Czech Technical University
Faculty of Electrical Engineering
Department of Measurement

Test and data acquisition module for Automotive Ethernet networks

bachelor thesis

created by
Tomáš Veselý

supervised by
doc. Ing. Jiří Novák, Ph.D.

Open Informatics
Internet of Things

24 May 2024

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Veselý Tomáš** Personal ID number: **507337**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Measurement**

Study program: **Open Informatics**

Specialisation: **Internet of Things**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Test and DAQ module for Automotive Ethernet networks

Bachelor’s thesis title in Czech:

Modul pro testování a sběr dat v sítích Automotive Ethernet

Guidelines:

Learn the Ethernet standards, specifically Automotive Ethernet, the PTP protocol and the ASAM CMP standard for data acquisition.
Design and implement basic bridge functionality in the AMD Kria module with support for filtering of transmitted frames and mirroring communication on the module's DAQ port.
Implement PTP protocol support on Automotive Ethernet links and verify module transparency in real vehicle network.
Try to implement at least minimal (HW accelerated) CMP protocol support on the module's DAQ interface.

Bibliography / sources:

1. IEEE Standard for Ethernet. IEEE Std 802.3-2022. LAN/MAN Standards Committee of the IEEE Computer Society. 2022.
2. IEEE Standard for Local and metropolitan area networks — Bridges and Bridged Networks. IEEE Std 802.1Q-2022. LAN/MAN Standards Committee of the IEEE Computer Society. 2022.
3. IEEE Standard for Local and Metropolitan Area Networks — Timing and Synchronization for Time-Sensitive Applications. IEEE Std 802.1AS-2020. LAN/MAN Standards Committee of the IEEE Computer Society. 2020.
4. Kria K26 SOM Data Sheet. DS987. v1.4. Advanced Micro Devices, Inc. July 2023.

Name and workplace of bachelor’s thesis supervisor:

doc. Ing. Jiří Novák, Ph.D. Department of Measurement FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **09.02.2024** Deadline for bachelor thesis submission: _____

Assignment valid until:
by the end of summer semester 2024/2025

doc. Ing. Jiří Novák, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____ Date of assignment receipt

_____ Student's signature

Abstract

Automotive industry is transitioning from slow communication over CAN bus to fast communication over Automotive Ethernet needed by cameras, radars, lidars and similar sensors. Manufacturers need a device for complex cyber security testing of new vehicles which could capture and modify frames on multiple Automotive Ethernet links at the same time. The goal of my bachelor thesis is to create such a device.

There are three main requirements for this device. First, it should forward Ethernet frames between pairs of Automotive Ethernet ports (either 100BASE-T1 for 100 Mb/s Ethernet or 1000BASE-T1 for 1 Gb/s Ethernet). Second, it should log the traffic via the CMP protocol to the DAQ port (implemented using SFP+ module for up to 10 Gb/s Ethernet). Third, it should support time synchronization via the PTP protocol on the Automotive Ethernet ports.

The AMD Kria platform built on the Zynq UltraScale+ system on chip has been chosen for implementation of the device. For better performance, forwarding and logging will be implemented in hardware using the programmable logic. On the other hand, time synchronization and configuration will be implemented in software using the processing system.

My thesis describes the whole design process starting with learning the relevant technologies and ending with testing the device in a real environment. In the first part, the hardware design is discussed, verified in simulation and implemented on the target platform. In the second part, the software design is discussed and built for the target platform. Finally, both parts are validated together in more complex tests.

Abstrakt

Automobilový průmysl postupně přechází z pomalé komunikace přes sběrnici CAN na rychlou komunikaci přes Automotive Ethernet, kterou potřebují kamery, radary, lidary a podobné senzory. Výrobci potřebují mít zařízení pro komplexní testování kybernetické bezpečnosti nových vozidel, které by dokázalo zachytávat a upravovat rámce na několika linkách Automotive Ethernet zároveň. Cílem mojí bakalářské práce je vytvoření takového zařízení.

Na zařízení máme tři hlavní požadavky. Zaprvé by mělo přeposílat Ethernetové rámce mezi dvojicemi portů pro Automotive Ethernet (100BASE-T1 pro 100 Mb/s Ethernet nebo 1000BASE-T1 pro 1 Gb/s Ethernet). Zadruhé by mělo zaznamenávat provoz skrz CMP protokol na portu pro sběr dat (realizovaném pomocí SFP+ modulu pro až 10 Gb/s Ethernet). Zatřetí by mělo podporovat synchronizaci času skrz PTP protokol na portech pro Automotive Ethernet.

Pro implementaci zařízení byla zvolena platforma AMD Kria postavená na systému na čipu Zynq UltraScale+. Pro dosažení co nejlepšího výkonu bude přeposílání a zaznamenávání rámců realizováno v hardware pomocí programovatelné logiky. Naproti tomu synchronizace času a konfigurace zařízení bude realizována v software pomocí procesorového systému.

Moje práce popisuje celý proces návrhu počínaje studiem příslušných technologií a konče testováním zařízení v reálném prostředí. V první části se diskutuje návrh hardware, jeho verifikace v simulaci a implementace pro cílovou platformu. V druhé části se diskutuje návrh software a jeho kompilace pro cílovou platformu. Nakonec jsou obě části validovány dohromady pomocí komplexnějších testů.

Acknowledgement

My thesis would not have been possible without the endless help and support from my supervisor. In particular, I am thankful to him for:

- creating such an amazing and interesting project I could work on and I could learn on
- providing and lending me all the resources that I needed for completing the project
- spending numerous hours discussing all possible solutions which I could think of
- helping out with final testing of the device in real vehicle network
- reading the thesis before hand off and correcting what I missed

Thanks should also go to my family for their support not only during my work on the thesis but also during my study at the university.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

Tomáš Veselý
24 May 2024
Prague

Contents

Chapter 1

Introduction

This chapter provides more information about the goals of this project and the reasons leading to this project. It is divided into the following sections:

- In section 1.1, the reasons which led to working on this project are described.
- In section 1.2, the goal of bridging the Ethernet ports with filtering capabilities is described.
- In section 1.3, the goal of using the PTP protocol for time synchronization is described.
- In section 1.4, the goal of using the CMP protocol for data logging is described.

1.1 The motivation

In modern vehicles, there is nearly one hundred of independent electronic control units (ECUs) which need to communicate together. For a long time, the CAN and LIN buses have been used for connecting the units. For high speed communication, such as transporting image from cameras or data from radars, those buses are too slow though. That is when automotive Ethernet started to appear in vehicle networks.

Just like any other computer network, the automotive Ethernet networks might potentially become an attack vector. Every device connected to that network should be tested thoroughly to discover potential vulnerabilities before the device is used in production vehicles. And that is where our device comes into the game.

There are commercial devices which allow tapping into automotive Ethernet links and logging the frames for debugging purposes. These already allow passive attacks which require only capturing the traffic. However, they usually do not allow interactive modification of the traffic. That is required for active attacks which introduce malicious frames into the network. Therefore, our device should be designed not only to capture the traffic, but also modify it when needed.

1.2 Bridging the Ethernet ports

The main goal is to create a bridge between a pair of Ethernet ports. Our device would ideally have multiple identical pairs but for simplicity, only one pair of ports will be considered in this section.

In the first stage, which we could call the **passive mode**, our device should be fully transparent. That means frames received on port 1 should be transmitted on port 2 without any modifications and frames received on port 2 should be transmitted on port 1 without any modifications. The device could then be called a simple repeater.

In the second stage, which we could call the **active mode**, our device would be transparent only for some frames. Those frames would still be handled by hardware to support full

bandwidth. The rest of the frames would be processed by software or dropped according to the current configuration. The software could modify the frames before passing them on, but also introduce new frames to the network.

In order to accomplish this goal, the frames will need to pass through the following components which will need to be designed:

- The frames will be received from one of the Ethernet ports so I will design a **MAC receiver** for that purpose. It is described in section 3.6.
- The frames will be split into two streams. The first stream will be passed by hardware without any modification. The second stream will be processed by software. For this purpose, I will design a **splitter** as described in section 3.7.
- The frames of the second stream will be transported from hardware to memory where they become available to software. Once they are processed, they will be transported back from memory to hardware. For this purpose, I will design a **DMA receiver** and **DMA transmitter** as described in section 3.8.
- The two streams will need to be joined back together. The first stream is the one passed by hardware without any modification. The second stream is the one coming from software with modifications. For this purpose, I will design a **joiner** as described in section 3.9.
- The frames will be transmitted to the other Ethernet port so I will design a **MAC transmitter** for that purpose. It is described in section 3.6.

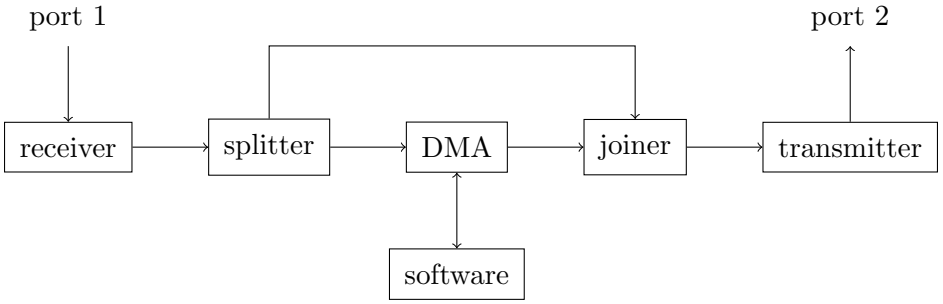

Figure 1.1: Bridging the Ethernet ports

1.3 Supporting the PTP protocol

Another goal is to support the Precision Time Protocol (PTP). It is specified in the IEEE 802.1AS standard [6]. The electronic control units use it to synchronize time with precision better than one microsecond.

The protocol uses hardware timestamping of both received and transmitted frames to synchronize time in two steps. First, the delay of the link is measured. Second, the time is synchronized with the delay accounted.

