of the two external modules, to route all the packets from one interface to the other. Unfortunately, this is not the case: the RGMII interface is carrying a DDR signal, therefore, before entering the

FPGA it should go through IDDR primitives. This problem, however, does not sound new: as seen in the previous design (see Chapter 5.2), the *GMII to RGMII* IP is instantiating the necessary IDDR and ODDR to correctly sample the incoming signal and eventually transform it into an SDR signal. Therefore, this time, two different modules will be needed, one per each external PHY. At this point, the reader might wonder how to link the two GMII interfaces effectively. The most straightforward solution seems to wire the RX signals of the first interface to the TX of the other. Unfortunately, this will lead to a colossal mistake. The two channels (RX and TX) are synchronous to two separate clocks: one is generated by the external PHY (RX clock), whereas the *GMII to RGMII* generates the other. Whenever the signals coming from two different clock domains are crossing, it is not possible to mix them, provided some counter measurements be taken to preserve the signal integrity and meet the timing constraints. The most common solution to make signal travel from one clock domain to another is to let it flow through a FIFO module, with independent read and write clocks. Hence, two FIFOs are instantiated: the former, connecting the RX data path of one interface to the TX data path of the other, the latter, vice versa.

6.3.1 First In First Out (FIFO)

A FIFO (First In First Out) is a hardware module able to buffer an input signal; it is defined by a *width* and a *depth*. The first term indicates the size of the incoming bus that is going to be buffered; on the other hand, the *depth* indicates the maximum number of samples the FIFO can hold while running. Basically, the FIFO is sampling the input signal, having a certain *width*, with the *write* clock. The samples are stored in a BRAM cell. Subsequently, those samples are pushed out of the BRAM with the *read* clock, following the same order of arrival. That is why it is referred to as First In First Out. As said before, this component can let a signal propagate through two different clock domains, which are used respectively to write and to read the FIFO. There are some additional signals that are useful when dealing with FIFOs and they are also summarized in figure 29: *write_enable*, *read_enable*, *programmable_full*, *programmable_empty*, *valid*. The first couple of signals is trivial to be understood: they enable the FIFO to be respectively filled or emptied. The

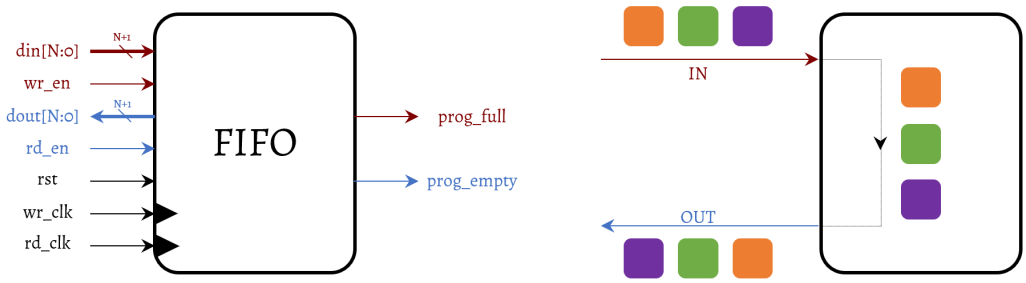

Figure 29: FIFO main Inputs/Outputs. A visual diagram is proposed to understand how data are buffered.

second couple instead is a flag that is raised whenever the number of samples in the FIFO is respectively greater or smaller than a fixed threshold. Such a threshold can be defined by the developer when configuring the module. The last signal is a simple flag that is raised whenever the FIFO is pushing real data out. Such a flag is de-asserted when, for example, the FIFO is empty, or is not read-enabled. As a last remark, the signals belonging to the read datapath (*rd_en*, *dout*, *prog_empty*, *valid*) are all synchronous with respect to the read clock (*rd_clk*). Analogously, the write datapath is synchronous with respect to its clock (*wr_clk*).

Now, the right way to configure the FIFOs must be figured out. One can think to use the *gmii_rx_dv* signal (indicating a valid data received) to drive the *write_enable* input of the FIFO and then use the *valid* output to drive the *gmii_tx_en* signal (indicating a valid data to be transmitted). This implementation suffers from a big problem, caused by the imperfections of the non-theoretical world. Said differently, the FIFO is supposed to be clocked by two different sources that theoretically are running at the same frequency. Indeed, the two interfaces must share the same line speed; otherwise, it would not be possible to ensure the continuity of data flowing from one port to the other. Unfortunately, in the real world, if the source is not the same, there will always be a (hopefully) tiny difference between the clock signals. As regards the FIFO, if the traffic rate is constant, sooner or later, it will be possibly full or empty, depending on which clock runs faster. The consequences of this events are definitely unwanted: if the FIFO is full, one byte of data will be lost per each *write* clock cycle. If the FIFO is empty, there will be at least one *read* clock

cycle with a null output. If this happens in the middle of packet transmission, the "valid" signal will be forced to '0', thus making the transmission end with a malformed packet. As a matter of fact, the *gmii_rx_dv* signal and also the *gmii_tx_en* signal are supposed to be de-asserted only at the end of a packet. In order to solve this issue, the FIFO is designed to be an elastic buffer, taking as inputs all the signals coming from the GMII interface, so, namely, *gmii_rx_dv*, *gmii_rx_er* and *gmii_rxd*. Therefore, the *width* size is set to 10 bits (8 for the data path, and 2 for *valid* and *error* signals). An elastic buffer has an additional set of "rules" with respect to a traditional buffer, allowing it to discard some data at the *write* interface when it is almost full, or stop the *read* interface if it is almost empty. As this solution is quite common in network applications, according to IEEE 802.3 Standard, there is supposed to be a temporal gap between the transmission of two consecutive packets [53]. Such a gap, referred to as Inter-Packet Gap (IPG) corresponds to the time required to transmit 96 bits. In a 1 Gbps link, since the clock has a period of 8 ns (125 MHz), it will correspond to 12 clock cycles. Moreover, the IPG reference value is defined with a tolerance, allowing a minimum of 64 bits time in case of synchronization problems. Considering the previously described scenario, although the *read* clock is not well aligned with the *write* clock the elastic FIFO will be complying with the specifications given by the IEEE. Even though the IPG is reduced by a couple of octets, it will not be a problem for the whole transmission.

So, to configure correctly such elastic buffer it is necessary to use the previously mentioned *gmii_rx_dv* and also the *gmii_tx_en* signal. These will determine whether the FIFO can discard data or halt their transmission. In order to do that, there are other two parameters to be configured, i.e., the two thresholds, warning about the filling level of the FIFO. Usually, such parameters are set around the middle of the FIFO depth, as shown in table 6.1. Now, these signals must be correctly combined before being connected to the enable inputs. In particular, the FIFO should be written when it is not too full, or when the incoming data are valid (*gmii_rx_dv* is asserted). On the other hand, it should be read when it is not too empty, or the outgoing data are valid (i.e., the FIFO output signal corresponding to the *gmii_rx_dv* line, in this case, linked to *gmii_tx_en*, is asserted). Summarizing, as

regards this preliminary configuration, the logic relationships between signals are:

```
wr_en <= gmii_rx_dv OR NOT prog_full;
rd_en <= gmii_tx_en OR NOT prog_empty;
```

That being said, this preliminary block design looks like figure 30. The reader can see that the reset signal of the FIFOs is linked to the *link_status* output of the *GMII to RGMIIs*. Whenever one of the two links is down, there is no reason to keep the data in; therefore each FIFO is reset. Only a few more steps are needed now to make this design work, and they will require a few lines of code to be executed by the CPU, to configure the interfaces correctly. This is the disruptive feature of the proposed design, as the PS and in general, the processor is almost entirely cut out of the game. Finally, it is possible to start wondering how to process the data in real time. The best idea is to break the connection between the FIFO and the *GMII to RGMII* modules because the data is Single Data Rate (SDR) in that region. However, there are two different configurations, depending on the position of the cut, as explained in figure 31.

The choice of the preferred configuration was carried out only during the design of such Packet Processing Unit (PPU) because it is strictly correlated with the risk of crossing clock domains and of missing the timing constraints. Hence, the chosen configuration sees the PPUs right after the read side of the FIFOs. The idea is to let all the buffered data flow inside the Packet Processing Unit, including the *valid* and *error* signals. This comes with a small modification on the read enable of the FIFO, that is not anymore generated by the *gmii_tx_en* signal. In the same logic expression, it is replaced by the newly labeled *DV_IN* belonging to the PPU, indicating the corresponding data valid signal indeed.

In the following sections, the Packet Processing Unit will be explained and, later, the hardware design will be completed.

FIFO configuration	
width	10
depth	2048
empty_threshold	800
full_threshold	1200
36 Kb BRAM resources required	1

Table 6.1: Configuration of the FIFO cells

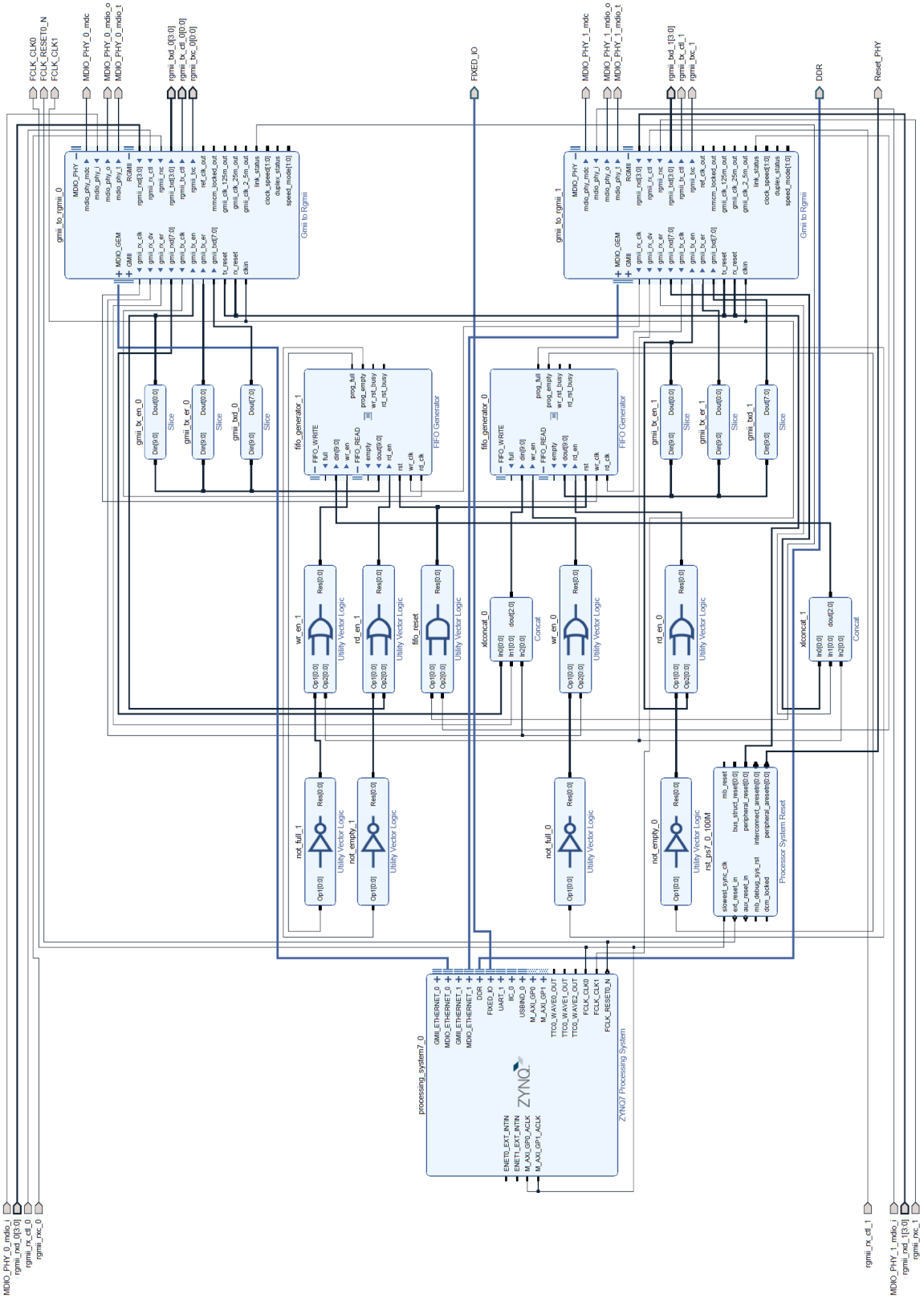

Figure 30: Preliminary block design of a FPGA with two “short-circuited” Ethernet interfaces

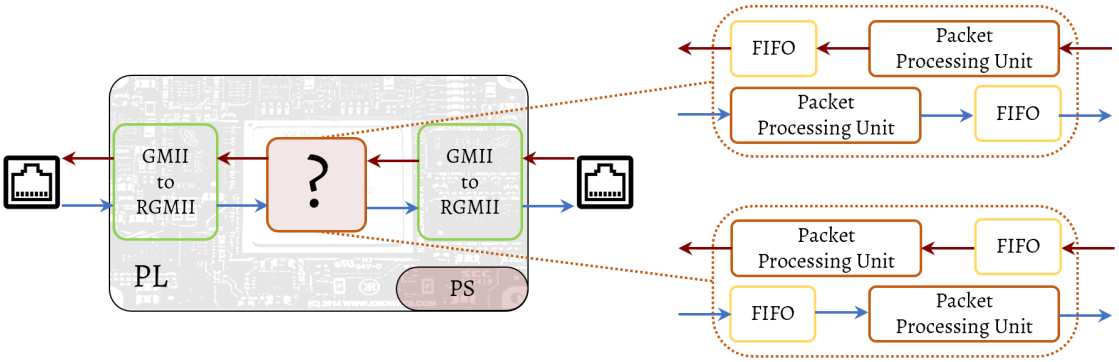

Figure 31: Two different configurations are available to install a packet processing unit, respectively on the read side and on the write side of each FIFO

6.4 Packet Processing Unit

This is probably the most critical section of this thesis since it contains the core of this hardware application, that was build from scratches. The reader will be proposed the same iter of development the application was given: feature after feature, it will be brighter and plainer why it is referred to as a toolbox. As said at the end of section 5, the idea is to build in the FPGA fabric a data path, well pipelined, made of several registers. Such datapath must be able to bring the data from one side of the packet processing unit to the other and apply possibly some modifications. This can be represented as a supply chain, where there are many workers, standing in a fixed position, operating on the object that is carried by the running belt. Such an object is, in this case, one byte (8 bits) of a network packet, recalling that the GMII data interface is an 8-bit wide bus. If all the workers can do their jobs before the belt moves, the whole system runs smoothly. Moreover, the whole stream of data preserves the same spacing in time between one packet and the following; the only measurable difference is the time of travel through the PPU. In this specific applications, it is roughly 200 ns, that is nothing if compared to the average latency in a wired gigabit communication (a hundred times greater). Almost all the implemented features of the Packet Processing Unit must be configured via software. As regards this section, only a hardware description will be provided; then, later, the reader will go through each software configuration, in the same order of appearance. The starting point for the hardware design is the implementation of the previously mentioned

"supply chain". This is nothing but a chain of twenty-four 10-bit serial registers, with synchronous resets, whose first input is wired to the read output of the FIFO, whereas the last output is connected to the TX interface of the *GMII to RGMII*. The number of 24 was chosen as a consequence of the ICMP killer function (explained in section 6.4.5). Indeed, one can work on the Packet Processing Unit to reduce the number of registers; however, this topic will be thoroughly examined in the Upgrades Section.

6.4.1 Clock Setup

The clock setup is probably one of the most sensitive parts of the design. Being the Packet Processing Unit placed right after the FIFO, the only way to guarantee the integrity of the stream of data is to use the same clock as the read-side FIFO. Such clock will also be fed to the TX side of *GMII to RGMII*. However, due to the implementation of filtering features in the Packet Processing Unit, this choice has to be slightly changed. In the following sections, the working principles of such filters will be explained in detail. For the moment, the reader has to believe that there is the need for a clock running two times faster than the main clock. Moreover, they must also be in phase: once every two events, they must transition from low to high together. There are a couple of ways to figure out this issue, but the bottom line is the same: the two clocks must have the same source. In order to generate a clock signal inside an FPGA, there are two different sources: Phase-Locked Loop (PLL) and Mixed-Mode Clock Manager (MMCM). While the first one is quite common in the world of electronics as a frequency synthesizer, the second is mostly used in the world of embedded systems. An MMCM can be seen as an upgraded version of a PLL, with additional features, among which phase control [54]. As reported in [55], this component is used to generate multiple clocks with defined phase and frequency relationships to a given input clock. As regards this specific application, the *GMII to RGMII* generates the clock that controls its TX side and feeds the read side of the FIFO. The starting point is the 200 MHz source, provided by the Zynq7 Processing System IP. Unfortunately, it is not possible to instantiate a component inside a pre-packaged IP; therefore, since the TX data path of the *GMII to RGMII* is synchronous to its internally generated TX clock, there

is no way to enforce a relationship with another clock. Luckily there is a workaround, that allows feeding an external clock to the *GMII to RGMII*, through the *gmii_clk* input. The new configuration implies that now the TX data path will be synchronous to the external clock, but there will be no more options to choose the speed of the link, being such input clock fixed. Enabling the external clock disables the clock generator that was providing the three different options for the TX data path. Although this little disadvantage does not allow much flexibility, the external clock is considered a weightier advantage. Therefore, starting from the 200 MHz clock, required anyways by the *GMII to RGMII* to configure the IDELAYCTRL, two additional clocks are generated by an MMCM: a 125 MHz one, to drive the Packet Processing Unit and the TX data path of the GMII interface, including the read side of the FIFO and then a 250 MHz one, for other purposes, explained later. These clocks are ensured to be in phase since they come from the same MMCM.