With our device working in the **passive mode** as described in the previous section, the synchronization would probably still work well. The delay introduced by our device would be constant so it would be added to the delay of the link and accounted for when synchronizing the time. There could be a problem if the total delay was too large because the specification says it should not exceed 800 nanoseconds. However, it should be possible to keep the total delay under that limit.

With our device working in the **active mode** as described in the previous section, the synchronization could easily become unstable. The delay introduced by our device would no longer be constant. The possiblity to modify the frames or even introduce new frames would mean that the frames could be transmitted in a different order or with a different timing than they were received. For that reason, the delay of the link could not be reliably measured.

The solution to that problem is to participate actively in the time synchronization. The specification allows creating PTP Relay devices which propagate time synchronization from one port to another. The delay between our device and the other unit will then be measured instead of the delay between the units. Our device will then use it to modify the correction field of the synchronization messages to account for the time of residence in our device.

Although it would be possible to implement this in hardware, it is much simpler to use software to accomplish this goal. Moreover, it does not require any modifications to the structure of the device proposed in the previous section. The only addition is that timestamping must be supported in hardware. Each frame received or transmitted will be timestamped and the timestamp will be made available to software.

1.4 Supporting the CMP protocol

Another goal is to support the Capture Module Protocol (CMP). Is is specified in the ASAM CMP standard [3]. The commercial devices similar to our device use it to log frames received and transmitted on the network.

The reason for using this protocol is that simply passing the frames from gigabit Ethernet ports to the ten gigabit Ethernet port would not allow any metadata added to the frames. Especially, it would be impossible to know which port the frame was received or transmitted on in case our device had more than one port. Moreover, it would appear that frames were going through the network sequentially while some of them could be actually received and transmitted at the same time. The CMP protocol includes fields for both the interface identificator and the timestamp and even more fields for other kinds of metadata.

Another reason for preferring this protocol to custom protocol is that it is already used by some devices and it is expected that more devices could start using it in the near future. That allows using the same software with multiple devices of different vendors including our device. Once that software is developed, it will be possible to use it with our device. For example, the CMP protocol is already supported in Wireshark, which will definitely simplify the development.

Although it would be simpler to implement this in software, it would significantly limit the performance of our device. For that reason, it was decided that frames should be packaged into the CMP protocol by hardware. It is the only way to fully use the potential of the ten gigabit Ethernet port.

Chapter 2

Learning the Technology

This chapter summarizes the basic information about the existing technology relevant to this project. It is divided into the following sections:

- Section 2.1 is about the Ethernet standard which is used to connect not only personal computers but also automotive control units.
- Section 2.2 is about the AMD Kria ecosystem which will be used as the target platform in this project.
- Section 2.3 is about the PTP protocol which is used to synchronize the time in automotive networks.
- Section 2.4 is about the CMP protocol which will be used to transport the captured frames in this project.

2.1 Ethernet

Ethernet is a communication technology standardized by the IEEE 802.3 standard [6]. It covers the first and the second layers of the ISO/OSI model.

2.1.1 Physical layer

The physical layer is the first layer according to the ISO/OSI model. Its main function is to receive and transmit bits on the physical medium. In case of Ethernet, the physical medium can be a copper cable composed of one or more twisted pairs, an optical cable composed of optical fibers or a coaxial cable.

The physical layer of Ethernet is implemented with a medium specific transceiver often abbreviated as PHY. On one side, the PHY is attached to the physical medium using a media dependent interface abbreviated as MDI. On the other side, the PHY is connected to the MAC which handles the data link layer using a media independent interface abbreviated as MII.

Thanks to that, any PHY and any MAC can be paired together if they use the same interface. For example, the same 1 Gb/s MAC using GMII can be used both with a 1000BASE-T PHY for Ethernet and with a 1000BASE-T1 PHY for automotive Ethernet.

The physical layer of Ethernet is divided into the following sublayers:

- The **PCS** (Physical Coding Sublayer) is the sublayer between the Media Independent Interface (MII) and the Physical Medium Attachment (PMA).
  - On transmission, it encodes the data bits to the code groups transmitted by the PMA.
  - On reception, it decodes the data bits from the code groups received by the PMA.

- The **PMA** (Physical Medium Attachment) is the sublayer responsible for transmission, reception, collision detection, clock recovery and skew alignment.
- The **PMD** (Physical Medium Dependent) is the sublayer between the Physical Medium Attachment (PMA) and the Media Dependent Interface (MDI).

The physical layer of Ethernet has many variants depending on the physical medium and the data rate. The variants relevant to our project are listed below.

100BASE-TX

The 100BASE-TX physical layer provides data rate of 100 Mb/s over two pairs of twisted pair cable. It is defined in clause 25 of the IEEE 802.3 standard [6].

100BASE-FX

The 100BASE-FX physical layer provides data rate of 100 Mb/s over two optical fibers. It is defined in clause 26 of the IEEE 802.3 standard [6].

1000BASE-T

The 1000BASE-T physical layer provides data rate of 1 Gb/s over four pairs of balanced twisted pair cable. It is defined in clause 40 of the IEEE 802.3 standard [6].

10GBASE-T

The 10GBASE-T physical layer provides data rate of 10 Gb/s over four pairs of balanced twisted pair cable. It is defined in clause 55 of the IEEE 802.3 standard [6].

100BASE-T1

The 100BASE-T1 physical layer provides data rate of 100 Mb/s over a single balanced twisted pair copper cable. Is is defined in clause 96 of the IEEE 802.3 standard [6].

1000BASE-T1

The 1000BASE-T1 physical layer provides data rate of 1 Gb/s over a single balanced twisted pair copper cable. It is defined in clause 97 of the IEEE 802.3 standard [6].

10GBASE-R

The 10GBASE-R physical layer provides data rate of 10 Gb/s. It is defined in clause 49 of the IEEE 802.3 standard [6].

2.1.2 Data link layer

The data link layer is the second layer according to the ISO/OSI model. Its main function is to receive and transmit frames via the physical layer. In case of Ethernet, it also allows addressing the frame to one or more specific destinations and checking the integrity of the frame.

Ethernet frame

All possible variants of Ethernet use the same frame format which allows for simple interfacing between different variants. It is described in clause 3 of the IEEE 802.3 standard [6].

The Ethernet frame is composed of the following fields:

- The **Preamble** consists of 7 bytes with alternating ones and zeros (10101010). It was used to achieve bit synchronization in early versions of Ethernet.
- The **Start Frame Delimiter** (SFD) consists of 1 byte with alternating ones and zeros ending with two ones (10101011). It was used to achieve byte synchronization in early versions of Ethernet. The following byte marks the start of the frame.
- The **Destination Address** consists of 6 bytes and tells the hardware address of the device to which the frame was sent.
- The **Source Address** consists of 6 bytes and tells the hardware address of the device which sent the frame.
- The **Length/Type** consists of 2 bytes and may have two different meanings depending on the value:
  - If the value is less than or equal to 1500, this field indicates the **Length** of the data contained in the frame. This option was used historically.
  - If the value is more than or equal to 1536, this field indicates the **Type** of the data contained in the frame. Nowadays, this option is used almost exclusively. The values relevant for this project are shown in table 2.1.
- The **Data** consists of at least 46 bytes and at most 1500 bytes. If the original data is shorter, a padding of zeros is appended to meet the minimal length. If the original data is longer, it must be split into multiple frames by the upper layer protocol to meet the maximal length.
- The **Frame Check Sequence** (FCS) consists of 4 bytes and allows checking the integrity of the frame.

Value	Protocol
0x0800	IPv4 (Internet Protocol version 4)
0x0806	ARP (Address Resolution Protocol)
0x86DD	IPv6 (Internet Protocol version 6)
0x88F7	PTP (Precision Time Protocol)
0x99FE	CMP (Capture Module Protocol)

Table 2.1: Common EtherType values

Ethernet address

The Ethernet address consists of 6 bytes where usually the first 3 bytes are assigned to one vendor by a central authority and the last 3 bytes are assigned by the vendor to individual devices.

The first bit determines whether the address is of Individual or Group type. The destination address may be of either Individual or Group type while the source address must be of Individual type.

Frame check sequence

The frame check sequence provides possibility to detect whether the frame got corrupt during transmission. It uses a slightly modified version of the CRC-32 algorithm with the following polynomial:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{1}1 + x^{1}0 + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$$

The following procudure taken from section 3.2.9 of the IEEE 802.3 standard [6] is used to generate the frame check sequence:

1. The first 32 bits of the frame are complemented.
2. The n bits of the protected fields are then considered to be the coefficients of a polynomial $M(x)$ of degree $n - 1$.
   - The first bit of the Destination Address field corresponds to the x^{n-1} term.
   - The last bit of the Data field (or Pad field if present) corresponds to the x^0 term.
3. $M(x)$ is multiplied by x^{32} and divided by $G(x)$, producing a remainder $R(x)$ of degree ≤ 31.
4. The coefficients of $R(x)$ are considered to be a 32-bit sequence.
5. The bit sequence is complemented and the result is the CRC.

Media independent interface

The media independent interface serves for the communication between the PHY which handles the physical layer of Ethernet and the MAC which handles the data link layer of Ethernet. Apart from the original definitions of the interfaces in the IEEE 802.3 standard [6], there are other variants introduced by the manufacturers of the transceivers with reduced pin count.

For 10 Mb/s and 100 Mb/s Ethernet, the following interfaces are used:

- The **MII** which uses clock signal, 2 control signals and 4 data signals for transmission and the same set of signals for reception. It is defined in clause 22 of the standard.
- The **RMII** (Reduced MII) which joins pairs of MII signals in order the reduce the number of signals.

For 1 Gb/s Ethernet, the following interfaces are used:

- The **GMII** (Gigabit MII) which uses clock signal, 2 control signals and 8 data signals for transmission and the same set of signals for reception. It is defined in clause 35 of the standard and decribed in section 2.1.3.
- The **RGMII** (Reduced GMII) which joins pairs of GMII signals in order to reduce the number of signals. It is described in section 2.1.4.
- The **SGMII** (Serial GMII) which encodes GMII signals into serial data stream using the 8b/10b encoding. It is described in section 2.1.5.

For 10 Gb/s Ethernet, the following interfaces are used:

- The **XGMII** (10 Gigabit MII) which uses clock signal, 4 control signals and 32 data signals for transmission and the same set of signals for reception. It is defined in clause 46 of the standard and described in section 2.1.6.
- The **SXGMII** (Serial XGMII) which encodes XGMII signals into serial data stream using the 64b/66b encoding. It is described in section 2.1.7.

2.1.3 Gigabit MII

The GMII (Gigabit Media Independent Interface) allows connecting any compatible Gigabit Ethernet PHY to any compatible Gigabit Ethernet MAC. Thanks to this standard, physical layer and link layer may be developed separately and even split between two physical chips.

The GMII is defined in clause 35 of the IEEE 802.3 standard [6]. The following subsections provide basic information about this interface.

TX signals

The GMII requires the following signals for transmission:

- **TX_CLK** (transmit clock) provides 125 MHz clock for synchronizing all other signals.
- **TX_EN** (transmit enable) controls transmission of data. When high, data is considered valid and transmitted by the PHY. When low, data is ignored. Transition from low to high marks start of packet (and should be followed by preamble). Transition from high to low marks end of packet (and should come after frame check sequence).
- **TX_ER** (transmit error) provides possibility to corrupt data and propagate reception error in this way. When high, invalid data is intentionally transmitted by the PHY. When low, valid data is normally transmitted by the PHY.
- **TXD** (transmit data) consists of eight data signals and transmits one byte (eight bits) at time.

The direction of all those signals is from the MAC to the PHY. The following table summarizes all possible values of the signals and their meaning:

TX_EN	TX_ER	TXD	Description
0	0	X	Inter-frame gap
1	0	X	Normal data trnasmission
1	1	X	Transmit error propagation
0	1	0x01	Assert LPI
0	1	0x0F	Carrier Extend
0	1	0x1F	Carrier Extend Error

Table 2.2: Meaning of GMII TX signals

Other combinations of signals are reserved according to the specification.

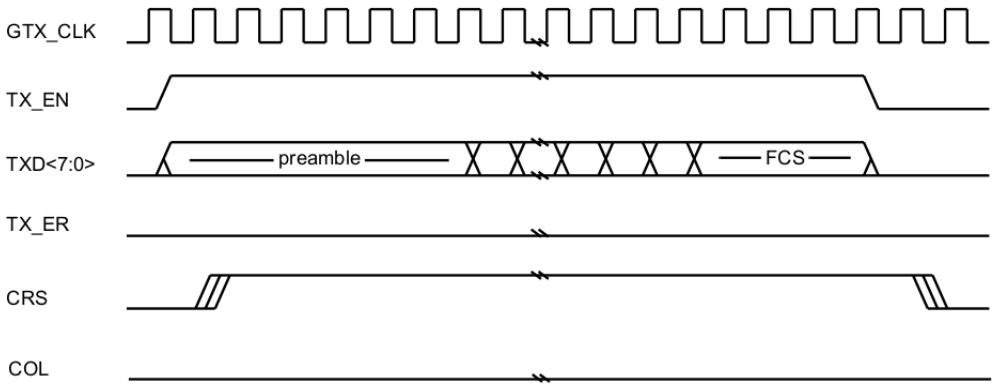

Figure 2.1: Timing diagram of GMII frame transmission [6]

RX signals

The GMII requires the following signals for reception:

- **RX_CLK** (receive clock) provides 125 MHz clock for synchronizing all other signals. This clock is usually recovered by the PHY from the incoming signal.
- **RX_DV** (receive data valid) indicates that received data is valid. When high, data is considered valid and processed by the MAC. When low, data is ignored. Transition from low to high marks start of packet (and should be followed by preamble). Transition from high to low marks end of packet (and should come after frame check sequence).
- **RX_ER** (receive error) reports to the MAC that an error occurred during the frame reception. When high, the current frame should be discarded by the MAC. When low, the current frame is being received correctly.
- **RXD** (receive data) consists of eight data signals and transmits one byte (eight bits) at time.

The direction of all those signals is from the PHY to the MAC. The following table summarizes all possible values of the signals and their meaning:

RX_DV	RX_ER	RXD	Description
0	0	X	Inter-frame gap
1	0	X	Normal data reception
1	1	X	Data reception error
0	1	0x01	Assert LPI
0	1	0x0E	False Carrier
0	1	0x0F	Carrier Extend
0	1	0x1F	Carrier Extend Error

Table 2.3: Meaning of GMII RX signals

Other combinations of signals are reserved according to the specification.

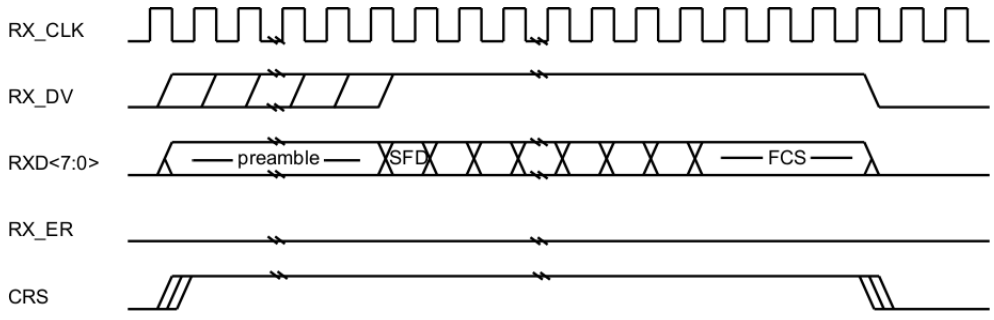

Figure 2.2: Timing diagram of GMII frame reception [6]

Additional signals

The following additional signals are used in the half duplex mode:

- **CRS** (carrier sense) indicates that either the transmit or receive medium is non-idle.
- **COL** (collision) indicates that collision has been detected on the medium.

Management signals

The management interface is described in section 2.1.8.

2.1.4 Reduced Gigabit MII

The RGMII stands for Reduced Gigabit Media Independent Interface and it reduces the number of signals required for interfacing between gigabit Ethernet PHY and MAC. The original GMII requires 22 signals in total - 8 data signals, 2 control signals and clock for reception and the same set of signals for transmission. The reduced GMII requires 12 signals in total - 4 data signals, control signal and clock for reception and the same set of signals for transmission.

The reduction is achieved by switching from SDR (single data rate) signaling to DDR (double data rate) signaling. That means instead of using only rising edge of the clock for data transmission, both rising and falling edges are used. The clock frequency (125 MHz) is the same for original GMII and reduced GMII.

The relationship between GMII and RGMII signals is shown in figures 2.3 and 2.4. The mapping is also summarized in table 2.4.

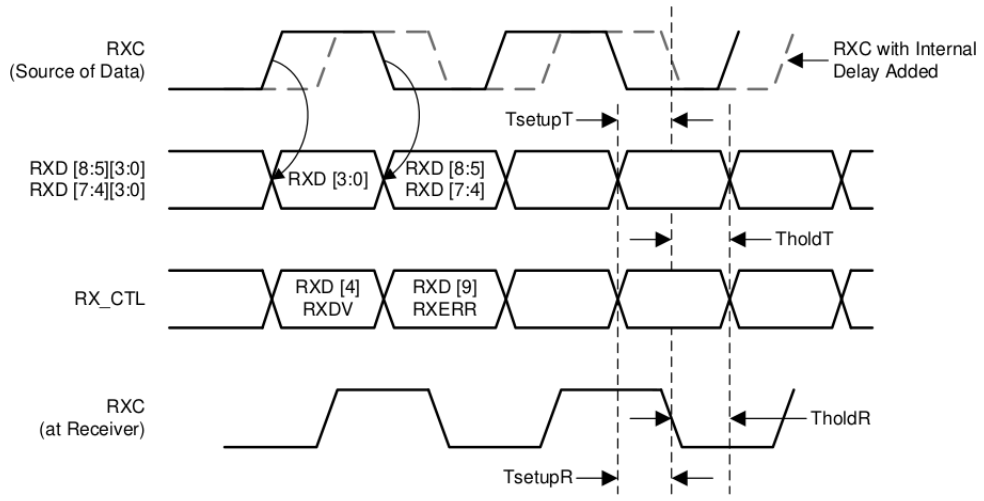

Figure 2.3: Timing diagram of RGMII reception [4]

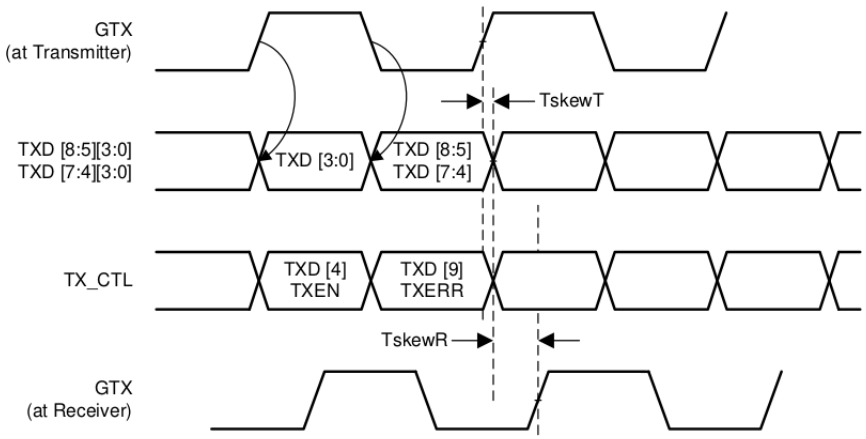

Figure 2.4: Timing diagram of RGMII transmission [4]

RGMII signal	Rising edge	Falling edge
TXD[3:0]	TXD[3:0]	TXD[7:4]
TX_CTL	TX_EN	TX_ER
RXD[3:0]	RXD[3:0]	RXD[7:4]
RX_CTL	RX_DV	RX_ER

Table 2.4: Mapping between GMII and RGMII signals

2.1.5 Serial Gigabit MII

The SGMII stands for Serial Gigabit Media Independent Interface and it allows interfacing between gigabit Ethernet PHY and MAC using purely serial communication. It requires only one differential pair for reception and one differential pair for transmission compared to 22 signals in total for original GMII. Thanks to the 8b/10b encoding, the clock can be recovered from the data and does not have to be passed alongside.