6.4.2 Preamble Detector

The most straightforward task that the Packet Processing Unit can perform consists in detecting whether a network packet is about to be streamed through its registers. Recalling section 3.4, an Ethernet packet always begins with a preamble, that is a sequence of 8 bytes all equal to 0x55, except the last one, equal to 0xD5. A simple Finite State Machine is sufficient to build a Preamble Detector, just by checking the value of the very first register and keeping track of the number of consecutive positive events. If the right sequence of 7 times 0x55 followed by a 0xD5 is detected, the PPU asserts an output signal for, roughly, 100 ms, called *PACKET_DETECTED*. Such output is driving a green LED on the Ethernet module that accepted the packet. Although it seems to be quite a simple and useless feature, it is instead fundamental, because it determines the starting point for all the following data processing functions. Indeed, one can manipulate a network packet, only if there is a clear sign of its inception.

6.4.3 MAC filter

An Ethernet frame contains the MAC address of the destination, followed by the MAC address of the sender (source), right after the preamble. It would be useful to create a blacklist of MAC addresses, both for source and destination. This feature allows the board to discriminate the network packets if their address matches the corresponding list (destination or source). Hence, a packet might be dropped on purpose, i.e., lost in wires, and thus prevented from flowing through the board.

The operation of filtering according to a given list is performed by first storing such list in a memory, in this case, a BRAM cell. Then, each time a new candidate MAC address is under inspection, the blacklist is browsed. If a match is found, then a flag is raised, and the registers of the Packet Processing Unit are cleared: the packet is immediately dropped, and the GMII interface is even prevented from receiving valid data. However, since the list might contain several entries, it must be browsed multiple times to check them all. The operation of finding a match is a sensitive point: practically speaking, the candidate MAC is compared with one of the stored elements, meaning that at least an equality check is performed. Therefore, one can deduce that a good filtering algorithm is equivalent to the implementation of a search algorithm. The time required by such an algorithm to run is proportional to the size of the list. As the "supply chain" of registers is not infinitely long, such algorithm should be optimized to run quickly; therefore the *binary search algorithm* seemed to be a good source of inspiration [56].

Whenever a piece of code is written for a hardware platform, the developer should always be aware of its primitive cells and functions. This is the reason why the proposed solution is quite elegant. Thinking about the search algorithm, as said above, a comparison operation is supposed to be carried out multiple times. In hardware description languages, such operation is not straightforward: it might be translated into a logical expression, or it might require to instantiate several multiplexers. The bottom line is that the synthesizer has to correctly evaluate the comparison operation required and guess how to implement it in the design correctly. This is not an elegant way of writing the code

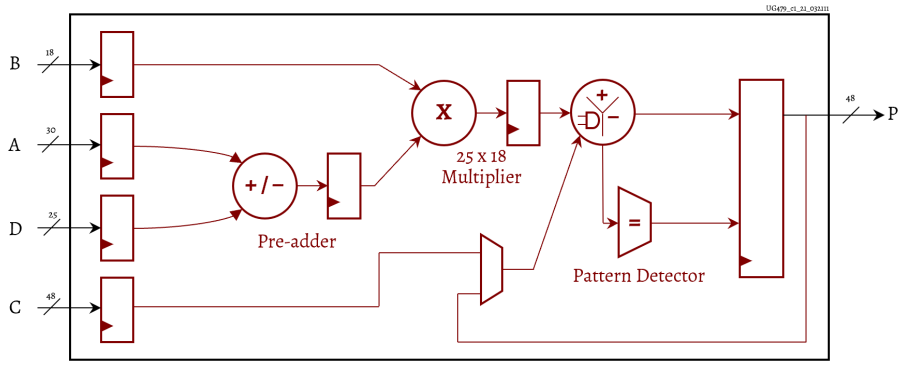

Figure 32: Basic functionality of DSP48E1 slice, as reported on the manufacturer datasheet [12]

since the outcome is not known a priori. Moreover, depending on the complexity of the implemented logic, there might be problems in signal propagation causing the timing constraints not to be met, due to the high number of consecutive logic levels. A much better approach, way more elegant indeed, is to instantiate a *DSP48E1* slice manually. Such slice is a 7-series FPGA primitive, designed explicitly for Digital Signal Processing (DSP). The basic functionality of the DSP48E1 slice is shown in figure 32, taken directly from the datasheet[12], and for what concerns the filtering task, the DSP is mainly used to make arithmetic operations. Trivially, a comparison operation (like *greater than* or *smaller than*) can be translated into a subtraction, followed by a sign check:

$$a > b \Leftrightarrow (a - b) > 0 \tag{6.1}$$

A sign check is an effortless operation for binary data since it just requires the Most Significant Bit (also called *sign bit*). If this bit is equal to 1 the number is surely negative, if equal to 0, it is surely positive (or null). Of course, the whole analysis works only if the variables are of the *signed* type. That being said, in order to configure the DSP to evaluate algebraic comparisons, it is first necessary to understand how to fetch the terms of comparison from memory. The reader must figure out the following steps:

- Setup the *DSP48E1* to make subtractions
- Configure BRAM and read it from the PL
- Translate the search algorithm into a finite state machine

Before continuing with the detailed description of these bullet points, a small spoiler is given. To increase the number of maximum elements contained in the blacklist, it was decided to store in memory only a portion of the MAC address to be filtered. Recalling from section 3.4, the MAC address is a 48-bit number, whose former 24 bits are referred to as Organizationally Unique Identifier (OUI). The implemented MAC filter feature here described should be better referred to as *OUI filter* since the information stored in memory that will be compared with the incoming data is just the first 24 bits of the MAC address. Although it seems to be a restriction, the act of preventing a specific set of devices, belonging to the same manufacturer, from interacting with the network is a good security application, used to cut off untrusted devices. In the future, one can surely improve the design, allowing the user to input a regular MAC blacklist.

1. *DSP48E1 Configuration*

The *DSP48E1* primitive allows a maximum of four different inputs (30-bit, 18-bit, 48-bit, 25-bit). However, as regards arithmetic operations that are not involving multiplier nor pre-adder needs, just two 48-bit inputs may be used (one is resulting from the combination of 30-bit and 18-bit inputs). After having carefully read the datasheet [12], the reader should figure out one of the most remarkable features of this component, i.e., the Single-instruction-multiple-data (SIMD) arithmetic unit. This feature allows the *DSP* to work in parallel onto multiple arithmetic operations, whose operands are concatenated in the 48-bit input. Said with different words, one can perform four different 12-bit operations or two different 24-bit operations, using the same component, and feeding the operands through the 48-bit inputs. This feature is exploited to run two comparisons in parallel over two 24-bit numbers, that is also the size of half MAC address (more specifically, the OUI). This motivation is justifying the design choice of saving only the OUI in the blacklist. The *DSP* executes its operations in a combinational way: however, in order to be correctly integrated into a sequential design, there are available registers for correctly pipelining both inputs and outputs. In general, these registers should be activated, also to achieve a better timing in the design. In this case, only one pipeline register was enabled,

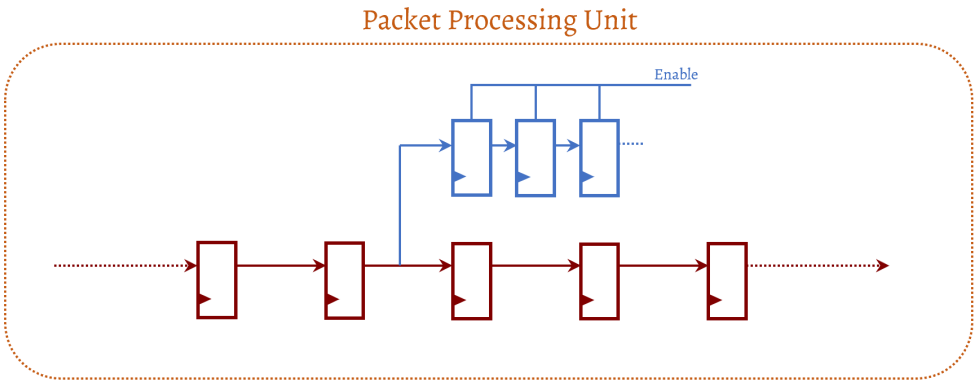

Figure 33: To acquire a signal from the "supply chain", it is sufficient to wire an intermediate signal to an external register. Such register has to be enabled at a specific time with respect to a reference, that, in this case, is coincident with the arrival time of the first byte

for both inputs and output (up to two pipeline registers are available for the inputs). Being the pipeline registers instantiated in the design, indeed sequential, the developer has to provide a reasonable clock.

Due to the reasons introduced before, and in general to speed up the design, the clock is chosen to be running at 250 MHz, thus exploiting the previously generated clock, in phase with the *gmii_tx_clk*. The choice is complying with the datasheet; therefore the output is guaranteed to be stably generated in one clock cycle [57].

Summarizing, the *DSP* will receive two 24-bit values from memory, concatenated in one 48-bit input. The second 48-bit input will receive the 24-bit OUI, concatenated twice. Such 24-bit OUI is extracted as soon as the Packet Processing Unit receives the packet bytes. After having detected the frame preamble, since the structure of an Ethernet Frame is fixed, the PPU can count the number of received bytes and know exactly the MAC address position inside the "supply chain". As shown in figure 33, as soon as the desired bytes transit through three specific consecutive registers[1], their values are conveyed towards the DSP, that can successfully load the corresponding operand register. After the subtraction is carried out, the sign of the output is checked by inspecting the *carryout* register. Such register is supposed to be 1 if the first operand is greater (or equal) than the second, 0 otherwise. The proof can be easily figured out through 2's complement algebraic

[1] recalling that each register stores an 8-bit input, hence three consecutive registers host 24 consecutive bits.

analysis. The following example on a 4-bit pair of operands is left to the reader, recalling that:

- The prefix *0b* expresses the following number in binary form
- The 2's complement of an N-bit number (i.e., its negated form) is obtained by negating all its N bits and then summing 1

$$a = 2 = 0b0010 \quad b = 3 = 0b0011$$

$$a - b = a + (-b) = -1$$

$$0b0010 - 0b0011 = 0b0010 + 0b1101 = 0b1111, \quad carryout = 0$$

$$b - a = b + (-a) = 1$$

$$0b0011 - 0b0010 = 0b0011 + 0b1110 = 0b0001, \quad carryout = 1$$

2. *BRAM Configuration*

Now that the DSP block is ready to compute all the comparisons, the BRAM must be prepared accordingly, to feed the right inputs. In this section, the hardware configuration of the BRAM will be checked out, although the remaining software programming will only later complete the analysis. Therefore, the reader has to temporarily assume that the PS can write the memory.

A BRAM cell can be configured to be a True Dual Port RAM[58]. This means that there are two separate interfaces to interact with the stored contents. Assuming that the memory is correctly configured, the DSP block needs to fetch two entries per time, as men-

BRAM interface	
clk	Indeed, the clock signal
address	Desired address to be read. Its size has to be configured
din	Input data bus: not useful for reading memory. Grounded
dout	Output data bus: its size has to be configured
en	Master enable: when high, it enables the BRAM cell to be accessed
wen	Write enable: not useful for reading memory. Grounded

Table 6.2: Bram interface signal description

tioned in the previous section. Being the configuration step on behalf of the PS, the PL is chosen to be only allowed to read memory. The signals required to perform a correct read operation over a BRAM interface are listed in table 6.2.

There are several variables to be configured before instantiating one BRAM cell: among these, the *address* and *dout* buses, and the *write depth*. As regards the first variable, the most common configuration is to set the address bus width to 32 bits. This also makes it compatible with a BRAM controller from the PS side, that will have to access such memory later, for configuring it correctly. As regards the data output bus, it is chosen to be 32 bits wide, that is the minimum available size when configured to be compatible with a BRAM controller. Eventually, the memory depth is set to 1024 entries, so that, one full 36 Kb unit can be filled. Table 6.3 summarizes the final BRAM configuration. There is a useful remark about the *dout* bus size. Being the OUI a 24-bit variable, there are eight additional bits per each memory entry that may encode extra information. A reasonable design choice consists in specifying the filtering list per each MAC address, indicating if it belongs to the source blacklist, the destination blacklist or both. In this case, the OUI is stored in the least 24 bits of the 32-bit word, whereas the 25^{th} and 26^{th} bit encode its belonging respectively to the destination or source list.

The procedure to read the BRAM is quite simple, and it takes only two clock cycles. First of all, the *en* pin must be driven high for the whole read procedure, then the desired address is written on the address bus. In the following clock cycle, the BRAM unit will put on the *dout* bus the corresponding 32-bit word.

A very last remark concerning this chapter consists in the interface between the BRAM cell and the PS, that has to configure the memory after powering up the system. This is probably not the best strategy; however, a different one will be proposed in the imple-

BRAM configuration for MAC filter

Memory type	True Dual Port RAM
Address width	32
Memory depth	1024
36 Kb BRAM resources required	1

Table 6.3: Configuration of the BRAM cells used for filtering the MAC addresses

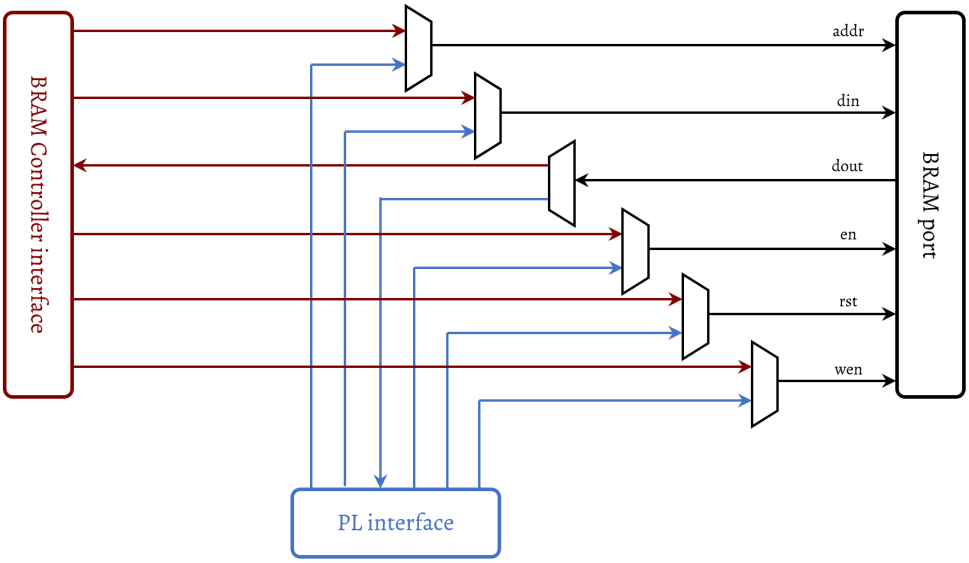

Figure 34: Multiplexing a BRAM interface: detailed schematic of the interconnections inside the PPU

mentation of the EtherType filter, that is probably better concerning wiring and resource usage. Unfortunately, there was no time left in order to upgrade the old design to include the new strategy, because the finite state machine and the hardware block design were supposed to be re-drawn.

That being said, the reader should understand that the DSP uses both the two available ports of the BRAM. Therefore, a multiplexer is required to connect also a BRAM controller. As regards the MAC filter function, the Packet Processing unit has three BRAM interfaces available: one Slave and two Master. The Slave interface is directly connected to the BRAM controller, whereas the two Master interfaces are connected respectively to the two BRAM ports. Inside the Packet Processing Unit, there is a multiplexer that is choosing whether to connect the Slave interface and thus the BRAM controller (and therefore the PS) or the DSP State Machine logic to the BRAM cell. The PS directly controls the select signal of such multiplexer through a General Purpose Input Output (GPIO) channel. Figure 34 summarizes such interconnections. There is only one critical signal that should never be multiplexed to avoid problems with timing constraints, i.e., the clock. This is the reason why the same 250 MHz clock drives both BRAM and DSP. Such a clock is generated by the MMCM described in section 6.4.1.