2.1.6 Ten Gigabit MII

The XGMII (10 Gigabit Media Independent Interface) allows connecting any compatible 10 Gigabit Ethernet PHY to any compatible 10 Gigabit Ethernet MAC.

The XGMII is defined in clause 46 of the IEEE 802.3 standard [6]. The following subsections provide basic information about this interface.

TX signals

The XGMII requires the following signals for transmission:

- **TX_CLK** (transmit clock) provides 156.25 MHz clock for synchronizing all other signals.
- **TXC** (transmit control) consists of 4 control signals (1 for each lane) and determines the meaning of each lane. When high, the corresponding lane contains a control character. When low, the corresponding lane contains a data byte.
- **TXD** (transmit data) consists of 32 data signals (8 for each lane) and is divided into 4 lanes. Each lane contains either one data byte or one control character, depending on the corresponding control signal.

The direction of all those signals is from the MAC to the PHY. The following table summarizes all possible values of the signals for each lane and their meaning:

TXC	TXD	Description
0	X	Normal data transmission
1	0x06	Request LPI
1	0x07	Idle
1	0x9C	Sequence
1	0xFB	Start
1	0xFD	Terminate
1	0xFE	Transmit error propagation

Table 2.5: Meaning of XGMII TX signals

Other combinations of signals are reserved according to the specification.

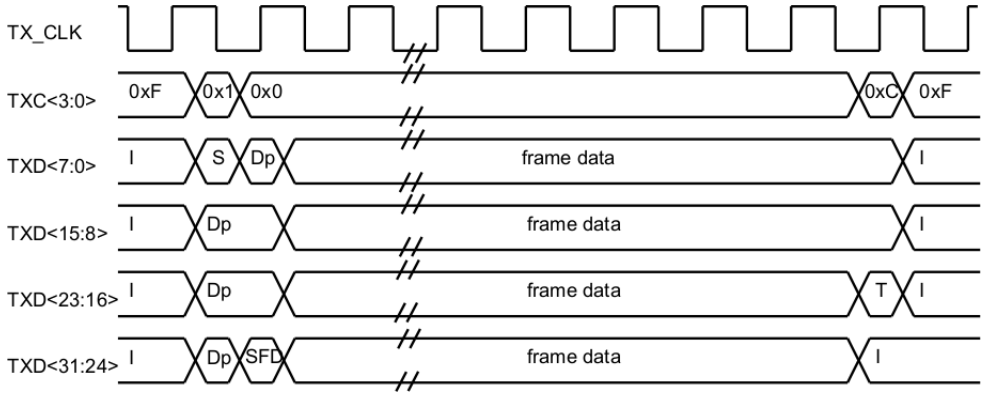

Figure 2.5: Timing diagram of XGMII frame transmission [6]

RX signals

The XGMII requires the following signals for reception:

- **RX_CLK** (receive clock) provides 156.25 MHz clock for synchronizing all other signals. This clock is usually recovered by the PHY from the incoming signal.
- **RXC** (receive control) consists of 4 control signals (1 for each lane) and determines the meaning of each lane. When high, the corresponding lane contains a control character. When low, the corresponding lane contains a data byte.
- **RXD** (receive data) consists of 32 data signals (8 for each lane) and is divided into 4 lanes. Each lane contains either one data byte or one control character, depending on the corresponding control signal.

The direction of all those signals is from the PHY to the MAC. The following table summarizes all possible values of the signals for each lane and their meaning:

RXC	RXD	Description
0	X	Normal data reception
1	0x07	Idle
1	0x9C	Sequence
1	0xFB	Start
1	0xFD	Terminate
1	0xFE	Receive error

Table 2.6: Meaning of XGMII RX signals

Other combinations of signals are reserved according to the specification.

Management signals

The management interface is described in section 2.1.8.

2.1.7 Serial Ten Gigabit MII

The (U)SXGMII stands for (Universal) Serial Ten Gigabit Media Independent interface and it allows interfacing between ten gigabit Ethernet PHY and MAC using purely serial communication. It requires only one differential pair for reception and one differential pair for transmission compared to 74 signals in total for original XGMII. Thanks to the 64b/66b encoding, the clock can be recovered from the data and does not have to be passed alongside.

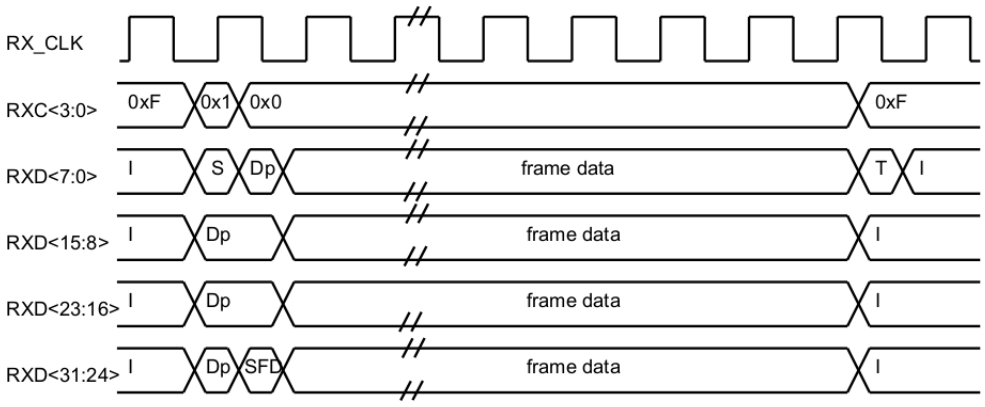

Figure 2.6: Timing diagram of XGMII frame reception [6]

2.1.8 Management MII

Apart from the data interface described in previous sections, the Ethernet transceiver usually has the control interface for management of the PHY by the host. This interface does not have any specific name but it is used both in conjunction with MII for 10 Mb/s and 100 Mb/s Ethernet and in conjunction with GMII for 1 Gb/s Ethernet. This interface is described in clauses 22 and 45 of the IEEE 802.3 standard [6].

Signals

The management interface uses the following signals as specified in section 22.2.2 of the standard:

- The **MDC** (management data clock) signal is generated by the host and is used to synchronize the interface. According to the standard, the frequency should be at most 2.5 MHz. According to the data sheet of the PHY on our board, the frequency could be up to 25 MHz.
- The **MDIO** (management data input/output) signal is driven either by the host or by the PHY and is used to transport the data. There must be a pull up resistor connected to this signal which drives the signal during the turnaround time.

Frames

Every frame transported on the management interface consists of 32 bits of preamble and 32 bits of data. The preamble is always 32 bits of ones and the PHY uses it to for synchronization with the host. It may be omitted if both the PHY and the host support frames without preamble.

When the management interface is used according to clause 22, the data has the following structure as specified in section 22.2.4.5 of the standard:

- The **SF** (start of frame) consists of 2 bits and marks the first bits of the frame. It is always **'10'** for clause 22.
- The **OP** (operation code) consists of 2 bits and determines whether the frame is a read transaction or a write transaction.
  - For read transactions, the operation code is '10'.
  - For write transactions, the operation code is '01'.
- The **PHYAD** (PHY address) consists of 5 bits and allows addressing up to 32 different PHYs connected to one host. The address is given by the manufacturer of the PHY. It can by usually modified by connecting different values of strap resistors.

- The **REGAD** (register address) consists of 5 bits and allows addressing up to 32 different registers in one PHY. Common registers are described in section 22.2.4 of the standard. Vendor specific registers are described in the data sheet of the PHY.
- The **TA** (turnaround) consists of 2 bits and provides time for taking over the bus during read transactions.
  - For read transactions, the turnaround is 'Z0'. That means neither the PHY nor the host hold the bus on the first bit and the PHY pulls down the bus on the second bit.
  - For write transactions, the turnaround is '10'.
- The **DATA** (data) consits of 16 bits and transports the value of the chosen register from or to the chosen PHY.

When the management interface is used according to clause 45, the data has the following structure as specified in section 45.3 of the standard:

- The **SF** (start of frame) consists of 2 bits and marks the first bits of the frame. It is always **'00'** for clause 45.
- The **OP** (operation code) consists of 2 bits and determines the type of the transaction.
  - For address transactions, the operation code is '00'.
  - For write transactions, the operation code is '01'.
  - For read transactions, the operation code is '11'.
- The **PRTAD** (port address) consists of 5 bits and allows addressing up to 32 different ports connected to one host. The address is given by the manufacturer of the PHY.
- The **DEVAD** (device address) consists of 5 bits and allows addressing up to 32 different devices of one port. Common devices are described in section 45.2 of the standard. Vendor specific devices are described in the data sheet of the PHY.
- The **TA** (turnaround) consists of 2 bits and provides time for taking over the bus during read transactions. It is the same as for clause 22.
- The last field consists of 16 bits and its meaning depends on the type of the transaction:
  - For address transactions, it sets the address of the register to be read or written by the following transaction.
  - For read and write transactions, it transports the value of the chosen register from or to the chosen device.

2.1.9 SFP+ module

The SFP+ standard stands for Small Form Pluggable and provides a common specification for modules serving as ten gigabit Ethernet transceivers. It is the next generation of the SFP standard which was limited to gigabit Ethernet.