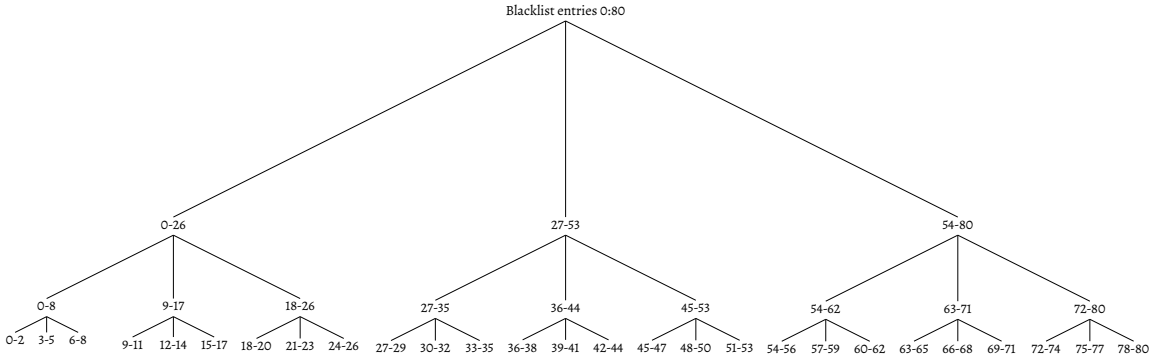

Figure 35: Decision tree of the ternary search algorithm

3. *Search Algorithm and Finite State Machine Implementation*

As mentioned in section 6.4.3, the main source of inspiration for implementing in hardware the search algorithm was the *binary search algorithm* [56]. Since the DSP is able to evaluate two comparisons per time, a *ternary* search algorithm is designed from scratches. A fixed maximum number of entries is defined during the design phase, possibly a power of 3 to make it optimized: in this case, such number is equal to 81. Then, upon its initialization from the PS, the list of entries is required to be sorted, still by the PS CPU, in ascending order. Descending order is also possible, but it will lead to a different hardware implementation. All the unused entries are set equal to the highest 24-bit value, that is 0xFFFFFF. Last but not least, no duplicates are allowed (except, indeed, the unused entries).

As soon as the half MAC address is ready to be searched in memory, the algorithm starts: the list of entries is divided in three chunks of 27 elements each, as shown in figure 35. The DSP is fed with the first element of the second and the third chunk. Depending on the results of the comparisons, one can understand which of the three blocks might contain a matching entry. The matching condition takes into account not only the half MAC address value but also the blacklist it belongs to, source or destination. Then, the algorithm starts over with the same approach, dividing into three chunks the new block, until three elements are left. The MAC State Machine keeps track of the number of cycles across the states. A summary of the implemented state machine is given below, together with a picture (figure 36):

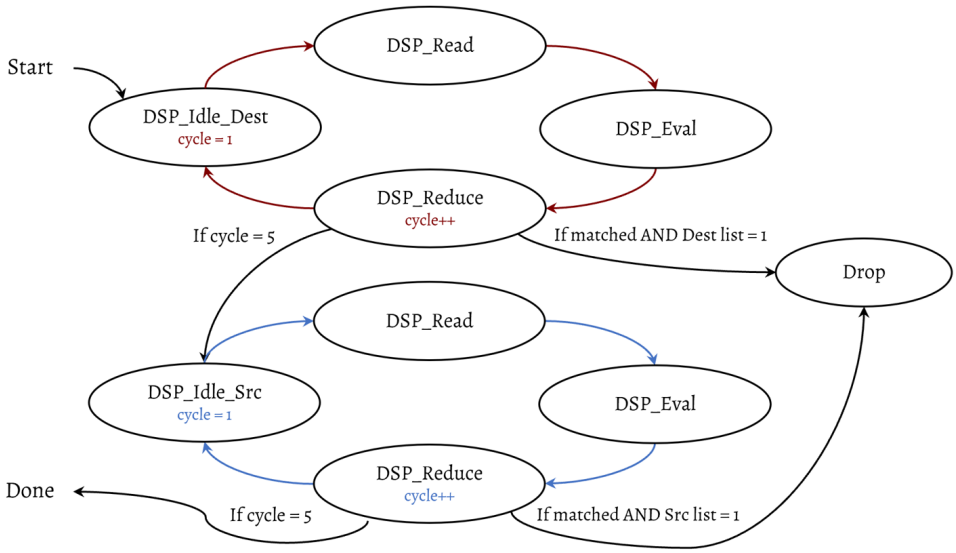

Figure 36: State machine diagram of the MAC Filtering. Destination and Source MAC Filtering are performed in the same way but changing the matching condition to the corresponding list. The variable *cycle* is counting the number of loops in the red colored mesh (Dest) or blue (Src)

- *DSP_Idle_Dest*: Idle state. The addresses to browse the BRAM are ready on their bus
- *DSP_Read_Memory*: The BRAM outputs the queried values, and they are given (concatenated) to the input register of the *DSP*
- *DSP_Eval*: The inputs are sampled by the *DSP*, which then performs the combinational operations
- *DSP_Reduce*: The *DSP* output is now available:
  - *If one of the two queried entries matches the input MAC and the corresponding list*: a flag is raised to tell the main Finite State Machine to drop the packet.
  - *If it is the fifth time this state is run*: it means that the input MAC does not belong to the blacklist. Go to *DSP_Idle_Src*.
  - *If the input MAC is greater than both the two entries*: the new entries to be fetched will belong to the rightmost chunk. Go to *DSP_Idle_Dest*.
  - *If the input MAC is smaller than both the two entries*: the new entries to be fetched will belong to the leftmost chunk. Go to *DSP_Idle_Dest*.

- *If the input MAC is between the two entries*: the new entries to be fetched will belong to the central chunk. Go to *DSP_Idle_Dest*.

There are two remarkable points to be discussed: one is good, the other is bad. The bad news is that the algorithm is not properly optimized, as the fifth cycle across memory can be avoided by reducing the maximum number of input entries to 80. Unfortunately this issue and the relative solution came out only during the development of the following filter (EtherType); moreover, the fix is not so quick. The process of updating the addresses for browsing the BRAM is the most sensitive part , therefore it was chosen to continue with the additional features, and add a new entry in the Update list. For the sake of curiosity, one example is proposed to the reader. Suppose there are just two entries in memory, whose value are 0x123456 and 0x789ABC. All the remaining 79 entries are filled with 0xFFFFFF as described before. Suppose that the received OUI is equal to 0x175317. In table 6.4, shown below, are reported all the steps performed by the algorithm.

Clearly, the last State Cycle is a waste of resources, since the DSP has two available inputs but they are used to make a single search. Anyway, in the following chapter it will be described how the improved algorithm works, solving this problem.

On the other hand, the good piece of news is that the algorithm always takes the same amount of clock periods to run, either when a match is found or not. Even though the drop condition is asserted, the MAC State Machine waits in the Idle state until the last calculated State Cycle. It would be interesting to evaluate the number of clock periods required: five cycles across four states are required by the State Machine to browse the whole memory. Since the running clock is 250 MHz fast, it will take $4\,\text{ns} * 4 * 5 = 80\,\text{ns}$. Such amount of time is equivalent to 10 clock periods in the main Finite State Machine

State Cycle	Entry number (1)	Value (1)	Entry number (2)	Value (2)	Next entry number (1)	Next entry number (2)
1	27	0xFFFFFF	54	0xFFFFFF	9	18
2	9	0xFFFFFF	18	0xFFFFFF	3	6
3	3	0xFFFFFF	6	0xFFFFFF	1	2
4	1	0x789ABC	2	0xFFFFFF	0	0
5	0	0x123456	0	0x123456		

Table 6.4: This table shows an implementation of the decision tree depicted in figure 35 applied on the example proposed

that drives the network packets in the Packet Processing Unit. Since the MAC address is made of 6 bytes, as soon as the Destination Filter is done, the Source is ready to start, because both Source and Destination OUI will be already inside the Packet Processing Unit. Considering the 10 bytes received during the Destination MAC Filtering, one can find:

- The last half of the Destination MAC Address (3 bytes)
- The Source MAC Address (6 bytes)
- The first byte of the 2-byte EtherType

This is yet another good piece of news, as the EtherType Filter can start upon reception of the following byte, in parallel with the Source MAC Filtering. Once again this is pointing out the beauty of "parallelizing" tasks when working with FPGAs.

6.4.4 EtherType Filter

Recalling section 3.4, the EtherType is a 2-byte identifier that follows the MAC addresses in an Ethernet frame. Such identifier contains the necessary information to climb the *OSI* stack, i.e., the protocol with which data is encapsulated in the payload of the frame. The reader should consider this scenario: suppose that one built an internal network in which the IPv4 protocol mediates all the exchanges of data. For security purposes, one might want to filter out, say, all the IPv6 packets, because they have nothing to do with this stream. Possibly, they might have been generated by a third party software, maybe with the purpose of stealing data. This is a good reason to implement an EtherType Filter. The working principle of such filter is very similar to the previous MAC Filter, i.e., searching in memory for a possible match. However, this is an upgraded version, correcting the small imperfections shown before. Also, in this case, there will be three steps, namely, the *DSP48E1* setup, BRAM configuration and search algorithm implementation to be checked out.

1. *DSP48E1 configuration*

Another *DSP* unit is required to make this feature work since it will run in parallel with the MAC filter. The configuration is almost the same, with two 24-bit simultaneous operations. This time, however, the EtherType is only a 16-bit entry. Moreover, the BRAM connections are chosen to be wired differently with respect to the previous one. The most significant change, as regards the *DSP*, is that there is only one port of the BRAM available to be accessed. Therefore, the two inputs must be provided by a single read in memory. As regards the pipeline registers and the clock speed, the same configuration as before is used.

2. *BRAM Configuration*

As regards the EtherType Filter, a different BRAM structure is chosen. First of all, the choice of using a separate BRAM cell with respect to the MAC Filter is obvious for two reasons: the first is that they run in parallel, the second is that the two ports are already busy; therefore another fancy multiplexer would be required to connect another BRAM interface to the same block. It also to avoid such multiplexers that the previous architecture is slightly changed. First of all, the BRAM must be programmed by the PS; therefore its Controller has to be instantiated and wired. One port is therefore reserved for this purpose, whereas the second port is connected to the DSP. This time, since the EtherTypes are 16-bit wide, it is sufficient to place them in memory, in contiguous positions, still sorted in ascending order. The EtherType State Machine will take care of one single address for querying the BRAM, and its 32-bit output will give two EtherTypes back. So, once again, the configuration of the BRAM blocks is shown in table 6.5.

BRAM configuration for EtherType filter

Memory type	True Dual Port RAM
Address width	32
Memory depth	1024
36 Kb BRAM resources required	1

Table 6.5: Configuration of the BRAM cells used for filtering the EtherTypes

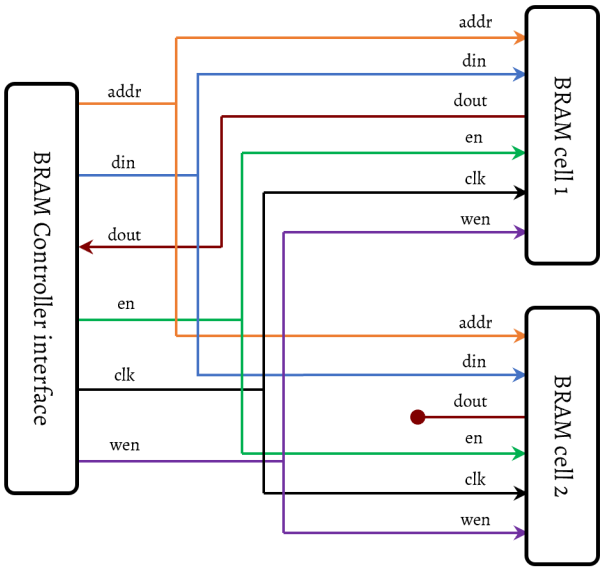

Figure 37: Interconnections between a single BRAM Controller and two twin BRAM cells

There is one last remark for the reader. The two Packet Processing Units (the reader should remember that there are two ways for the data to flow across the FPGA) are identical; as a result, they need the same BRAM entries. Whenever the PS is accessing an external memory mapped peripheral, it should correctly refer to it through its address offset. As regards the BRAM, such mapping is given only to the Controller, because it is mediating the communication. A BRAM cell itself is indistinguishable from another for the Controller point of view. That is why in this design only one Controller is used to write on two BRAM cells at the same time. The detailed schematic is shown in figure 37. There is no need for additional components, but there is a small drawback to pay. Since the *din* bus is an input for the BRAM controller, this cannot be driven twice by the two BRAM cells. Therefore, one of the two BRAM pins is left merely disconnected. This is not causing any malfunctioning because the Controller will always be able to write memory (both will be written) and to read (only one will be read, but it has the same information as its twin).

3. *Search Algorithm and Finite State Machine implementation*

The Search Algorithm, as said in the previous chapter is improved regarding optimization. Unfortunately, it is not anymore a *ternary search*, because there is only one BRAM port to be accessed. However, it is not even a proper *binary* algorithm, because the BRAM

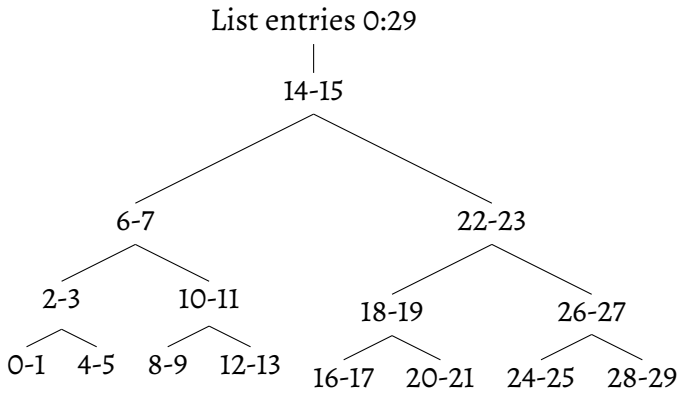

Figure 38: Decision tree of the improved binary search algorithm

is giving two entries per time. The maximum number of entries is set to 30, and only four cycles are necessary to browse the whole memory. The principle is the same as the *binary* search: the whole list of entries is split into two chunks. Then, each of them is again split into other two chunks and so on. However, if this approach is followed, it will lead to the same conclusions as to the previous implementation, that was not the best. The difference is quite peculiar: in the previous example, the entry to be queried was corresponding to the first element of the equal-size chunks. However, this was not taking into account that the *DSP* can provide both a comparison operation and an equality check. If such check provides a negative result on a specific entry, it is useless to query it once again. Therefore, following this principle, the decision tree is re-drawn, and looks like figure 38.

Summarizing, the median couple of the list is checked first. Then, if the input EtherType is:

- *greater than both the two entries*: the rightmost chunk will be next
- *smaller than both the two entries*: the leftmost chunk will be next
- *between the two entries*: the input EtherType is not belonging to the list

There is still one point to be discussed, related with the encoding of the EtherTypes in memory. In the previous example of the MAC Filter, there were eight empty bits per each entry where to store additional information. In that case, it was stored the belonging to the Source or Destination list. It was implicit, but, if this information was missing, the

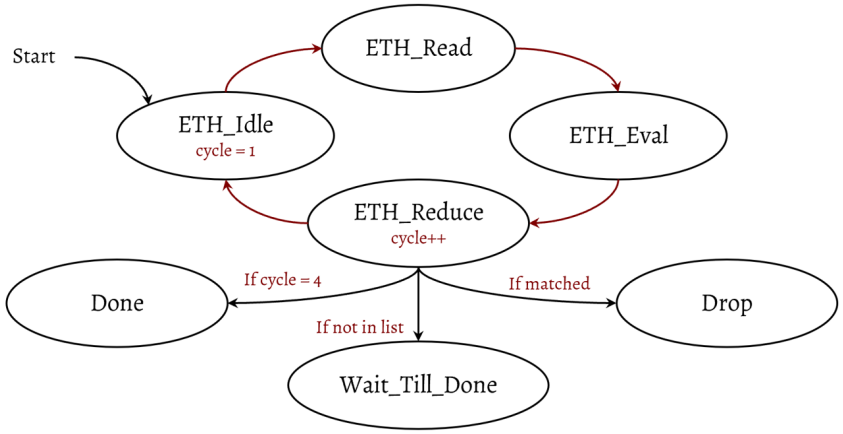

Figure 39: State machine diagram of the EtherType Filter. The variable *cycle* counts the number of loops in the red colored mesh

entry in memory was considered not to be valid. Therefore, if the user decides not to enable the MAC filter, the memory is filled with 0x00FFFFFF, leaving untouched both the 25th and 26th bit, encoding the above-mentioned information. If a packet with an OUI equal to 0xFFFFFF is received, the MAC Filter will detect the match in memory, but not in the corresponding list (either Source or Destination): this is, of course, the correct behavior. Unfortunately, there are no extra bits as regards the EtherTypes, because the 32-bit memory words are wholly filled with the two 16-bit EtherTypes. This time, if the user decides not to enable the EtherType filter, the memory is filled with 0xFFFFFFFF values. If a packet with an EtherType equal to 0xFFFF is received, the EtherType Filter will detect it in memory and drop the packet. This is indeed a wrong mistake. Therefore, even though it is not very elegant, a workaround is provided. Basically, from the PS it is possible to wire some signals directly to the PL, through a GPIO channel, as described at the end of section 6.4.3. As regards the design of the EtherType Filter, two signals are chosen to be wired through the GPIO towards the Packet Processing Units. The first one implements an additional feature that allows the user more flexibility: the EtherType list can be set to be either a whitelist or a blacklist. The former indicates that only the elements matching with the contents of the list are allowed to flow through the FPGA. On the other hand, a blacklist indicates that the elements matching with the list are not allowed to flow through the FPGA. The second input provided by the GPIO channel tells the state machine whether

the particular case 0xFFFF is included in the list provided by the user. In this way, the hardware surely knows whether to drop or save the packets with 0xFFFF EtherType.

Last but not least, the EtherType State Machine looks almost identical to the MAC State Machine and is depicted in figure 39.

6.4.5 ICMP Killer

This feature is fascinating from the security point of view. In a few rough words, it puts an invisibility cloak over the workstation connected to one side of the board. Before checking this out, the reader must know what the Internet Control Message Protocol (ICMP) is. It is a supporting protocol, used mainly to transmit malfunctionings of the network or control information [59]. Among these, there are the common network applications *ping (packet internet groper)* and *traceroute*: the first one is used to measure the time taken by a packet to reach a device in the same network and come back [60]. To do that, a device sends an ICMP packet of the type *echo request* over the network and then listens for a response. Such a response is another ICMP packet, of the type *echo reply* and it allows drawing conclusions about the time of travel. This application is widely used to check if a device is "alive", i.e., reachable, on the network. According to the standards [60], the ICMP protocol is encapsulated in IPv4. As a result, it uses IP datagrams for transport. Its assigned protocol number on IP is 1 and is written to the 24^{th} byte of an IPv4 packet.

That being said, this function was a source of inspiration to build a specific filter application. It is working straightforwardly because it checks the EtherType and the IP protocol number. If they match the ICMP fingerprint, and so, respectively, 0x0800 and 0x01 the packet might be dropped. The final word depends on yet another signal through the GPIO channel, that is set by the user in order to enable or disable this function. In the end, when such a feature is enabled, every ICMP packet is prevented from flowing through the board. Suppose that the FPGA is connecting a workstation to a local network. If another device on the same network tries to send a ping towards the workstation, such packet will be dropped. Therefore, the device will be tricked to believe that the workstation is not connected to the network.

6.4.6 Encryption Environment

The last feature implemented for the Packet Processing Units is here referred to as an *encryption environment*. The choice of these words is due to the adopted approach rather than complexity. The implemented encryption system is not very strong, and of course it must be upgraded in the future: however, the way it is implemented in the design is quite impressive, since it allows a transparent modification of data, real-time, while they are traveling through the Packet Processing Unit. The bottom line is always the same: the most powerful advantage of working with FPGAs is the possibility of running several tasks in parallel, possibly even interacting on the same set of data.

Before starting with the analysis of the encryption functions and algorithms, it is mandatory to consider the immediate consequences when the data belonging to an Ethernet packet are modified. The reader should remember that the last four bytes of a valid packet, referred to as Frame Check Sequence (FCS), contain the so-called checksum. Therefore, whenever one wants to apply modifications on the network data, the FCS must be recomputed. Usually this task is performed by the MAC controller, however, since there is no embedded MAC controller in this application, a checksum generator must be implemented first.

1. Checksum Generator

The algorithm defined by the Ethernet Standard [17] to compute the FCS is the Cyclic Redundancy Check (CRC32). Such an algorithm is easily implemented in hardware because it can be represented as a Linear Feedback Shift Register with a 32-bit characteristic polynomial [61]. As regards the combinational part, some parts of the source code were taken from an online resource [62]; on the other hand, the synchronization with the main State Machine and the architectural implementation in the design were built from scratches. The CRC32 algorithm has to start working from the first byte after the preamble. It can be seen as a black box, taking an 8-bit input (each incoming Ethernet byte) and giving a 32-bit output, starting from the fourth consecutive input received. As shown in figure 40, it is easy to branch one of the "supply chain" registers, to feed the CRC genera-

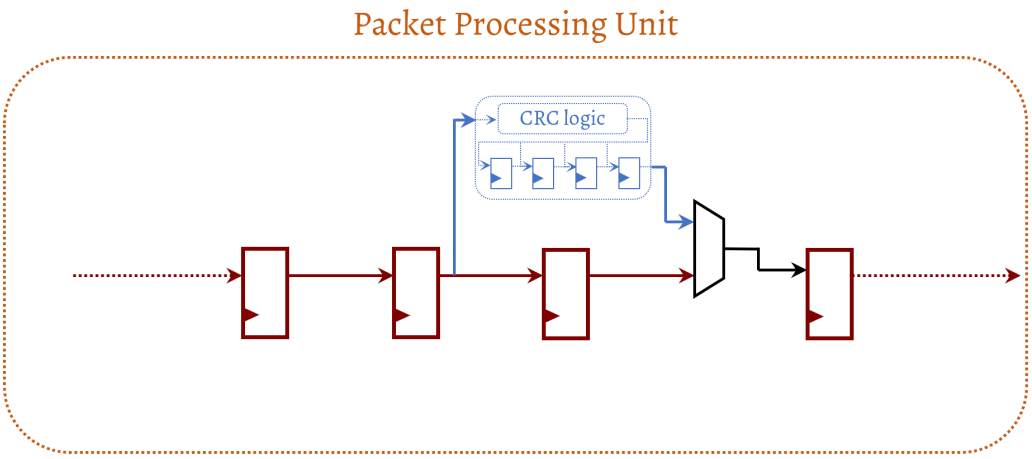

Figure 40: Architectural implementation of the CRC generator, extracting data from the main flow and multiplexing the outputs one stage later

tor in parallel to the regular flow of data. However, it is a bit more challenging to provide an "affluence" path for the computed FCS. This is indeed necessary to include the recomputed CRC, in case the data are modified on their way. To implement this insertion in the supply chain, a simple multiplexer is placed between two consecutive registers, taking as input the previous register in the chain, and the last register of the CRC generator block, whereas taking as output the following register in the chain. The selector of this multiplexer must be enabled as soon as the last byte of the frame payload flows through it. In this way, the fresh FCS is appended to the payload, smoothly.

2. *Encryption Algorithm*

Finally, the most awaited feature is about to be implemented. First of all, as a matter of simplicity, it was chosen to operate only on 8 bits per time. The working principle is simple: the user is supposed to input a password during the configuration step. Then, the Fowler-Noll-Vo (FNV) hashing function is applied on such input, and the resulting 32-bit hash is used to encrypt the 8-bit data.

The FNV hashing function [63] is chosen because it seems a good candidate for future development. At the moment, such function is executed by the PS, that writes then the output on the PS side and transmits it to the PL through a dedicated GPIO channel. However, in the future, the algorithm can be mapped into logic cells, in such a way that the

board can work on its own. The goal of this encryption implementation is to provide a Proof-Of-Concept solution, even though it has not a true *cryptographic* proficiency. The FPGA should deal smoothly with both encryption and decryption, since the packets are supposed to be used sooner or later by someone else, having the same system, configured with the same password. The goal is to find a quick function, able to produce modifications to the input data, but also able to retrieve the original information. There is a class of functions referred to as *symmetric functions*: their definition is shown in (6.2):

$$f \text{ is symmetric} \Leftrightarrow f(f(a)) = a \tag{6.2}$$

From the cryptographic point of view this a weakness, however, as said previously, the goal is to design first a Proof-of-Concept device. The simplest logic operation that is also a symmetric function is the exclusive or (XOR, symbol $\oplus$). Indeed:

$$a \oplus b \oplus b = a$$

On the basis of the previously defined principles, the procedure for encrypting the stream of data in the Packet Processing Unit can be broken down in the following steps:

- An 8-bit key is taken by slicing the 32-bit primary key
- Such 8-bit key is XORed with the stream of data, starting from the first byte after the EtherType (payload)
- A new 8-bit key is taken from the 32-bit primary key

The slicing of the primary key allows a maximum of 32 different keys before they become redundant. Such an operation is performed by merely taking a subset of the whole key and increasing the boundaries by one unit. Therefore, if the primary key bit representation is [31:0], the subsets will be [7:0], [8:1], [9:2] and so on, until the end of the key. Then, those bits are merely wrapped, and the algorithm starts over. In order to make the whole structure ordinate and deterministic, the first byte of the payload is always encrypted with the first subset of the key. To ensure this, a signal is wired to the corresponding State

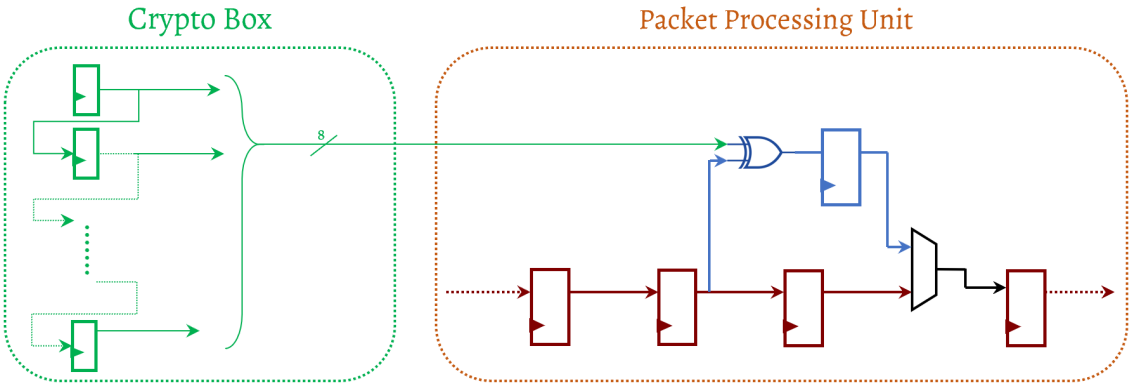

Figure 41: Architectural implementation of the encryption box, extracting data from the main flow and multiplexing the outputs one stage later

Machine. In order to replace the plain payload with its encrypted form, it is implemented the same strategy as the CRC. Therefore, a fork is wired from a specific register in the chain to the encryption box. Then, the output of the encryption box is multiplexed with the output of the following register, as shown in figure 41. Of course these components must be displaced before the CRC generator to ensure correct operation; otherwise, the CRC32 output will not be consistent with data.

Before explaining how to decrypt data, one should take into account that from an external point of view there is not any evidence of encryption. Said differently, the packet does still have a regular preamble, MAC address declaration, and EtherType. Moreover, the FCS is also confirming that the packet is valid. So how can the Packet Processing Unit detect an encrypted packet to start decrypting? That is the reason why a clear signature must be provided. The EtherType field was considered to be a right place where to store the signature. First of all, if the user decides to use the encryption box, as a matter of design choice, only the IPv4 packets will be encrypted. Such choice prevents the FPGA to mess up with other protocols that are ensuring the correct mapping of a device on the network (i.e., Address Resolution Protocol, ARP). The corresponding EtherType of regular IPv4 packets is 0x0800, so a new EtherType is searched, in order to label the encrypted packets. Such custom EtherType has to be unique, not taken by other protocols; consequently, a suitable candidate is 0x1753. In conclusion, two additional registers are included in the "supply chain", to allow the insertion of the new EtherType in case the en-

cryption feature is enabled and an IPv4 packet is received. The reader should know very well how to do this job, after the CRC generator and encryption box examples.

As soon as the leading State Machine detects the custom EtherType, it raises a flag to decrypt the payload in the Packet Processing Unit. First of all, the previous EtherType has to be restored; then the payload is processed by the encryption box, which follows precisely the same steps performed during encryption. If the password used to generate the key is the same, it is 100% sure that the original data is restored, since the XOR is a symmetric operation and the procedure is carried out in the same order. Indeed, the FCS will be computed accordingly, and the PHY will gladly accept the restored packet.

6.5 Final Hardware Design

The final hardware design looks like Figure 42. Although the picture is complicated to be read by just a glance, the reader can find all the previously described components. There are only a few more words to be written about the reset management and some useful signals that guarantee the correct interaction between the logic-fabric, the PS and the user.

6.5.1 Reset Management

As soon as the system is powered up, all the peripherals instantiated in the design must be reset, in order to achieve a defined initialization before running the main application. Such reset signal is usually asserted as soon as all the clocks are generated and stable. Luckily, the Clock Generator instantiated in the design, mentioned in section 6.4.1, has one useful output, namely, *locked*. As written in [64], when the *locked* output is asserted, it indicates that the output clocks are stable and usable by downstream circuitry. Hence, this signal can be used to trigger the reset of all the peripherals. In particular, a small state machine is defined, taking as input the *locked signal* and giving as output the reset signal. As soon as the input is asserted, the state machine counts, roughly, 50 ns before raising the reset signal, lasting one clock cycle (8 ns).

Signal	Description
MASTER_CONTROLLER_ENABLE	Enables the PPU as soon as the PS is done with configuration
IS_PL_WRITING_BRAM	Switches the control of the BRAM between the PS (0) and the DSP (1)
IS_ICMP_KILLER	Enables the relative feature in the PPU (see section 6.4.5)
IS_ETHERTYPE_BLACKLIST	When high (1) the EtherType list is a Blacklist, when low (0) a Whitelist
IS_ETHERTYPE_SPECIAL_CASE	Indicates that the EtherType 0xFFFF is a true input of the list
IS_PORT_0_CYPHERING	Enables payload encryption out of Ethernet Port 0
IS_PORT_1_CYPHERING	Enables payload encryption out of Ethernet Port 1

Table 6.6: List of control signals through the GPIO interface

6.5.2 Control and Debug Signals

Although it is not easy to find oneself bearings, the first signal the reader should check out from figure 42, is the so-called *MASTER_CONTROLLER_ENABLE*. Such signal is added to the control logic of the FIFOs but in general the whole Packet Processing Unit. If the software is still in the configuration phase, the PPU should not be active. That is why at the end of the configuration phase, this signal is raised and propagated through the GPIO to the PL. The *link_status* signal described in section 5.2.1 plays the same role in controlling the flow through the PPU: whenever one of the two interfaces is down, the PPU should immediately stop. That is why this couple of signals is conveyed to a synchronizer block, referred to as *Sync_Master_Enable*: as the PPU works synchronously, also the reset signal must be synchronous with the main clock to avoid unexpected behaviors. This concludes the set of control signals delivered by the GPIO interface, and they are well summarized in table 6.6. As regards debug signals, there is nothing special to mention: they are all routed to different LEDs on the board, and they are summarized in table 6.7.