Connector

The connector of the SFP+ module is described in section 3 of the SFF-8419 specification [14]. It consists of 20 pins with a pitch of 0.8 mm. The pinout of the connector is shown in figure 2.7.

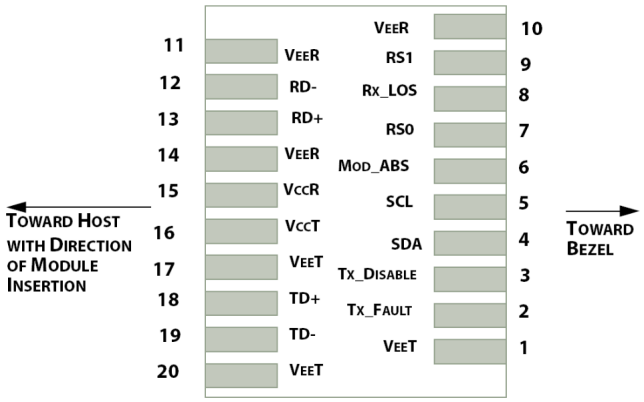

Figure 2.7: SFP+ connector

High speed interface

The high speed interface of the SFP+ module is described in section 3 of the SFF-8418 specification [13]. Actually, there are several different interfaces described in the specification. Every host and module may support more than one interface but only one of them is chosen for the communication.

- The **10GBASE-R** PHY interface as specified in clause 49 of the IEEE 802.3 standard with signaling rate of 10.3125 GBd.
- The **10GBASE-W** PHY interface as specified in clause 50 of the IEEE 802.3 standard with signaling rate of 9.95328 GBd.

In our case, we will be using special SFP+ modules with RJ-45 ports which support multiple speeds of Ethernet ranging from 10 Mb/s to 10 Gb/s. After some investigation, it turned out that they contain special transceivers that perform the conversion between the different physical layers. On the host side, they provide the **10GBASE-R** interface to comply with the SFP+ specification. On the copper side, they support the **10GBASE-T** interface and also its slower alternatives to provide the Ethernet port.

Low speed interface

The low speed interface of the SFP+ module is described in section 4 of the SFF-8419 specification [14]. It consists of the following signals:

- The **Tx_Fault** signal goes from the module to the host and indicates that the transmitter has detected a fault.
- The **Tx_Disable** signal goes from the host to the module and controls the transmitter.
  - The transmitter is disabled when it is high or left open.
  - The transmitter is enabled only when it is low.
- The **Rx_LOS** signal goes from the module to the host and indicates that the received signal strength is below the specified range.
- The **Mod_ABS** signal goes from the module to the host and can be used to detect whether the module is present or absent.
- The **RS0** and **RS1** signals go from the host to the module and set the signaling rate of the module.

All signals use the LVTTL (low voltage TTL) standard with a voltage of 3.3 V.

Two wire interface

The two wire interface of the SFP+ module is described in section 5 of the SFF-8419 specification [14]. It consists of the following signals:

- The **SCL** (serial clock) goes from the host to the module and provides the clock.
- The **SDA** (serial data) goes in both directions and transports the data.

The two wire interface is compatible with the I2C bus. The transactions used for reading from and writing to SFP+ management registers use the same format as transactions used for reading from and writing to I2C memories. The address of the SFP+ module is always 0xA0 and 0xA2 as defined in section 5.5 of the specification. The memory map of the SFP+ module is defined in the SFF-8472 specification [12].

Read transaction

In order to read a value from a register of the SFP+ module, the following steps shall be performed as described in section 5.6.3 (or 5.6.4) of the specification:

1. Generate the START condition.
2. Transmit the address of the SFP+ module with the write bit and check the ACK.
3. Transmit the address of the register to be read from and check the ACK.
4. Generate another START condition.
5. Transmit the address of the SFP+ module with the read bit and check the ACK.
6. Receive the value of the register and send the NACK (negative ACK).
7. Generate the STOP condition.

Write transaction

In order to write a value into a register of the SFP+ module, the following steps shall be performed as described in section 5.6.5 (or 5.6.6) of the specification:

1. Generate the START condition.
2. Transmit the address of the SFP+ module with the write bit and check the ACK.
3. Transmit the address of the register to be written to and check the ACK.
4. Transmit the value of the register and check the ACK.
5. Generate the STOP condition.

2.2 AMD Kria

The AMD Kria product line consists of several system on modules and starter kits featuring the AMD Zynq UltraScale+ system on chip.

2.2.1 Kria K26 System on Module

This system on module integrates a custom-built Zynq UltraScale+ system on chip with DDR memory and non-volatile storage devices. More information about the system on module can be found in the data sheet [9]. The board is shown in figure 2.8.

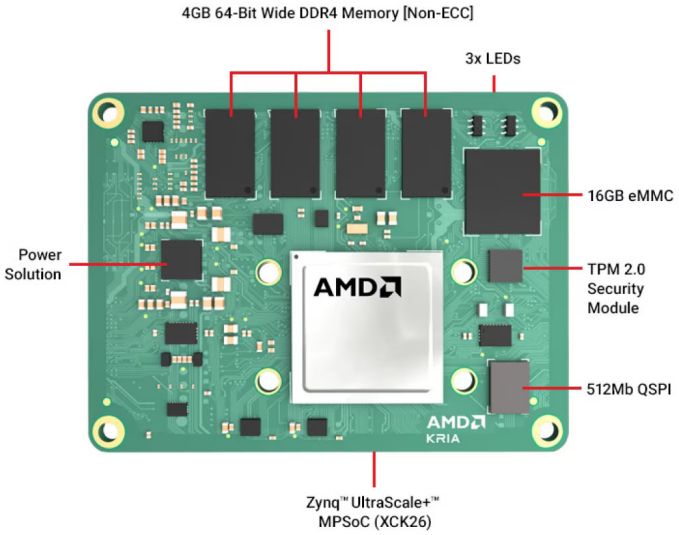

Figure 2.8: Image of Kria K26
(taken from web)

Components

This system on module contains the following main components:

- Zynq UltraScale+ MPSoC
- 4 GB of DDR4 memory
- 16 GB of eMMC memory
- 512 Mb of QSPI memory
- 64 Kb of EEPROM memory
- TPM2.0 security module

The MPSoC (Multi Processor System-on-Chip) combines the processing system (PS) and the programmable logic (PL) onto one chip and thus allows tight integration of the custom software (in the PS) and the custom hardware (in the PL) that will be required for completing this project.

Processing system

The processing system (PS) includes the following hardware per the data sheet [9]:

- Application Processing Unit (**APU**) consisting of quad-core ARM Cortex-A53 processors with maximum frequency of 1333 MHz
- Real-time Processing Unit (**RPU**) consisting of dual-core ARM Cortex-R5F processors with maximum frequency of 533 MHz
- Graphics Processing Unit (**GPU**) based on ARM Mali-400 with maximum frequency of 600 MHz
- Platform Management Unit (**PMU**) for power management

More information about the processing system can be found in the technical reference manual of the Zynq UltraScale+ platform [23].

Programmable logic

The programmable logic (PL) includes the following hardware per the data sheet [9]:

- 256K programmable logic cells
- 144 blocks of 36 Kb Block RAM
- 64 blocks of 288 Kb Ultra RAM
- 1248 slices of DSP (multiplier and accumulator)
- 4 GTH transceivers (up to 12.5 Gb/s serial transceivers)
- 1 video codec (H.264 and H.265 encoder and decoder)
- 69 high-density I/O (with 1.2V to 3.3 V rails)
- 58 high-performance I/O (with 1.0 to 1.8 V rails)

In our project

The specifications are good enough for implementing the whole project on this single module.

The 1000BASE-T1 transceivers will be connected using the high-performance I/O which supports data rates of up to 2.5 Gb/s. It also provides support for DDR (double data rate) interfaces according to chapter 2 of the corresponding user guide [19].

The 10GBASE-T transceivers will be connected using the GTH transceivers which support data rates of up to 12.5 Gb/s. They also provide support for SFP+ modules according to chapter 1 of the corresponding user guide [17].

2.2.2 Kria K24 System on Module

This system on module integrates a custom-built Zynq UltraScale+ system on chip with DDR memory and non-volatile storage devices. More information about the system on module can be found in the data sheet [8]. The board is shown in figure 2.9.

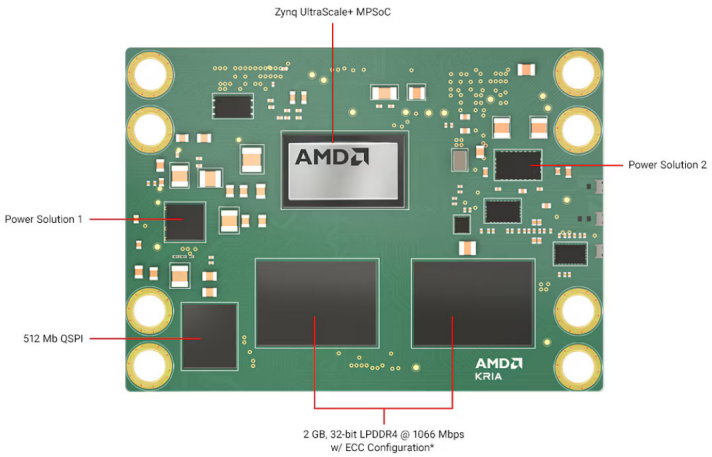

Figure 2.9: Image of Kria K24
(taken from web)

Components

This system on module contains the following main components:

- Zynq UltraScale+ MPSoC
- 2 GB of DDR4 memory
- 32 GB of eMMC memory
- 512 Mb of QSPI memory
- 64 Kb of EEPROM memory
- TPM2.0 security module

Processing system

The processing system (PS) is the same as in the Kria K26 System on Module.