Signal	Description	LED color
PACKET_DETECTED	Notifies the detection of a frame preamble	G
PACKET_FLUSHED	Notifies that the current packet has been dropped due to filtering	R
RECEPTION_ERROR	Asserted when both valid and error signal are high. The packet is anyway dropped	R
clock_speed[1:0]	Indicates the speed of *rgmii_clk*. Green light alone when 125 MHz	G- R
speed_mode[1:0]	Indicates the *GMII to RGMII* configuration for TX clock. Green light alone when 125 MHz	G- R

Table 6.7: List of debug signals routed to the board LEDs

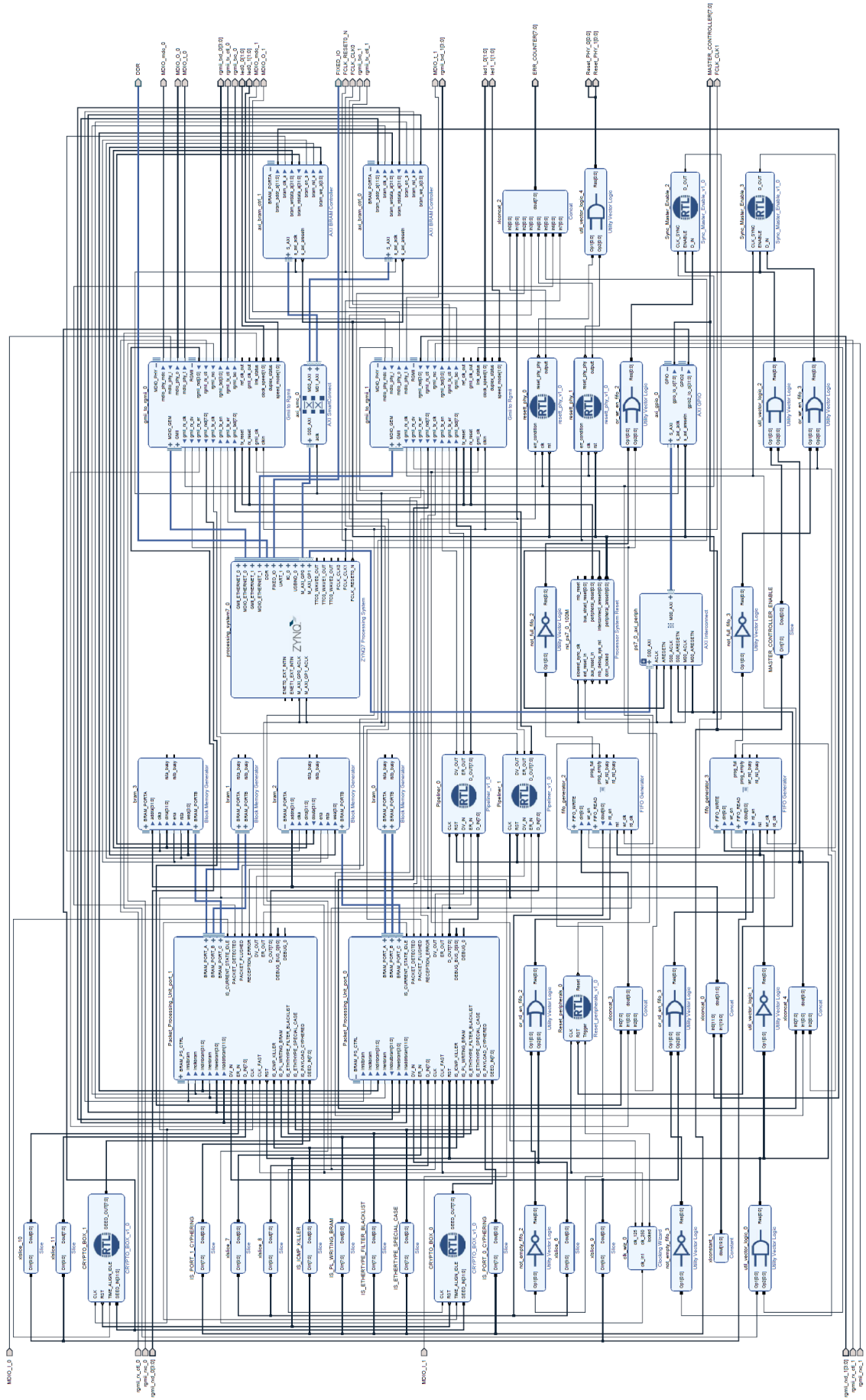

Figure 42: Final Block Design layout of the Proof of Concept

6.6 Software Design

The source code used to configure the two external PHYs is not complicated. The base is similar to the First implementation of section 5.4. Few tasks have to be performed by the processor on the PS side, and they are respectively:

1. Initialization of the GPIO channel
2. Initialization of the two Ethernet Instances
3. Setup the MDIO clock
4. Setup the *GMII to RGMII* speed mode
5. MAC filter configuration
6. EtherType filter configuration
7. ICMP Killer setup
8. Encryption configuration

Point number 1 requires just a simple instruction:

```
XGpio_Initialize(&GpioPtr, GPIO_DEVICE_ID);
```

where `GPIO_DEVICE_ID` is the memory mapped address of the peripheral. Then, from point number 2 to 4 the reader can refer to section 5.4.

The user configures the MAC filter through an interactive User Interface (UI) over the Universal Asynchronous Receiver-Transmitter (UART). As mentioned in the Physical Setup section, the test workstation is connected to the board through UART via a USB connection. Therefore, such a workstation can open a serial terminal to interact with the board via the keyboard. The UI is designed in such a way that allows the user to insert the half MAC addresses (OUI) to be filtered specifying for each of them if they have to be included in the Source list, Destination list, or both. The UI checks for the existence of duplicates and refuses to store them. In the end, the list is sorted by the embedded C

quicksort algorithm [65], and finally encoded to the corresponding BRAMs, one per each Packet Processing Unit.

The procedure is carried out almost the same way as regards the EtherType Filter: the difference is that a list of the standard entries is displayed on the screen, to the help the reader identifying the right protocol. Such list is also reported on table 6.8. Eventually, the user is asked for the list configuration, either blacklist or whitelist.

During the Encryption setup, the user is asked to insert a password, that is double checked to avoid unwanted inputs. In the following, the user must indicate what the Ethernet port connecting the board to the external world is. Indeed, the encryption feature must not be enabled on the port that connects the user's workstation to the FPGA, because otherwise, the workstation will receive a lot of incomprehensible data. Instead, both the two Packet Processing Units are always able to decrypt any packet with a 0x1753 EtherType. Lastly, the ICMP Killer setup is a simple yes/no question.

EtherType	**Corresponding Protocol**
0x0800	Internet Protocol version 4 (IPv4)
0x0806	Address Resolution Protocol (ARP)
0x0842	Wake-on-LAN
0x86DD	Internet Protocol version 6 (IPv6)
0x809B	Apple Talk
0x8870	Jumbo Frames
0x88CC	Link Layer Discovery Protocol (LLDP)

Table 6.8: List of the most common EttherTypes

CHAPTER 7

RESULTS

All the three different designs proposed in this thesis were tested, however only the last one (Proof of Concept) was tested thoroughly. Starting from the preliminary approach, being the BRAM slow clock frequency limiting the operations, the system is not able to reply a packet back to the sender in less than 50-100 µs. That is of course not good at all considering the 1Gbps link. In the worst case, considering the smallest packet size of 64 bytes, including checksum, the link can drive almost 2 million packets per second. In order to be able to respond in time, still in the worst case, the software should be able to reply the packet back in a time window of the order of 1 µs. One of the weaknesses of the first two designs, apart from being slowed down by memory management, is that the time of response depends both on the packet size and on the packet rate. Therefore, the system must always be tested in the worst case (minimum packet length) to draw some conclusions. Figure 43 shows the minimal User Interface.

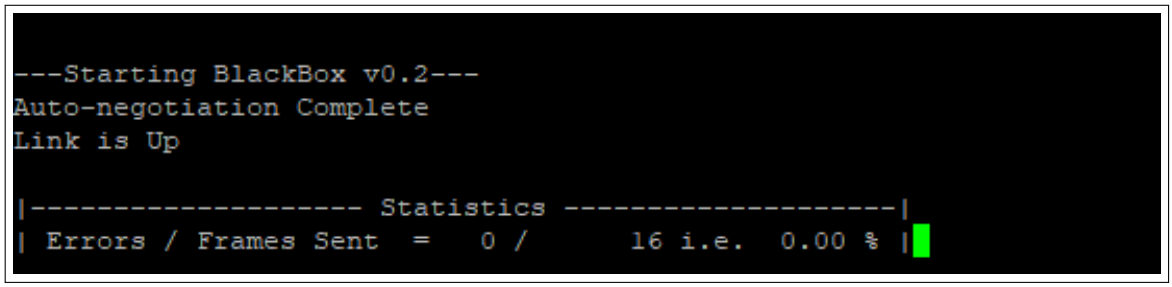

Figure 43: Minimal User Interface of the first Frame Repeater designed. The number of errors refers to the condition in which the FIFO on RX path is full and cannot find a free Buffer Descriptor in list

7.1 Simulation

As regards the final design, there is way more to say. Such a complex hardware design cannot be implemented without an adequate simulation; therefore a supplementary testbench must be added to the main project. Such testbench is better to be loaded with authentic data in order to recreate a realistic testing environment. Hence, before starting with the simulations, a couple of debug probes were used to sample a few packets directly from the GMII interface. It is interesting to see that when the line is idle, not transmitting any packets, the RX data path is steady on 0xDD value, with both *data_valid* and *error* signals low. Then, one byte before the *data_valid* is asserted and the preamble starts, the data value changes to 0xFF and the *error* signal is asserted. In figure 44 is shown the simulation of the Packet Processing Unit, with a matching condition found on the Source MAC Address. The signals *state_reg* and *state_next* show respectively the current and the next state as regards the main Finite State Machine, driving the Packet Processing Unit. Then, the array *PIPELINE* is a subset of what was referred to as "supply chain" in the previous chapters. The beginning of such chain of registers corresponds to the 24^{th} element, and it contains only the 8-bit data bus from the GMII interface. *BUF_DV_ER* instead, contains the *data_valid* and *error* signals. The two separate streams are flowing in parallel, but on separate variables, since the modification on data is not involving the control signal, and therefore it is more comfortable to refer to a single signal rather than a subset.

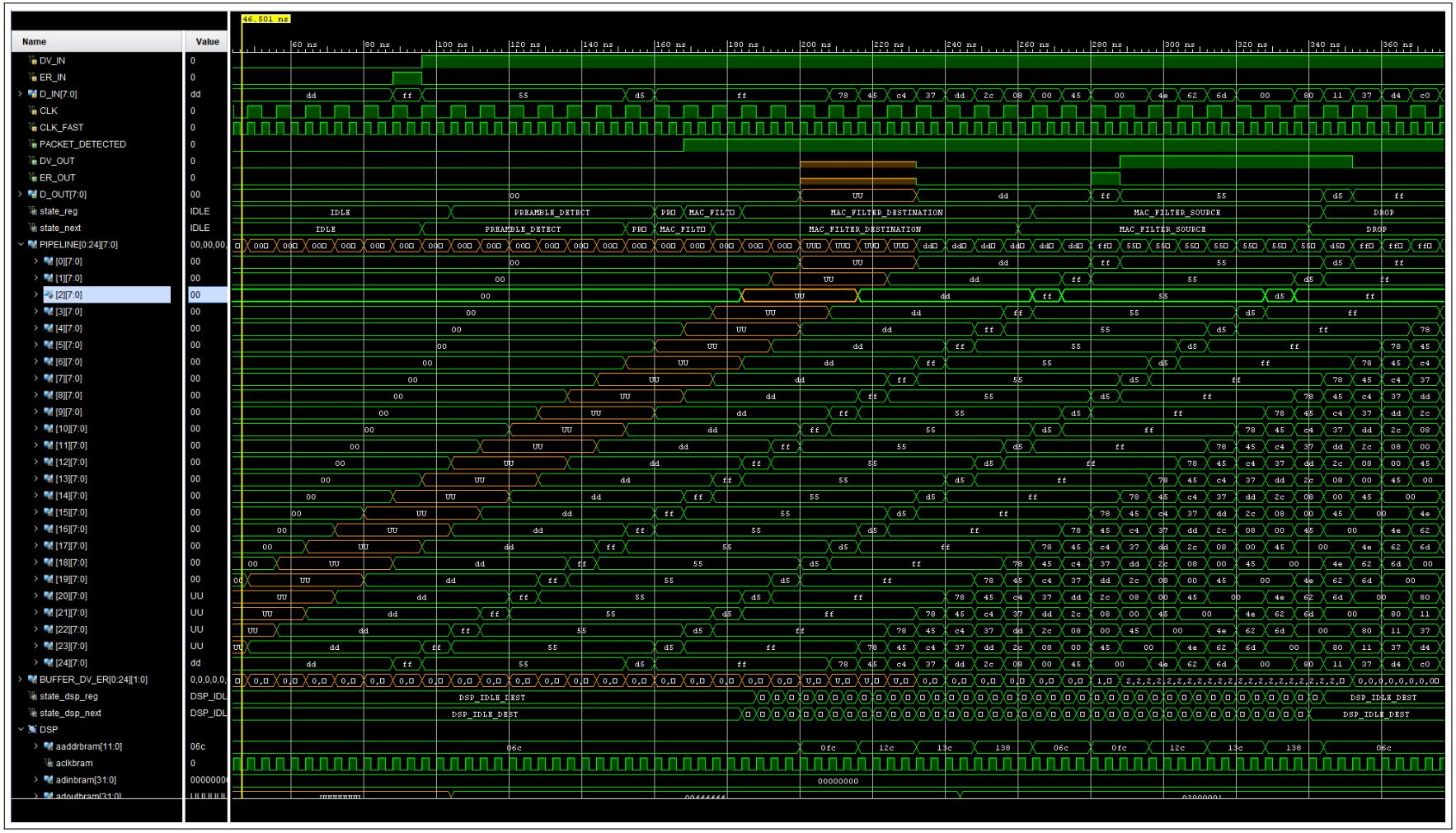

Figure 44: Simulation waveforms of the Packet Processing Unit main State Machine. Notice that the Undefined signals in the *PIPELINE* registers are just coming from the initialization of the state machine

7.2 Implementation

As soon as the hardware description language sources are synthesized and pass the simulation check, the elaborated netlist needs to be placed on the corresponding components, according to the board model. That is not a trivial task since the compiler has to take into account the resource availability and the length between interconnections, to meet all the timing constraints. The strategy used to help the tools in this complicated task is the so-called *Performance_Early_Block_Placement*: as said in the User Guide by Xilinx [66], this strategy is mostly focused on the timing-driven placement of RAM and DSP blocks. The RAM and DSP block locations are finalized early in the placement process and are used as anchors to place the remaining logic. Moreover, one additional constraint was set to relieve the placement task. Such constraint allows the user to define a specific region over the hardware floor plan and assign to it some of the components to be implemented. In this case, the Packet Processing Unit is placed close to the I/O pins, as shown in the detail of figure 45. After the implementation step is done, one can realize that the amount of used resources is meager, as shown by the full floor plan in figure 46 or table 7.1.

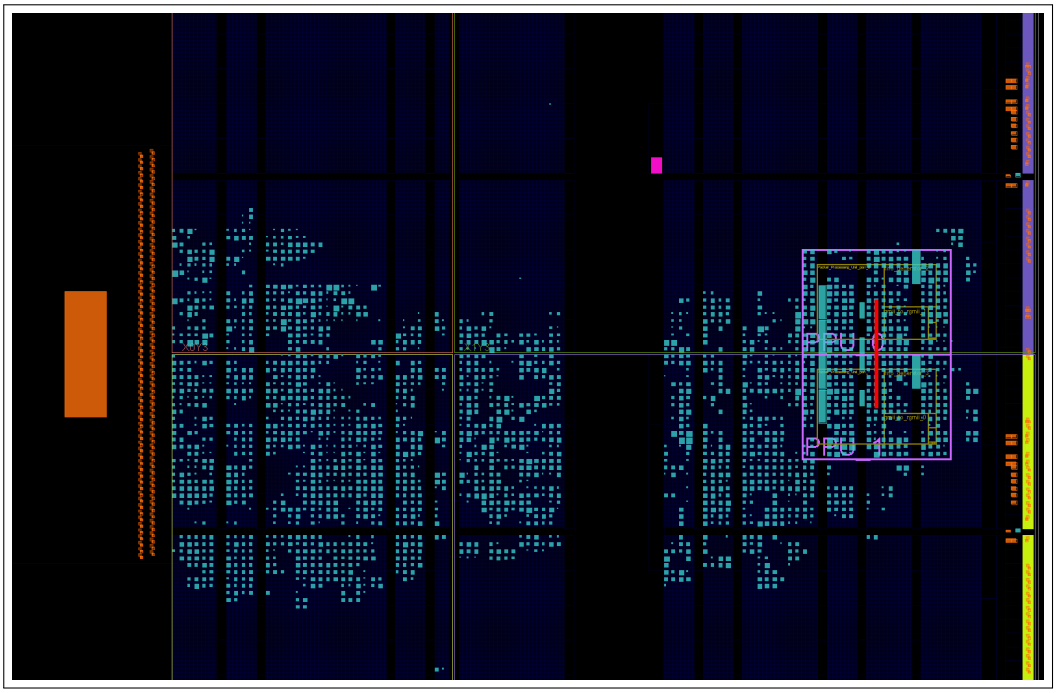

Figure 45: Closeup of the Zynq7030 hardware, showing the Packet Processing Unit confined in the pink rectangle, close to the Input/Output side