Programmable logic

The programmable logic (PL) includes the following hardware per the data sheet [8]:

- 154K programmable logic cells
- 216 blocks of 36Kb Block RAM
- 360 slices of DSP (multiplier and accumulator)
- 23 high density I/O (with 1.2V to 3.3V rails)
- 56 high performance I/O (with 1.0V to 1.8V rails)

2.2.3 Kria KR260 Robotics Starter Kit

This starter kit extends the Kria K26 System on Module by adding several peripherals with focus on automation and robotics applications. The purpose of the starter kit is to allow custom hardware and software development before designing and manufacturing a custom board. More information about the starter kit can be found in the data sheet [10]. The board is shown in figure 2.10 and the block diagram can be found in figure 2.11.

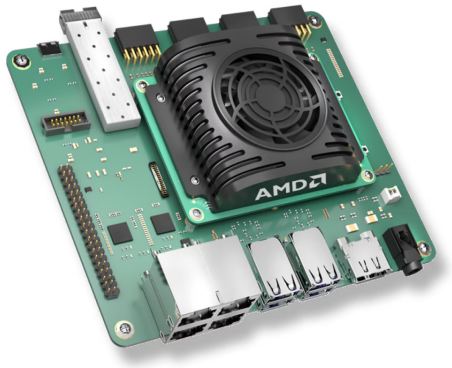

Figure 2.10: Image of Kria KR260
(taken from web)

Components

This starter kit consists of the following components:

- AMD Kria K26 System on Module described in subsection 2.2.1
- 4 * Triple speed Ethernet PHY by Texas Instruments
- 2 * Four port USB 3.0 hub by Microchip
- 1 * SFP+ connector with support of 10 Gb/s Ethernet
- 1 * USB to JTAG and UART converter by FTDI
- 1 * DisplayPort 1.2a video output
- 1 * microSD card slot
- 1 * SLVS-EC interface
- 1 * Raspberry Pi header
- 4 * PMOD 12-pin interface

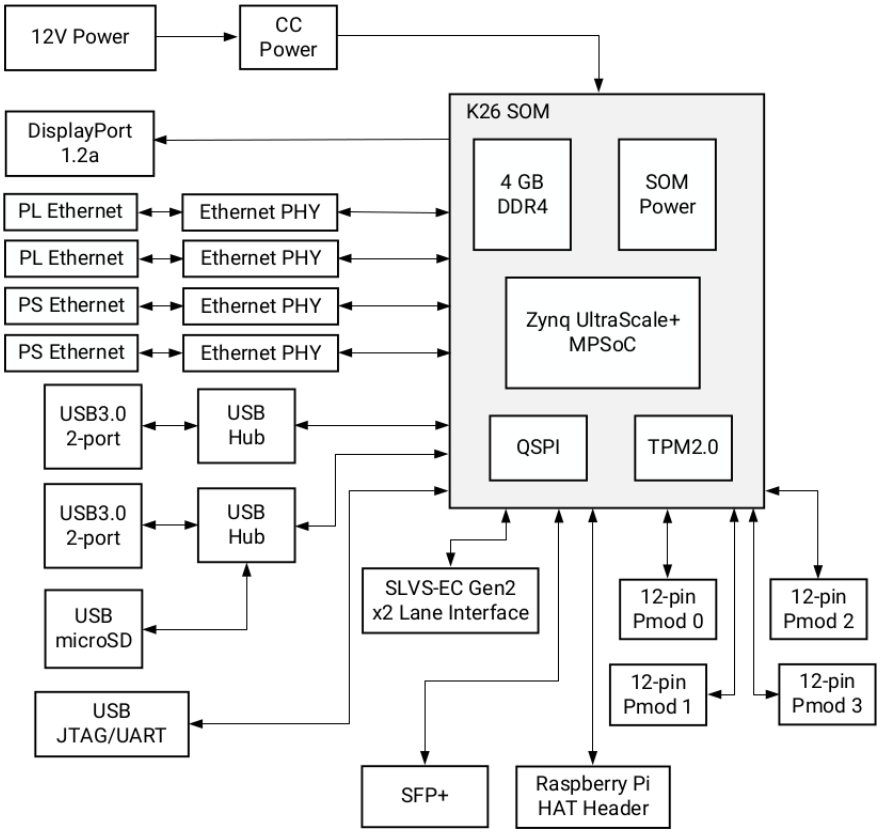

Figure 2.11: Block diagram of Kria KR260
(taken from [10])

In our project

The most important components for this project are four gigabit Ethernet transceivers and one SFP+ slot for ten gigabit Ethernet transceiver. Due to the internal structure of the chip and the connections of the board, only two of the gigabit Ethernet ports can be used with custom hardware in the programmable logic. The other two ports are directly connected to the processing system (one of them via RGMII and one of them via SGMII) and cannot be accessed from the programmable logic.

2.2.4 Kria KV260 Vision AI Starter Kit

This starter kit extends the Kria K26 System on Module by adding several peripherals with focus on vision and artificial intelligence applications. The purpose of the starter kit is to allow custom hardware and software development before designing and manufacturing a custom board. More information about the starter kit can be found in the data sheet [11]. The board is shown in figure 2.12 and the block diagram can be found in figure 2.13.

Figure 2.12: Image of Kria KV260
(taken from web)

Components

This starter kit consists of the following components:

- AMD Kria K26 System on Module described in subsection 2.2.1
- 1 * DisplayPort 1.2a video output connected via video splitter
- 1 * HDMI 1.4 video output connected via video splitter
- 1 * Triple speed Ethernet PHY by Texas Instruments
- 1 * Four port USB 3.0 hub by Microchip
- 1 * Image sensor processor by OnSemi
- 1 * USB to JTAG and UART converter by FTDI
- 1 * microSD card slot
- 2 * IAS MIPI sensor interface
- 1 * Raspberry Pi camera interface
- 1 * PMOD 12-pin interface

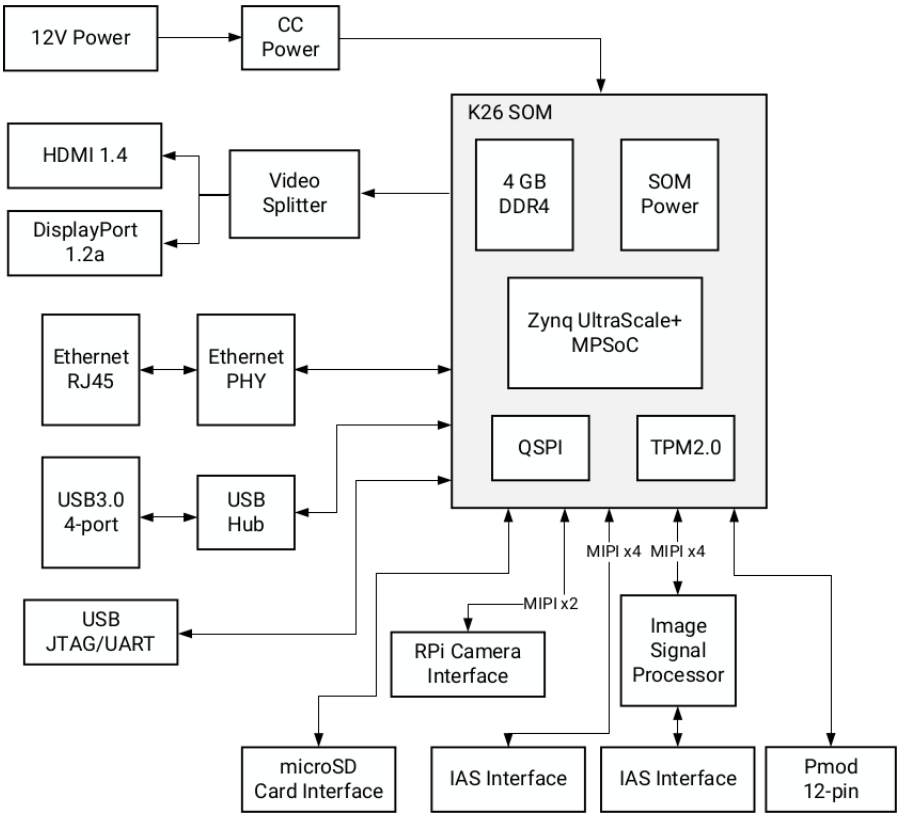

Figure 2.13: Block diagram of Kria KV260
(taken from [11])

2.3 PTP protocol

PTP stands for Precision Time Protocol. It is a network protocol which can be used to synchronize time between devices via Ethernet networks with precision better than one microsecond. It is specified by the IEEE 802.1AS standard [7].

2.3.1 Delay measurement

The first part of the process is the propagation delay measurement. The goal is to measure the time it takes every frame to travel between two devices via the Ethernet link as precisely as possible. The propagation delay is expected to be constant because it should only depend on static factors like the length of the Ethernet cable.

That is why the timestamping of received and transmitted frames must be done in hardware as close to the Ethernet medium as possible. In case the timestamping was done in software, the propagation delay would also include the latency between hardware and software which depends on dynamic factors like the current utilization of the system.

The delay measurement is always initiated by one of the two devices connected via the Ethernet link. If both devices need to know the propagation delay, both devices must initiate their own delay measurement. However, that is not required when the master and slave roles are set statically so the direction of the time synchronization never changes. In that case, only the device with the slave role must initiate the delay measurement.

In case of devices with multiple Ethernet ports like switches and routers, the delay measurement must be done separately for every Ethernet port which requires PTP support. In this section, the delay measurement will be described for two devices each having only one Ethernet port for simplicity.

Process of measurement

The process described below is shown in figure 2.14 taken from the standard [7]. More information about the process can be found in section 11.1.2 of that standard.