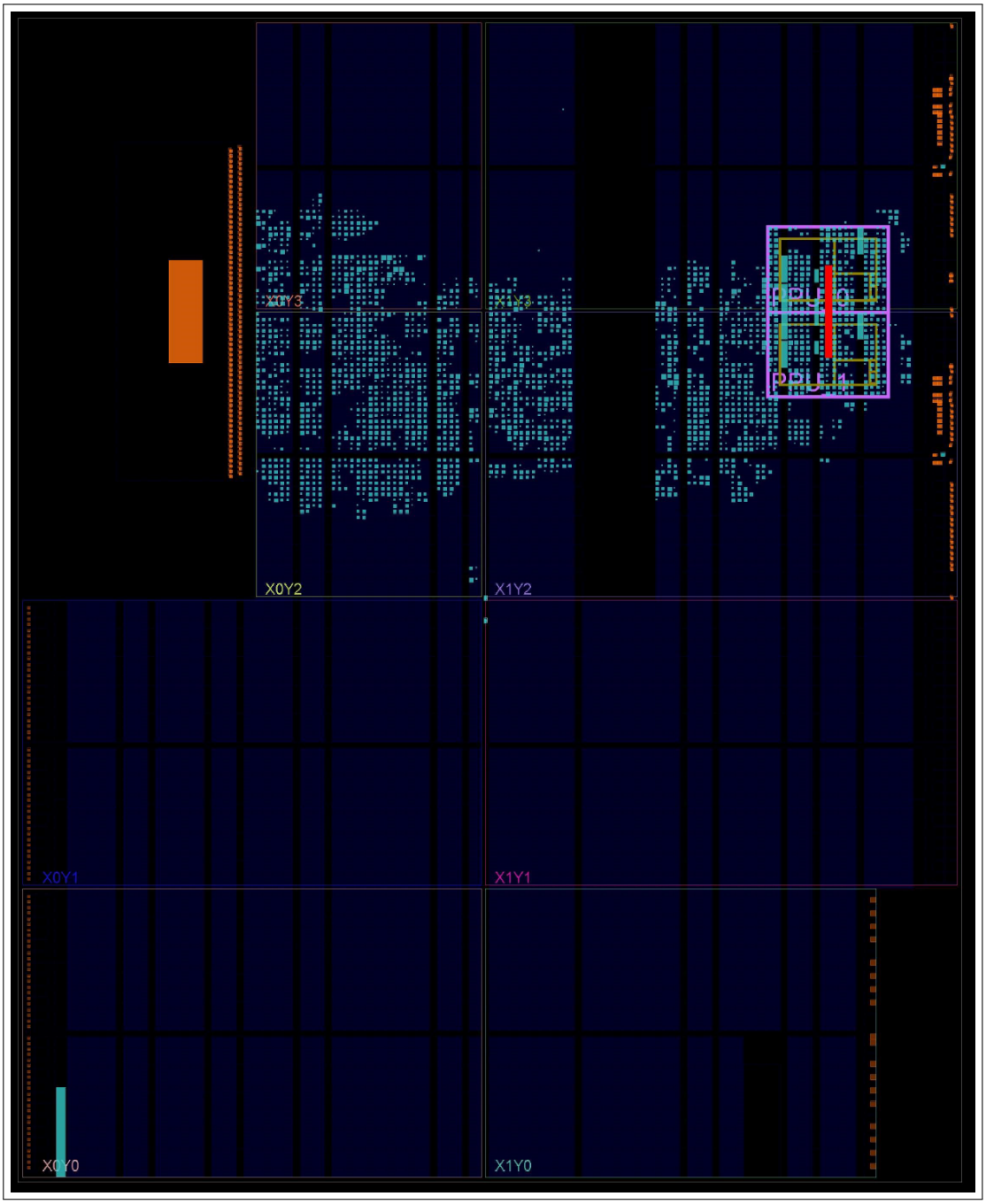

Figure 46: Full overview of Zynq7030 hardware, containing the implemented design

Resource Utilization on the 7030 module

LUT	8%	DSP	1%
LUTRAM	3%	I/O	24%
FLIP FLOP	4%	BUFG	19%
BRAM	2%	MMCM	20%

Table 7.1: List of used resources after implementation, with respect to their availability on the 7030 module. The BUFG is a global clock buffer and also the most commonly used clock routing resource. These truly global clocks can connect to every clocking point on the device, bringing a healthy clock signal to all the components that need it[54]

7.3 Testing the Device with Real Data

Third-party software is required to test the design in the real world. In particular, the workstation used Ostinato to generate traffic, whereas WireShark®to analyze the packets exchanged through the Ethernet interfaces. The first test performed was concerning a transparent usage of the FPGA: all the security features were turned off. As a result, the board was only supposed to route the traffic from one port to the other. The physical setup is shown in figure 47. In figure 48 it is possible to see how the data on the right (from `Ethernet2` interface) are identically transmitted to the left side interface (`Ethernet`) and vice versa. To reach full line speed in a real case scenario, a nice experiment was performed. The board was configured to be transparent and was connected from one side to the workstation, from the other side to the Internet. Upon a successful connection, the workstation was even able to load and stream an 8K resolution video!

The example depicted in figure 49, instead, shows that the MAC filter is configured to drop all the packets directed towards a Xilinx device (OUI = 0x000A35). In this case, such packets are generated by Ostinato software, which allows to craft network packets with custom Destination or Source MAC addresses. As clearly shown in the picture, all the packets sent through the `Ethernet` port are reaching the `Ethernet2` port, except the ones with Xilinx OUI in the Destination field (they are highlighted in white). The interface for configuring the MAC is very intuitive, as shown in figure 50 and figure 51.

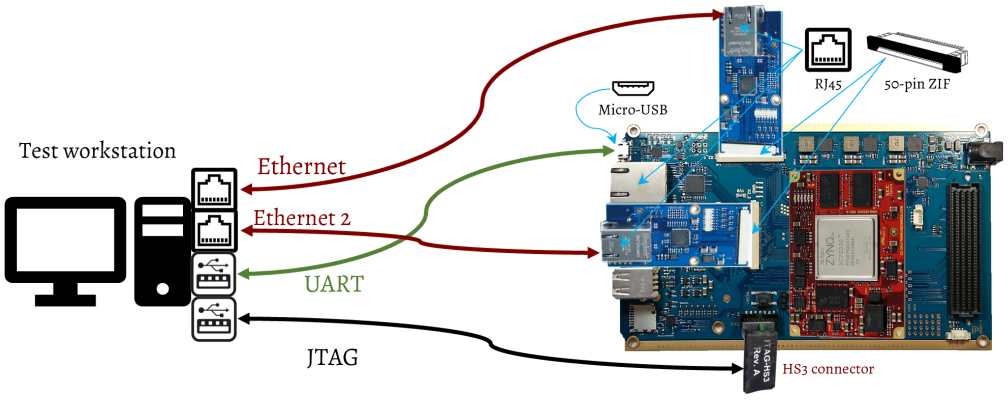

Figure 47: Final setup for testing. The interfaces labeled as *Ethernet* and *Ethernet 2* are correctly shown on the picture

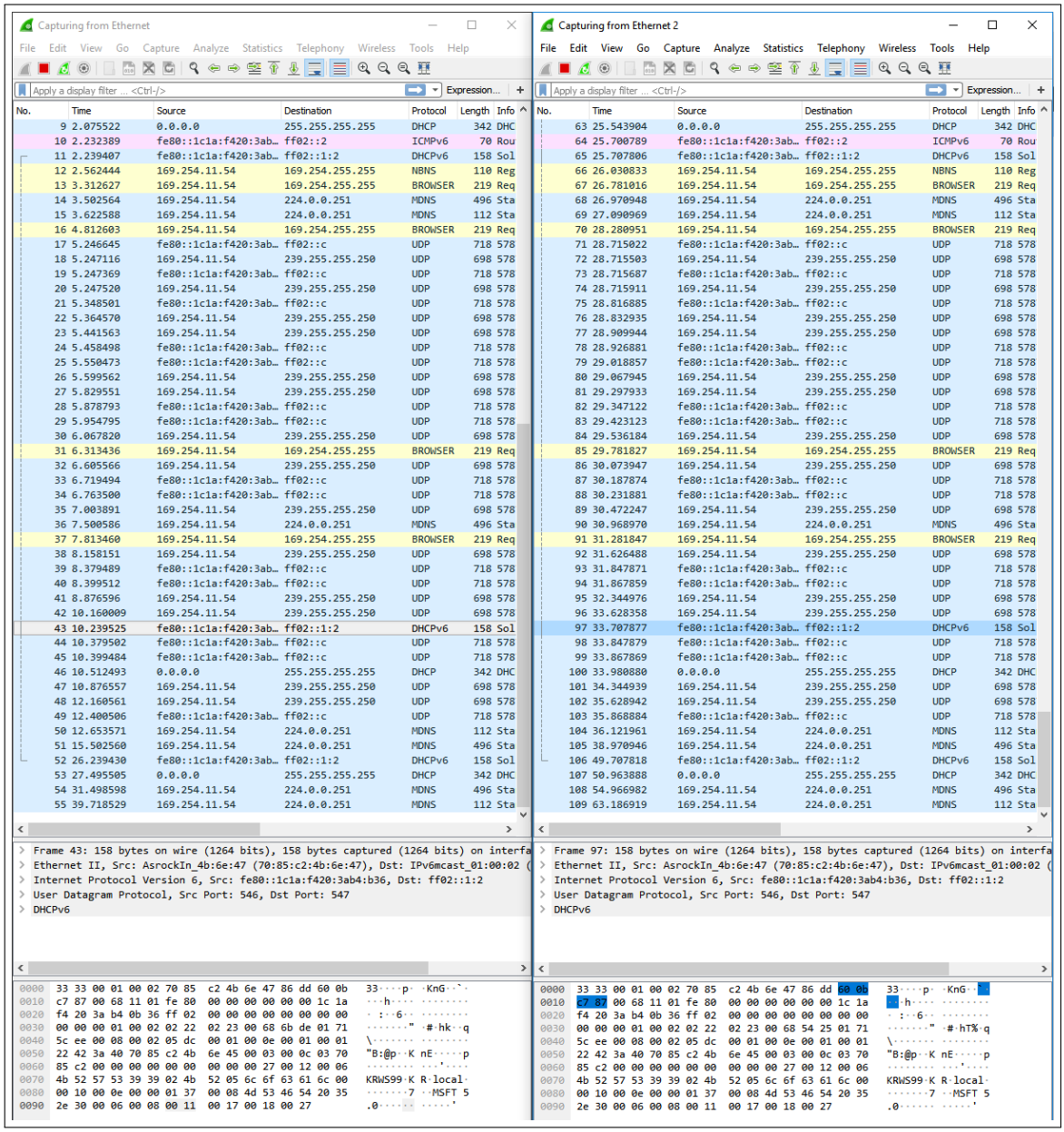

Figure 48: Transparent operation of the FPGA, simply forwarding packets from one port to the other

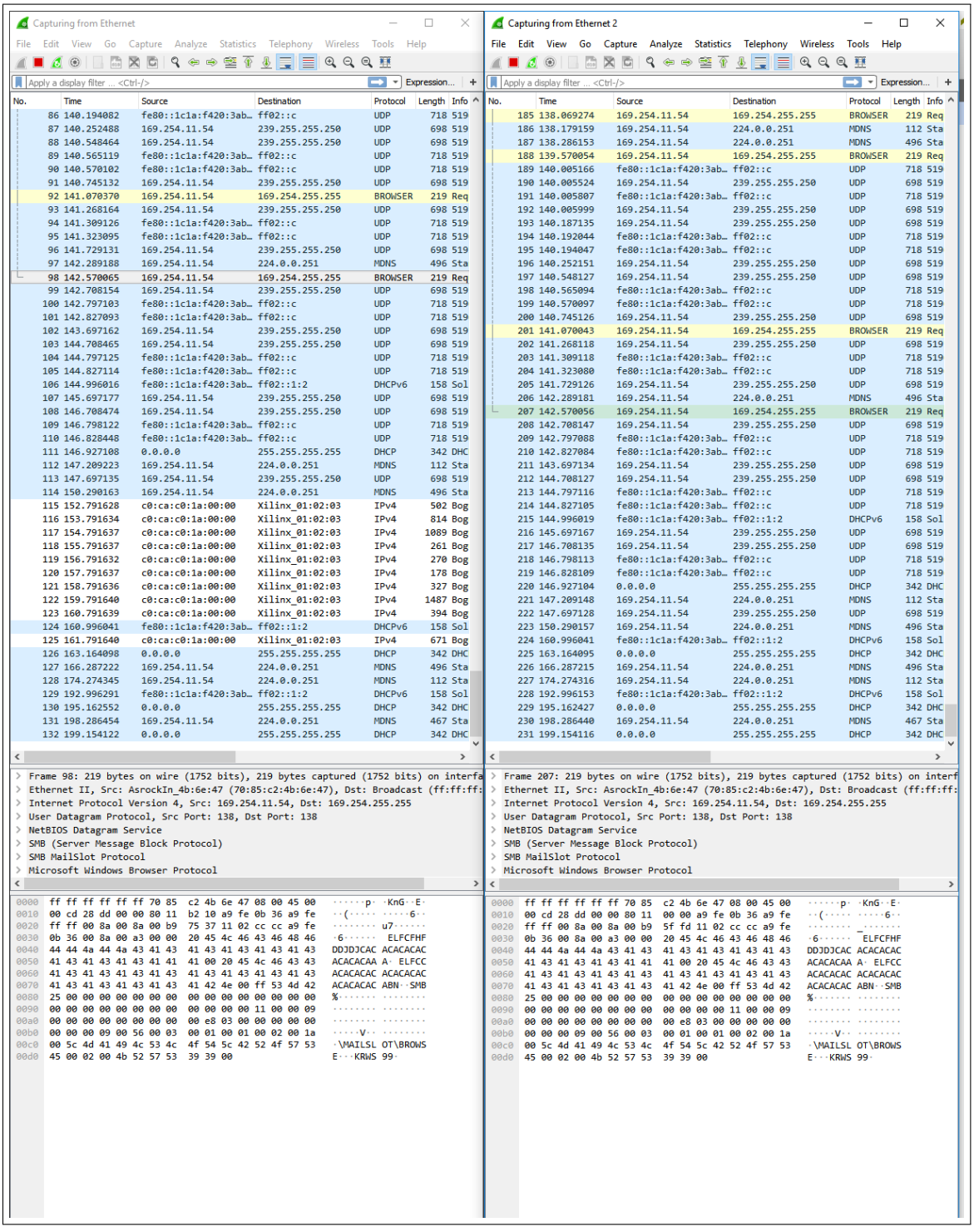

Figure 49: MAC Filter configured to drop all the Xilinx packets. Such packets are injected into the *Ethernet* interface, but they will never reach *Ethernet 2*

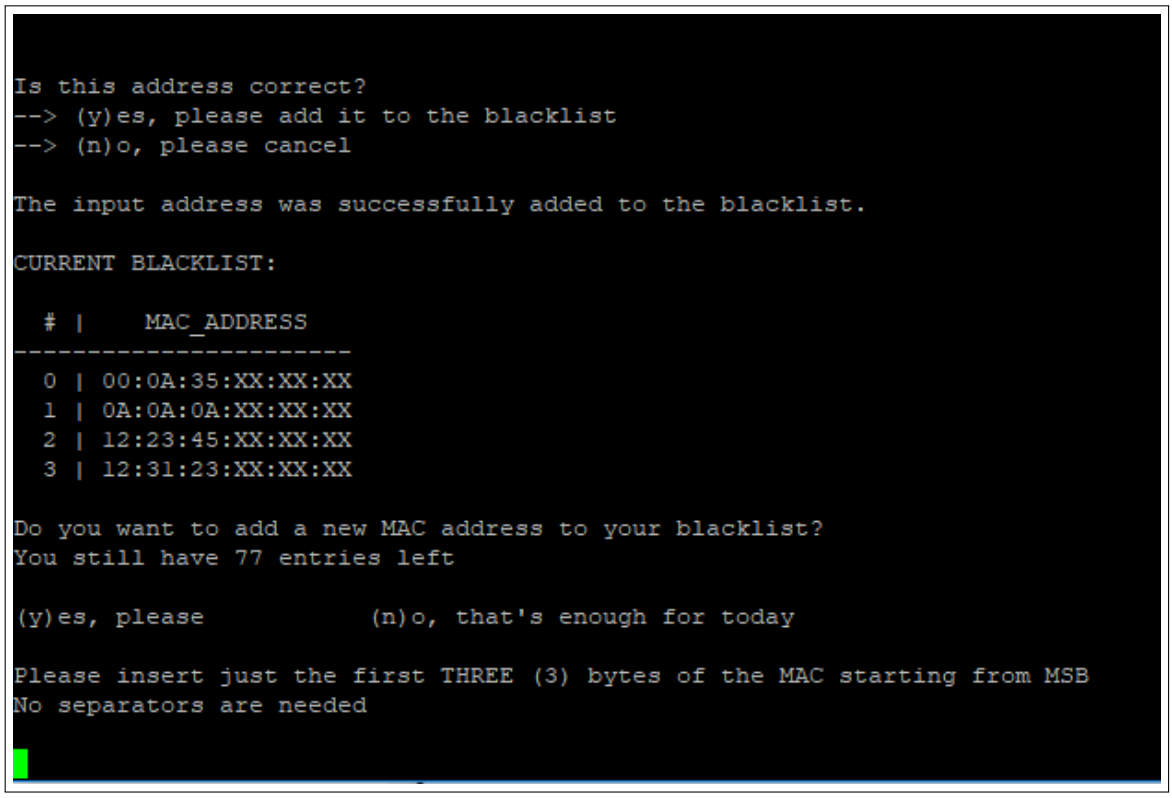

Figure 50: MAC Filter configuration interface: the user is asked to insert each OUI (first three bytes of the MAC) and confirm. Then is shown a summary with the number of entries left plus a table containing the previously inserted ones

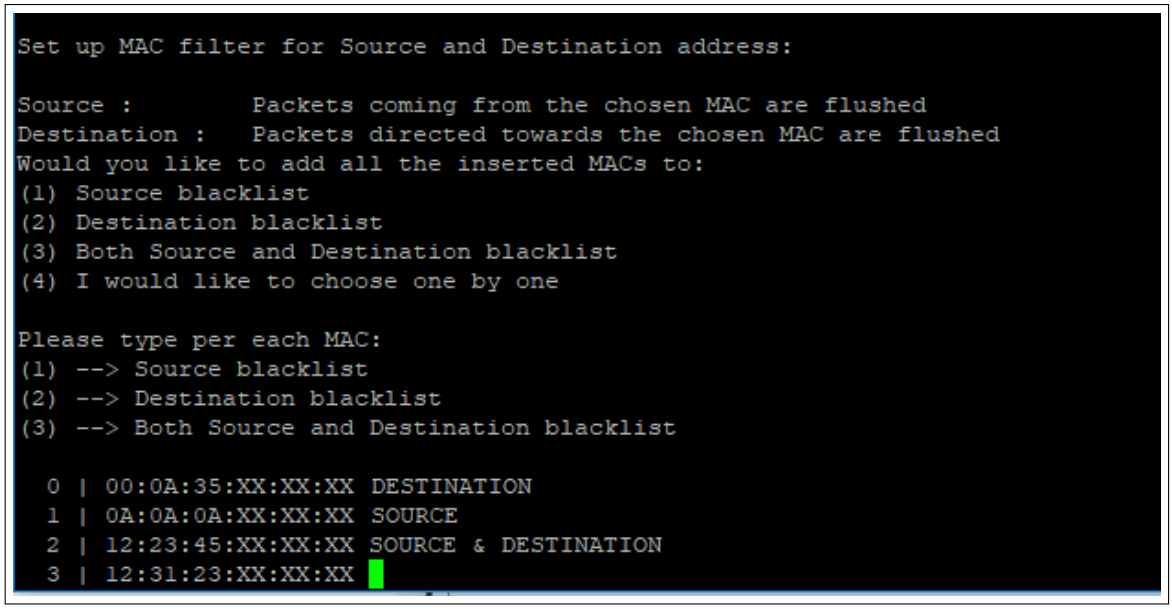

Figure 51: MAC Filter configuration interface: by pressing the indicated keys (1, 2, 3, 4) the corresponding actions are performed. In this case, each OUI is assigned to a specific list: Source, Destination or both

The last test involves the encryption/decryption engine. As shown in figure 52, the user is asked to indicate the encryption port and then to type the password twice. Then, the workstation crafts a bunch of packets and pushes them through one of the two interfaces, say, `Ethernet2`; if the process is successful, one expects to receive modified information per each packet from `Ethernet` interface. Recall that the physical setup is the same as figure 47.

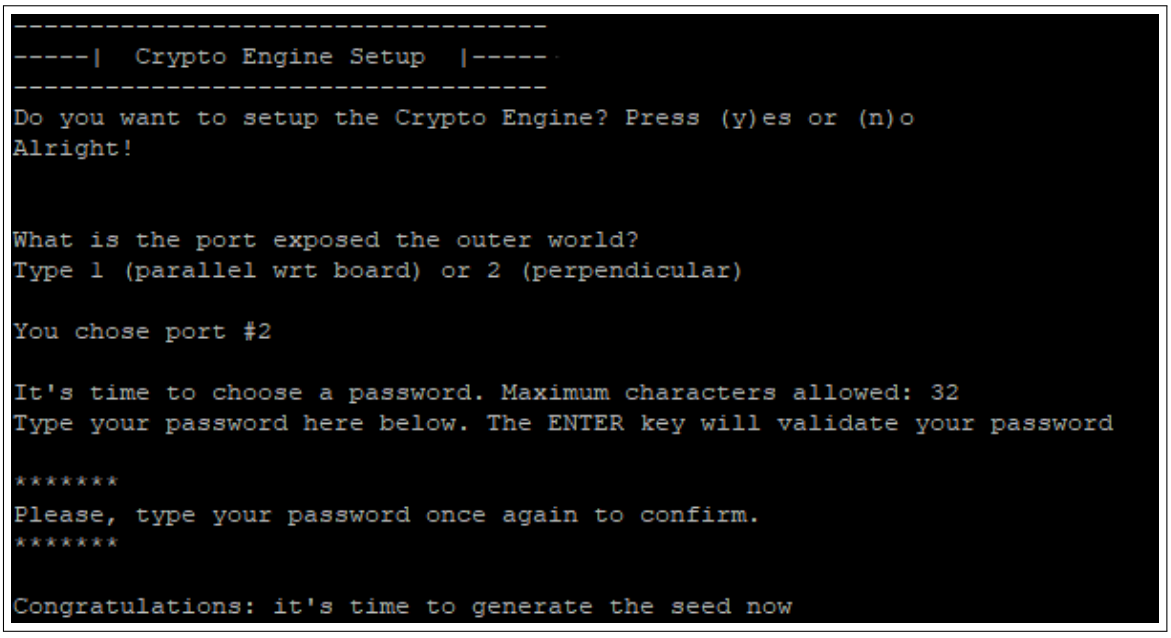

Figure 52: Crypto engine configuration. The user is asked to indicate what is the port connected to the external world, to enable encryption correctly. The only visible difference between the two ports is their displacement (parallel or perpendicular to the board)

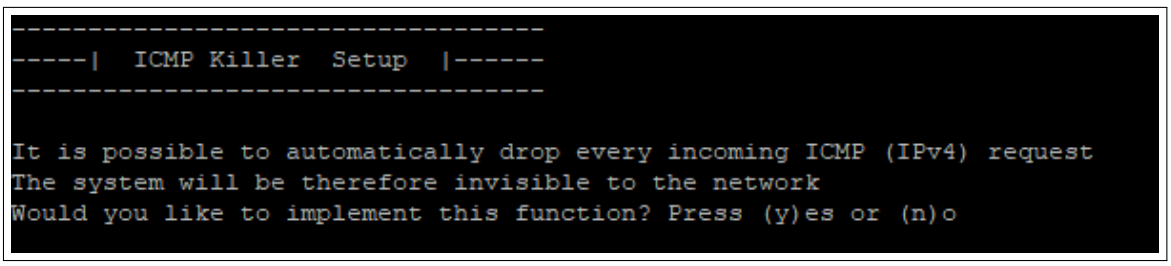

Figure 53: ICMP killer configuration. The user is informed of this possibility and is asked whether to enable this feathre or not

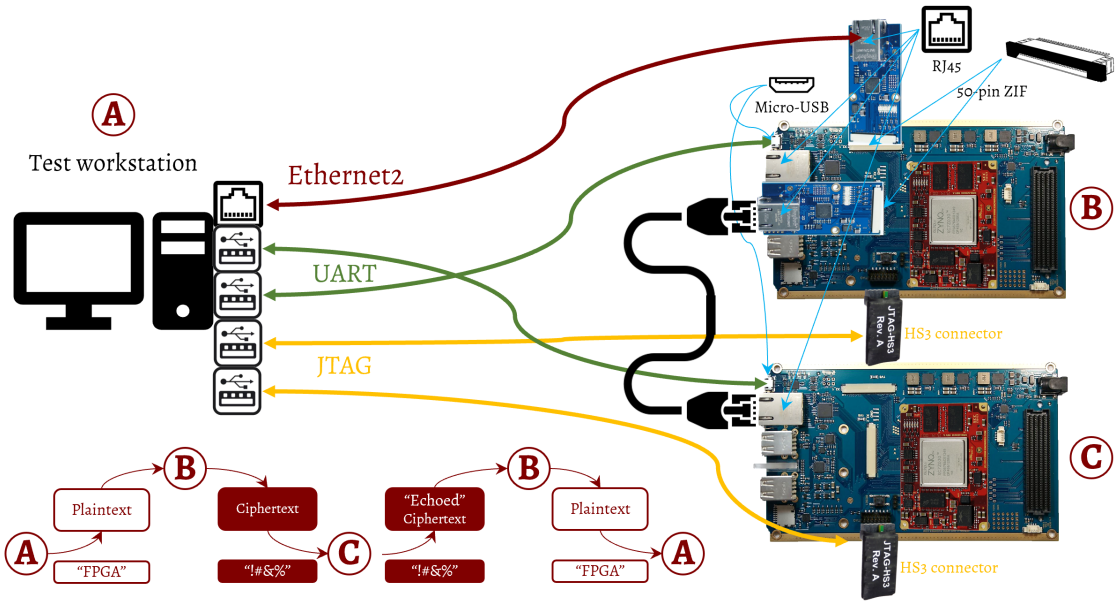

Figure 54: New setup for testing decryption capabilities

The decryption feature, instead, requires additional material to be tested. Hence, a different setup is arranged, by taking another KRC3701 kit, coming along with the KRM-3Z7030 board, as shown in figure 54. The second FPGA is configured to run the preliminary design of the Frame Repeater, allowing to send back the packets over the same line and is connected to the secondary port of the first board (B), replacing the previous connection with the workstation. In this way, the data generated by the workstation will be sent to one port of the first board (B), encrypted and pushed towards the other board (C). Here, they will be mirrored back, decrypted by the first board, and finally shown again on the primary Ethernet interface belonging to the workstation (A).

Figure 55 shows how the IPv4 packets pushed through `Ethernet2` port are correctly encrypted and accepted by `Ethernet` port (setup shown in figure 47). After changing the cable connections as in figure 54, the second test is run. Such simple and effective test puts the different designed applications together, and eventually confirms the expected results, i.e., the packets are showing up on the WIRESHARK®panel in couples: first, the original packet, followed by its looped back copy, correctly encrypted and decrypted. These results are available in figure 56.

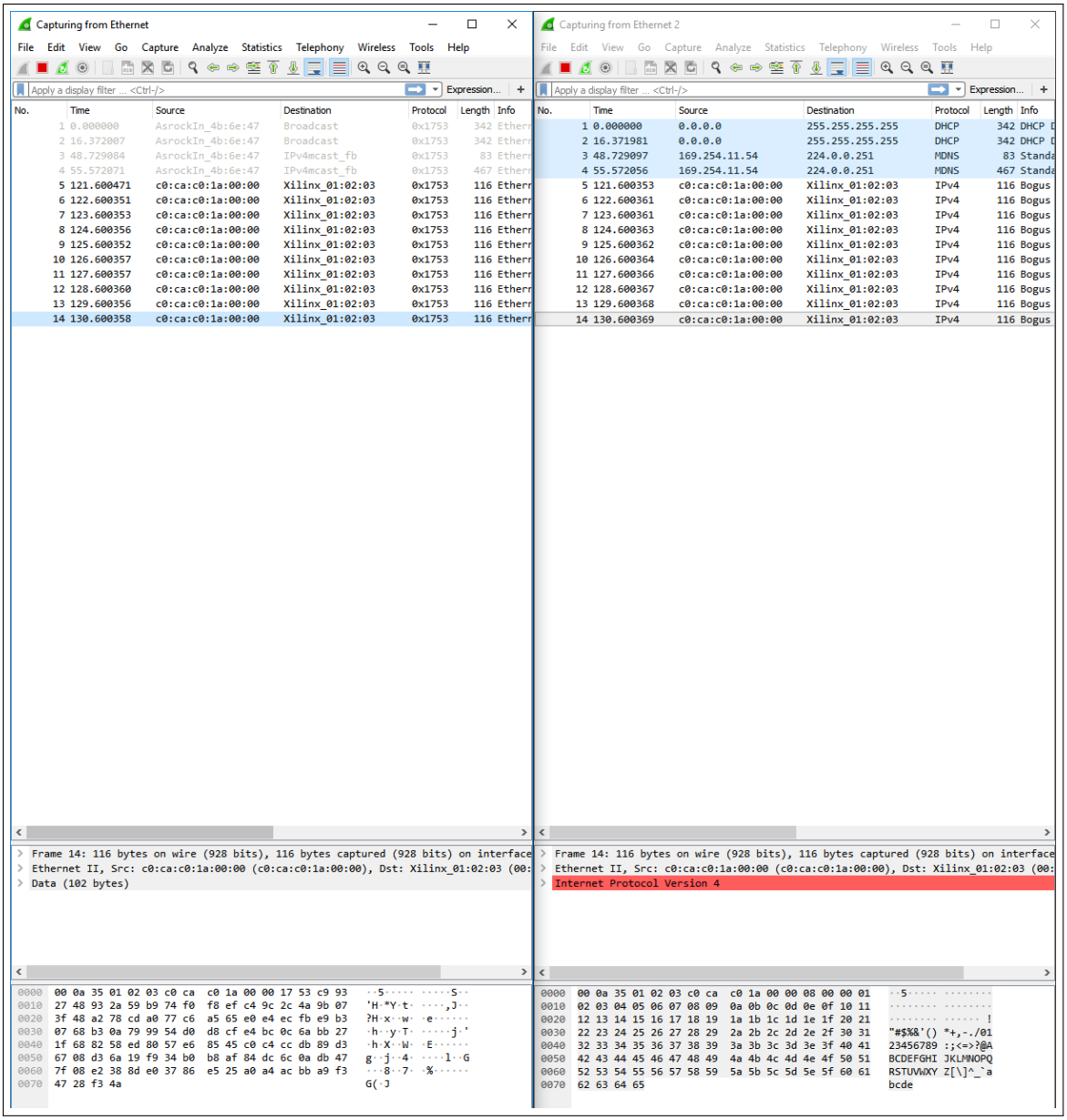

Figure 55: By enabling the encryption feature, each IPv4 payload is encrypted, together with the EtherType. On the right side there are original packets, on the left side, they are encrypted. At the bottom of this picture, the payload contents can be compared