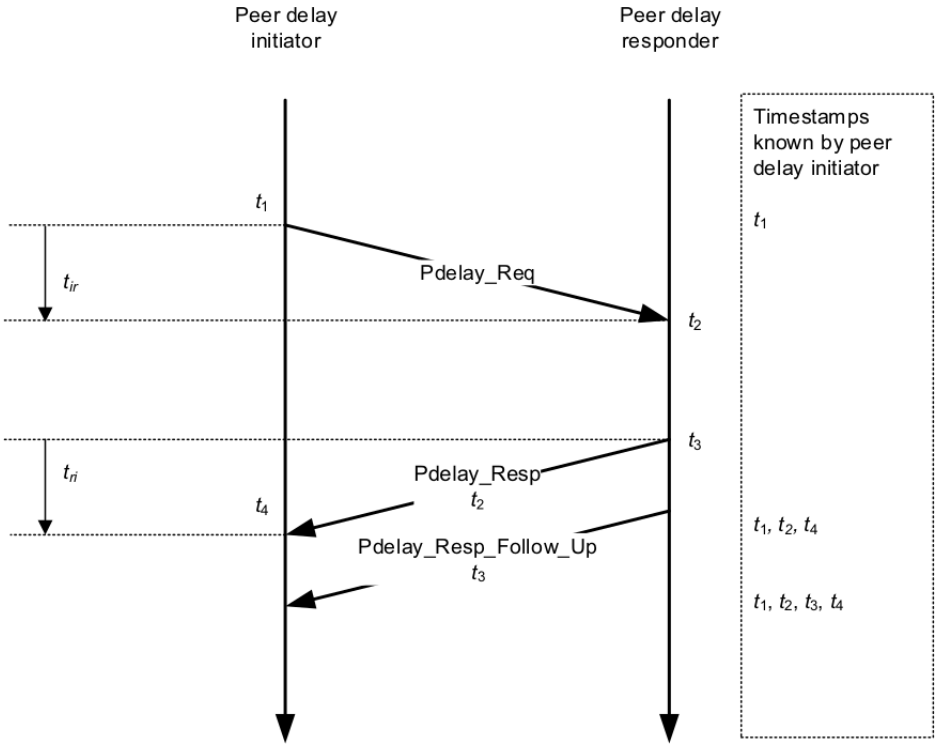

Figure 2.14: PTP delay measurement [7]

The initiator starts by sending the *Pdelay_Req* message. It then saves the egress timestamp of that message (the local timestamp of when the message was transmitted by the Ethernet hardware). It is marked as t_1 in the figure.

The responder waits until the *Pdelay_Req* message is received. It then saves the ingress timestamp of that message (the local timestamp of when the message was received by the Ethernet hardware). It is marked as t_2 in the figure.

The responder continues by sending the *Pdelay_Resp* message. It puts the saved ingress timestamp of the *Pdelay_Req* message into that message. It also saves the egress timestamp of that message. It is marked as t_3 in the figure.

The responder finishes by sending the *Pdelay_Resp_Follow_Up* message. It puts the saved egress timestamp of the *Pdelay_Resp* message into that message.

The initiator waits until both the *Pdelay_Resp* and the *Pdelay_Resp_Follow_Up* messages are received. It then saves the ingress timestamp of the *Pdelay_Resp* message. It is marked as t_4 in the figure.

The initiator finally calculates the propagation delay using the four timestamps. Two of them have been captured by the initiator itself. Two of them have been transported from the responder to the initiator by the messages. The initiator first computes the propagation delay from the initiator to the responder (marked as t_{ir} in the figure) and the propagation delay from the responder to the initiator (marked as t_{ri} in the figure). It then uses the equation below to compute the propagation delay (marked as D).

$$D = \frac{t_{ir} + t_{ri}}{2} = \frac{(t_4 - t_1) - (t_3 - t_2)}{2}$$

2.3.2 Time synchronization

The second part of the process is the time synchronization. The goal is to establish synchronization of current time between the grand master which is usually the device with the most accurate source of current time and the slaves which are the other devices of the network.

The time synchronization is always done between two devices connected via an Ethernet link similar to the delay measurement. The time is first synchronized between the grand master and the first layer of slaves directly connected to it. If any of those slaves has more Ethernet ports, it may become the master for another layer of slaves directly connected to it. That continues until every device of the network has the time synchronized with the grand master.

The grand master can be selected by the Best Master Clock Algorithm (BMCA) described in section 10.3.1 of the standard. Another way to configure the master and slave roles in the network is to set them statically for every port of every device. That is actually the preferred setup in automotive networks because their topology is static by design.

Modes of synchronization

The time synchronization may work in two different modes:

1. In the **one step** mode, there is only one message sent from the master to the slave. For that to work, the master must be able to fill in the timestamp of when the message was transmitted into the message during transmission. It requires special support in the Ethernet hardware.

2. In the **two step** mode, there are two messages sent from the master to the slave. The second message transports the timestamp of when the first message was transmitted. It does not require special support beyond hardware timestamping in the Ethernet hardware. It is the preferred mode in automotive networks.

Process of synchronization

The process described below is shown in figure 2.15 taken from the standard [7]. More information about the process can be found in section 11.1.3 of that standard.

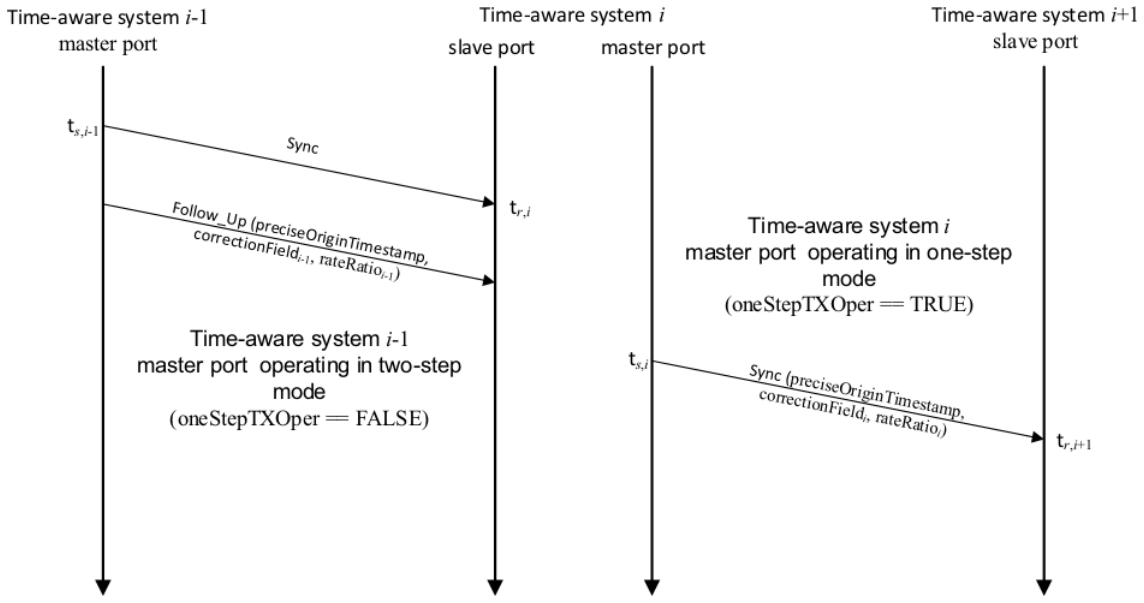

Figure 2.15: PTP time synchronization [7]

Let us consider three PTP instances. The time synchronization will happen from the left to the right. The instance on the left labeled $i-1$ is the closest to the grand master. The instance in the middle labeled i uses the instance labeled $i-1$ for synchronizing the time. In our case, that would be our device being transparent in terms of PTP when inserted into the network. The instance on the right labeled $i+1$ uses the instance labeled i for synchronizing the time.

The instance $i-1$ starts by sending the *Sync* message and the *Follow_Up* message.

The instance i starts by receiving the *Sync* and *Follow_Up* messages. It saves the ingress timestamp of the *Sync* message which is marked as $t_{r,i}$ in the figure. It also saves the fields of the *Follow_Up* message to be used later in the process.

The instance i continues by sending the *Sync* message. It saves the egress timestamp of that message which is marked as $t_{s,i}$ in the figure.

The instance i finishes by sending the *Follow_Up* message. The fields of that message are set as follows:

The origin timestamp is the time of the grand master clock when the sync information has been sent by the grand master. It is taken from the *Follow_Up* message. It does not change throughout the process of time synchronization.

The correction field represents the difference between the moment the sync information has been sent by the grand master and the moment the sync information has been sent by the current instance. The correction field for instance i is computed as the sum of the following components:

- The correction field for instance $i-1$ received as part of the *Follow_Up* message. It accounts for the delay introduced between the grand master and the instance $i-1$. It is already expressed in the grand master clock time base.
- The mean link delay between instance $i-1$ and instance i determined by the delay measurement process described above. It accounts for the delay introduced by the Ethernet hardware including the transceiver and the cable. It must be multiplied by the rate ratio for instance $i-1$ received as part of the *Follow_Up* message in order to express it in the grand master clock time base.
- The residence time in instance i computed as the difference of $t_{s,i}$ and $t_{r,i}$. It accounts for the delay introduced by the PTP software which receives and transmits the messages according to this process. It must be multiplied by the rate ratio for instance i computed below in order to express it in the grand master clock time base.

The rate ratio represents the difference between the period of the grand master clock and the period of the local clock of the current instance. The rate ratio for instance i is computed as the product of the following components:

- The rate ratio for instance $i-1$ received as part of the *Follow_Up* message. It accounts for the drift between the grand master and the instance $i-1$.
- The neighbor rate ratio between instance $i-1$ and instance i determined by the delay measurement process described above.

2.3.3 Messages

This subsection describes the format of PTP messages in general.

Ethernet header

Every PTP message is transported via Ethernet with Ethernet header. The values of the fields of the Ethernet header are specified in sections 10.5 and 11.3 of the standard:

- The destination MAC address should be **01-80-C2-00-00-0E**.
- The source MAC address should be the MAC address of the respective egress port.
- The EtherType should be **88-F7**.

PTP header

Every PTP message starts with the PTP message header. Its structure is specified in sections 10.6.2 and 11.4.2 of the standard. It is also shown in table 2.7.