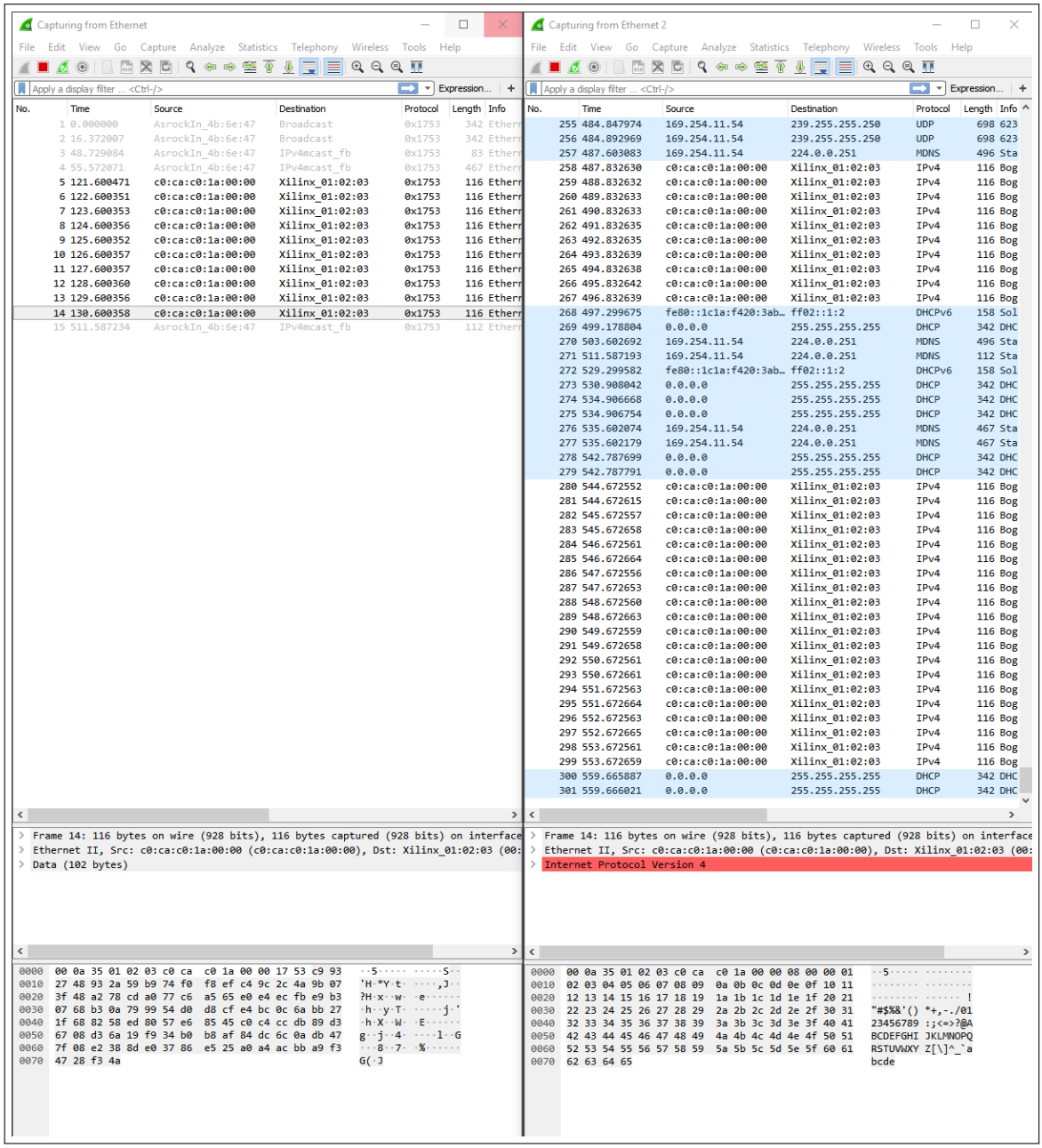

Figure 56: Following the new setup described in figure 54, the packets are sent out from `Ethernet2` port and they come back as if they were untouched. Actually they are encrypted and decrypted correctly. Please note that `Ethernet` interface is still in the screenshot but this time does not play any role, being it disconnected in this setup

Chapter 8

Future Upgrades and Conclusion

The achieved results with the last design are really satisfactory. Even though the device is not showing super robust security features, the design is still improvable, and some hints will be given in the following. In the end, the primary goal to design a real-time packet processing unit was successfully met, opening different strategies for the future.

- Reduce the datapath size from 10 to 9 bits

In this design, the data path is composed of the data interface (8 bits) and the two 1-bit control signal. During the regular operation of the device, it was observed that the error signal is rarely raised. It sometimes happens when the network is overloaded, meaning that there is a continuous stream of packets and one transmission error occurs. In this case, the FPGA knows how to drop the packet, and in general, it does not really need to buffer the error signal. Moreover, reducing the number of sampled bits from 10 to 9 significantly improves the resource management, saving possibly more resources. The FIFOs supposed to sample the network packets are nothing but BRAM cells. Such cells are optimized if the size of data to be stored matches one of the supported aspect ratios [58]. A 9-bit bus matches perfectly with 2Kbx9 or 4Kbx9 aspect ratios, whereas a 10-bit bus must comply with 1Kbx18 or 2Kbx18 and so losing half of the available space.

- Optimize the interaction PS-BRAM-DSP

The reader should remember that the final design contains two different strategies for connecting the BRAM cells to the DSP and to the PS. The second approach is indeed more

elegant from the design point of view. The following upgrade consists in understanding how to change the output properties of the port connected to the DSP by leaving the first port settings untouched, allowing still a perfect communication with the BRAM controller. One can try to output a wider set of data and adjust the search algorithm accordingly. About the search algorithm, the reader should have understood how to upgrade the *ternary search algorithm* to finish the job in just four clock cycles. By lowering the number of elements in the list to 80, such an algorithm will start the comparisons from element number 26 and 53. Then the following chunks will be checked according to the comparison results: [0-25] if it was smaller than both, [54-79] if greater than both, [27-52] otherwise. Then suppose the first block is chosen, the new elements to be checked will be 8 and 17. Therefore the new chunks: [0-7], [9-16], [18-25], and so on.

♖ Remove avoidable "abuse" of GPIO signals

That is undoubtedly one of the easiest upgrades. Due to lack of time, many signals were routed from the PS to the PL through the GPIO interface. This is not an elegant solution, because the PS can write such information on a memory cell, like a dedicated BRAM, or just can use the extra space on another BRAM cell, removing possibly the GPIO peripheral. There is nothing wrong with the presented design; nevertheless, it might look more elegant. One thing that should really change to avoid the "abuse" of this peripheral is the transmission of the evaluated key for encryption. Moreover, since the size of such key is supposed to increase, the GPIO channel will not be anymore an option.

♘ Upgrade the encryption algorithm and the key management

The first step to allow better management of the encryption system is to increase the size of the key and develop a different strategy for avoiding redundancy. One option consists in increasing the number of bytes to be encoded, from 1 to, say, 5 or 8. In this way, one can exploit powerful algorithms like the Secure Hash Algorithm (SHA), SHA-1 or SHA-256 [67] that produce respectively an output of 5 or 8 bytes. An additional improvement can be made on the symmetric function used to combine the original data with the generated

key: by choosing another function, different from XOR, or even a combination of multiple XOR operations, one can improve the reliability of the encryption system.

- Implement the same design on a smaller and less powerful device

This upgrade was already attempted in the very last days of work. The new device under development was the KRM-3Z7020 module [68], still from Knowledge Resources, with a smaller and less performing logic-fabric (Artix-7). It was quite difficult to port the design on such device and make it work properly because the high-speed clock and the implemented architecture could not find a way to be placed in the design without missing some of the timing constraints. This means that the whole architecture needs to be simplified, avoiding high fan-out on the central nodes of the design or pipelining more the signals and making the state machines work with extra time.

8.1 Conclusion

This work showed how to approach a networking problem with an unusual class of devices like FPGAs, starting from scratches. The advantages and drawbacks related to the design choices were pointed out carefully across this Thesis. Moreover, different design strategies were described to cope with the available resources on such electronic device, belonging to the non-Von Neumann class of architectures. The goals defined at the beginning of the Thesis have been met, and, although the system does not show a perfect and robust implementation of all its functions, it lays the foundation for new development, more ambitious, aspiring such perfection. The reader should have realized the enormous potential of the proposed application regarding customization, as the proposed features are just examples of packet inspection functions. Should the end user of such a product demand an additional feature, it might be implemented in the Packet Processing Unit, together with the already existing ones. Eventually, being the number of used resources very low if compared to the specifications, it is worth to spend a significant amount of time trying to port the design onto a cheaper FPGA, with much fewer resources, because

it will prove once again that processing network packets in real time is affordable with cheap electronics.

Bibliography

[1] *Protonvpn*. [Online]. Available: https://protonvpn.com/.

[2] *High performance computing, notes of class 11, ernet*. [Online]. Available: https://web.archive.org/web/20131227033204/http://hpc.serc.iisc.ernet.in/~govind/hpc/L10-Pipeline.txt.

[3] Cavium, "Introduction to ethernet latency," 2017. [Online]. Available: https://www.cavium.com/Documents/TechnologyBriefs/Adapters/Tech_Brief_Introduction_to_Ethernet_Latency.pdf.

[4] N. Wirth, *Digital Circuit Design An Introduction Textbook*. Springer, 1995.

[5] G. G. Lemieux and S. D. Brown, "A detailed router for allocating wire segments in field programmable gate arrays," *Proceedings of the ACM Physical Design Workshop*, 1993.

[6] Altera, "In the beginning," 2015. [Online]. Available: https://www.altera.com/solutions/technology/system-design/articles/_2013/in-the-beginning.html.

[7] P. Clarke, "Xilinx, asic vendors talk licensing," *EE Times*, 2001. [Online]. Available: http://www.eetimes.com/story/OEG20010622S0091.

[8] C. Maxfield, *The Design Warrior's Guide to FPGAs*, ser. The Design Warrior's Guide to FPGAs: Devices, Tools and Flows. Newnes, 2004.

[9] Xilinx, *Zynq-7000 soc data sheet - ds190 data sheet*, Jul. 2018.

[10] Xilinx, *7 series fpgas configurable logic block - ug474 user guide*, Sep. 2016.

[11] Xilinx, *7 series fpgas memory resources - ug473 user guide*, Sep. 2016.

[12] Xilinx, *7 series dsp48e1 slice - ug479 user guide*, Mar. 2018.

[13] *Krm-3z7030 product page*. [Online]. Available: http://www.knowres.ch/products/krm-3z030-768/.

[14] *Krc3701 product page*. [Online]. Available: http://www.knowres.ch/products/krc3600-carrier-kit/.

[15] *Krm-3z7030 datasheet*. [Online]. Available: http://www.knowres.ch/wp-content/uploads/KRM-3Z7xxx_DS.pdf.

[16] P. Possa, D. Schaillie, and C. Valderrama, "Fpga-based hardware acceleration: A cpu/accelerator interface exploration," *2011 18th IEEE International Conference on Electronics, Circuits, and Systems*, pp. 374–377, 2011.

[17] I. S. Association, "802.3-2012 - ieee standard for ethernet," 2012. [Online]. Available: http://standards.ieee.org/findstds/standard/802.3-2012.html.

[18] Xilinx, *Axi reference guide - ug761 user guide*, Jan. 2012.

[19] ARM, *Amba axi and ace protocol specification*, 2011. [Online]. Available: http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf.

[20] *Vivado design suite*. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html.

[21] Xilinx, *Processing system 7 - pg082 product guide*, 5.5, May 2017.

[22] Xilinx, *Vivado design suite 7 series fpga libraries guide - ug953 user guide*, Jul. 2012.

[23] Xilinx, *Vivado design suite user guide synthesis - ug901 user guide*, Jun. 2013.

[24] Xilinx, *Vivado design suite : Implementation - ug904 user guide*, Oct. 2016.

[25] O. Y. H. Cheung and P. H. W. Leong, "Implementation of an fpga based accelerator for virtual private networks," *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.*, pp. 34–41, Dec. 2002.

[26] O. Y. H. Cheung, "Implementation of an fpga based accelerator for virtual private networks," *Master of Philosophy in Computer Science and Engineering, The Chinese University of Hong Kong*, 2002.

[27] S. Mingarelli, "Streaming di immagini via ethernet con zynq con sistemi operativi standalone e linux," *Alma Mater Studiorum Università di Bologna, Tesi di laurea sperimentale in Ingegneria Informatica*, 2016.

[28] D. S. Gallego, "Desarrollo multiplataforma de un psoc basado en microprocesador arm para aplicaciones empotradas," *Escuela Técnica superior de ingenierìa de telecomunicatión. Universidad Politécnica de Cartagena*, 2014.

[29] T. An, E. Nolan, and D. Zhang, "Ethernet on the zynq zc706," *Carnegie Mellon University Pittsburgh*, 2015.

[30] *Petalinux product page*. [Online]. Available: http://www.xilinx.com/petalinux.

[31] T. Whelan, "Hardware based packet filtering using fpgas," *Bachelor of Science Honours in Computer Science of Rhodes University*, 2010.

[32] P. Födisch, B. Lange, J. Sandmann, A. Büchner, W. Enghardt, and P. Kaever, "A synchronous gigabit ethernet protocol stack for high-throughput udp/ip applications," 2015.

[33] B. V. del Pino, P. P. Carballo, and A. Nuñez, "Design and implementation of a tcp/ip packet filter and classifier ip block through high level synthesis," *IUMA, Institute for Applied Microelectronics, University of Las Palmas Gran Canaria, Spain*,

[34] S. Ilčev, *Global Satellite Meteorological Observation (GSMO) Theory*, v. 1. Springer International Publishing, 2017.

[35] Cisco, "Ethernet technologies. internetworking technology handbook," [Online]. Available: http://docwiki.cisco.com/wiki/Ethernet_Technologies.

[36] "Ieee standard for ethernet," *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)*, pp. 1–4017, Mar. 2016. DOI: 10.1109/IEEESTD.2016.7428776.

[37] D. Pannell, "Mac address issues in ieee 802.1," [Online]. Available: http://www.ieee802.org/1/files/public/docs2014/New-pannell-MAC-Address-Issues-in-802dot1-1114-v1.pdf.

[38] "Ieee standard for local and metropolitan area networks," *IEEE Std 802.3-1997*, 1997. [Online]. Available: https://standards.ieee.org/standard/802_3x-1997.html.

[39] Marvell, *Marvell alaska 88e1116r: Single-port gigabit ethernet transceiver with integrated passives*, 2007. [Online]. Available: https://www.marvell.com/documents/nrdtpakoxnfkopzrimwn/.

[40] *Kr-lan-a1 product page*. [Online]. Available: http://www.knowres.ch/products/kr-lan-a1/.

[41] Ostinato.org, *Network traffic generator and analyzer*. [Online]. Available: https://ostinato.org/downloads.

[42] */lib/sw_apps/lwip_echo_server/src/echo.c, belonging to LWIP library*. [Online]. Available: https://github.com/Xilinx/embeddedsw/.

[43] */sw_services/lwip141/src/lwip-1.4.1/src/core/memp.c, belonging to LWIP library*. [Online]. Available: https://github.com/Xilinx/embeddedsw/blob/master/ThirdParty/.

[44] J. G. Urgellés, *Matematici, spie e pirati informatici*. RBA Italia, 2013.

[45] L. Sacco, *Manuale di Crittografia*. Roma, 1947.

[46] *LWIP library*. [Online]. Available: https://github.com/Xilinx/embeddedsw/tree/master/ThirdParty/sw_services/lwip141/src/lwip-1.4.1.

[47] Xilinx, *Zynq-7000 soc - ug585 user guide*, Jul. 2018.

[48] I. Freire, "Understanding the gigabit ethernet controller's dma on zynq devices," 2016.

[49] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, Chapter 10. Interrupt Handling*, 3rd ed. O'Reilly Media, 2005.

[50] *XEmacPs driver, bdring.c source*. [Online]. Available: https://github.com/Xilinx/embeddedsw/blob/master/XilinxProcessorIPLib/drivers/emacps/src/xemacps_bdring.c.

[51] Xilinx, *Gmii to rgmii v4.0 - pg160 product guide*, Jun. 2018.

[52] D. Behera, K. Rao, and D. Mahajan, "Understanding the basics of setup and hold time," *EDN*, 2012. [Online]. Available: https://www.edn.com/design/analog/4371393/Understanding-the-basics-of-setup-and-hold-time.

[53] IEEE, "802.3 part 3: Carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications, 2005," [Online]. Available: http://www.ieee802.org/3/interp/interp-1-1109.pdf.

[54] Xilinx, *7 series fpgas clocking resources - ug472 user guide*, Jul. 2018.

[55] Xilinx, *Mmcm product description*. [Online]. Available: https://www.xilinx.com/products/intellectual-property/mmcm_module.html.

[56] D. Knuth, *The Art of Computer Programming: Sorting and searching*, 2nd ed. Reading, Massachusetts: Addison-Wesley, 1998, vol. 3.

[57] Xilinx, *Kintex-7 fpgas data sheet: Dc and ac switching characteristics - ds182 data sheet*, Aug. 2018.

[58] Xilinx, *Block memory generator v8.3 - pg058 product guide*, Apr. 2017.

[59] J. Postel, *Internet control message protocol*. DOI: 10.17487/RFC0792.

[60] Forouzan and B. A., *Data Communications And Networking*. Boston:McGraw-Hill, 2007.

[61] C. Borrelli, *Ieee 802.3 cyclic redundancy check - xapp209 application note*, Mar. 2001. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp209.pdf.

[62] OutputLogic.com. [Online]. Available: http://outputlogic.com/?page_id=321.

[63] Fowler, Noll, and Vo, *Fnv hash*. [Online]. Available: http://www.isthe.com/chongo/tech/comp/fnv/index.html.

[64] Xilinx, *Clocking wizard - pg065 product guide*, Oct. 2016.

[65] C. Anthony and R. Hoare, *Quicksort: Algorithm 64*. ACM, 1961.

[66] Xilinx, *Vivado design suite user guide implementation - ug904 user guide*, Dec. 2013.

[67] H. Gilbert and H. Handschuh, "Security analysis of sha-256 and sisters," *Selected Areas in Cryptography*, 2003. [Online]. Available: http://standards.ieee.org/findstds/standard/802.3-2012.html.

[68] *Krm-3z7020 product page*. [Online]. Available: http://www.knowres.ch/products/krm-3z20-512-a/.