Name	Length	Description
majorSdoId	1/2	Always 0x1 as specified by section 8.1 of the standard.
messageType	1/2	**Type of the PTP message.**
minorVersionPTP	1/2	Always 0x1 as specified in section 10.6.2 of the standard.
versionPTP	1/2	Always 0x2 as specified in section 10.6.2 of the standard.
messageLength	2	**Total length of the PTP message in bytes.**
domainNumber	1	Always 0x00 for the default domain as specified in section 8.1 of the standard.
minorSdoId	1	Always 0x00 as specified in section 8.1 of the standard.
flags	2	The only relevant flag is twoStepFlag which should be set for *Sync* and *Pdelay_Resp* messages.
correctionField	8	**Correction of the PTP timestamp.**
messageTypeSpecific	4	May be used for implementation specific purposes.
sourcePortIdentity	10	**Port identity of the egress PTP port.**
sequenceId	2	**Sequence number of the PTP message.**
controlField	1	Always 0x00 as specified in section 10.6.2 of the standard.
logMessageInterval	1	**Interval of the PTP message.**

Table 2.7: Structure of PTP header

PTP timestamp

The PTP protocol uses a specific structure for transporting timestamps. It is defined in section 6.4.3.4 of the standard. It has the following fields:

- The **seconds** represent the complete seconds that have passed since the start of the epoch. This field uses 48 bits (6 bytes).
- The **nanoseconds** represent the additional nanoseconds that have passed since the start of the epoch so it is always less than 10^9. This field uses 32 bits (4 bytes).

The start of the epoch is set to 1/1/1970 00:00:00 in section 8.2.2 of the standard.

PTP port identity

The PTP protocol uses a specific structure for identifying the ports. It is defined in section 6.4.3.6 of the standard. It has the following fields:

- The **clock identity** identifies the PTP instance. It must be unique in the network so it should be generated from the MAC address or another unique device identifier according to the IEEE 1588 standard. This field uses 64 bits (8 bytes).
- The **port number** identifies the PTP port of the PTP instance. It must be unique for the instance. This field uses 16 bits (2 bytes).

2.3.4 Sync messages

This subsection describes the PTP messages relevant for implementing the time synchronization on our device.

Sync message

The *Sync* message is used to synchronize the time. It is sent from the master to the slave. Its structure is specified in section 11.4.3 of the standard and also shown in table 2.8.

Name	Length	Description
header	34	PTP header as shown in table 2.7.
reserved	10	Reserved.

Table 2.8: Format of PTP Sync message

Follow_Up message

The *Follow_Up* message transports the egress timestamp of the *Sync* message. Its structure is specified in section 11.4.4 of the standard and also shown in table 2.9.

Name	Length	Description
header	34	PTP header as shown in table 2.7.
preciseOriginTimestamp	10	The egress timestamp of the *Sync* message.
followUpInformationTlv	32	The *Follow_Up* information TLV.

Table 2.9: Format of PTP Follow_Up message

2.3.5 Delay messages

This subsection describes the PTP messages relevant for implementing the delay measurement on our device.

Pdelay_Req message

The *Pdelay_Req* message is used to measure the delay of the link. It is sent by the initiator of the delay measurement. Its structure is specified in section 11.4.5 of the standard and also shown in table 2.10.

Name	Length	Description
header	34	PTP header as shown in table 2.7.
reserved	10	Reserved.
reserved	10	Reserved.

Table 2.10: Format of PTP Pdelay_Req message

Pdelay_Resp message

The *Pdelay_Resp* message is used to measure the delay of the link. It is sent by the responder of the delay measurement. Its structure is specified in section 11.4.6 of the standard and also shown in table 2.11.

Name	Length	Description
header	34	PTP header as shown in table 2.7.
requestReceiptTimestamp	10	The ingress timestamp of the *Pdelay_Req* message.
requestingPortIdentity	10	The source port identity of the *Pdelay_Req* message.

Table 2.11: Format of PTP Pdelay_Resp message

Pdelay_Resp_Follow_Up message

The *Pdelay_Resp_Follow_Up* message transports the egress timestamp of the *Pdelay_Resp* message. Its structure is specified in section 11.4.7 of the standard and also shown in table 2.12.

Name	Length	Description
header	34	PTP header as shown in table 2.7.
responseOriginTimestamp	10	The egress timestamp of the *Pdelay_Resp* message.
requestingPortIdentity	10	The source port identity of the *Pdelay_Req* message.

Table 2.12: Format of PTP Pdelay_Resp_Follow_Up message

2.4 CMP protocol

CMP stands for Capture Module Protocol. It is a network protocol which can be used to log traffic from all kinds of automotive networks including Ethernet. It is specified by the ASAM CMP standard [3].

The protocol is used to transport data, status and control messages between the capture module and the data sink via Ethernet. The capture module is the device where the traffic is captured from the automotive network. The data sink is the device where the traffic is stored for further processing. There may be multiple capture modules and multiple data sinks on one Ethernet network as shown in figure 2.16.

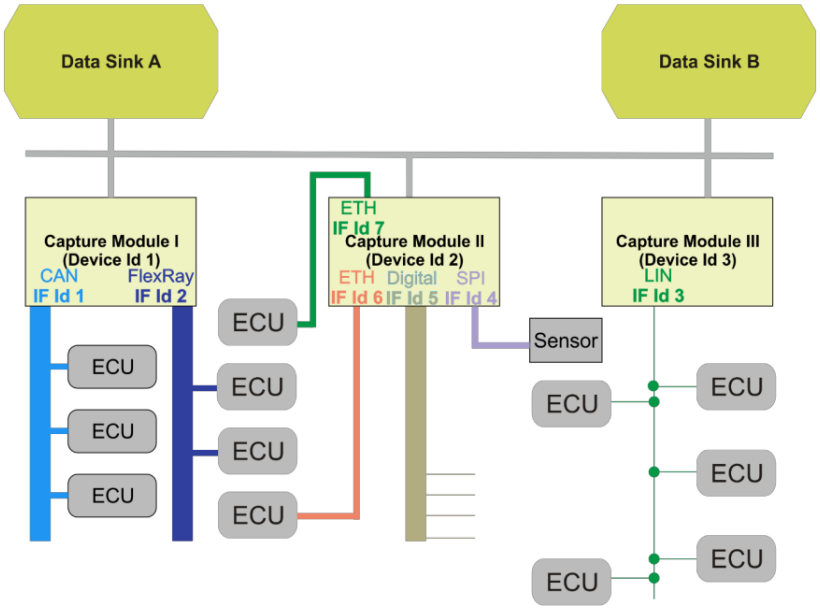

Figure 2.16: CMP capture modules and data sinks [3]

In our case, the capture module is our device and the data sink is a personal computer running Wireshark until another piece of software is developed for that purpose.

2.4.1 Messages

This subsection describes the format of CMP messages in general.

Ethernet header

Every CMP message is transported via Ethernet with Ethernet header. The values of the fields of the Ethernet header are specified in section 3.4.1 of the standard:

- The destination MAC address should be the MAC address of the data sink.
- The source MAC address should be the MAC address of the capture module.
- The EtherType should be **99-FE**.

CMP message

Every CMP message (with no aggregation or segmentation) consists of the following components as described in section 3.2 of the standard:

- The CMP header described in section 3.2.1 of the standard. It has 8 bytes. The structure is also shown in table 2.13.
- The CMP message header and payload. The structure depends on the type of the message.

Name	Length	Description
Version	1	Version of the CMP protocol. It is always 0x01 for the current version.
Reserved	1	Reserved for future use.
Device ID	2	Identifier of the device. It should be unique within the entire network.
Message Type	1	Type of the message following this header.
Stream ID	1	Identifier of the stream. In our case, only one stream will be generated.
Stream Counter	2	Sequence counter of the stream. It should be incremented for every message generated.

Table 2.13: Structure of CMP header

Aggregation

Multiple CMP payloads of the same type may be aggregated into single CMP package. It is described in section 3.3.2 of the standard. Every CMP package then consists of the following components:

- The common CMP header.
- The first CMP message header and payload.
- The second CMP message header and payload.
- The third CMP message header and payload.
- ...

That allows lowering the overhead of the CMP protocol and the workload of the network by using single CMP header for multiple CMP messages.

Segmentation

Single CMP payload may be segmented into multiple CMP packages of the same type. It is described in section 3.3.3 of the standard. The CMP payload is then transported in the following messages:

- The first CMP message with the Segment Start flag in the message header.
- Any number of CMP messages with the Segment flag in the message header when there are more than two segments.
- The last CMP message with the Segment End flag in the message header.

That allows transporting data longer than 1500 bytes (or 9000 bytes when jumbo frames are considered) by splitting the original data into multiple CMP frames for transport via Ethernet.

2.4.2 Data messages

Data messages transport frames or other data captured on one of the interfaces of the capture module. More information about data messages can be found in chapter 4 of the standard [3].

Every data message consists of the following parts:

- The data message header described in section 4.2 of the standard. It has 16 bytes. The structure is shown in table 2.14.
- One of the data message payloads described in section 4.3 of the standard. The structure depends on the type of the payload.

Payloads

Each payload type represents one network type used in the automotive industry. The following payloads and networks are supported by the current version of the protocol:

- CAN / CAN-FD (see section 4.3.1 or 4.3.2)
- LIN (see section 4.3.3)
- FlexRay (see section 4.3.4)
- Digital signals (see section 4.3.5)
- UART / RS-232 (see section 4.3.6)
- Analog signals (see section 4.3.7)
- **Ethernet** (see section 4.3.8)
- SPI (see section 4.3.9)
- I2C (see section 4.3.10)

It is also possible to create vendor specific data message payloads by using 0xFF as the payload type in the header.

In our case, we will always generate the Ethernet payload described in section 4.3.8 of the standard. The structure is shown in table 2.15.

Name	Length	Description
Timestamp	8	Timestamp when the frame was captured.
Interface ID	4	Identifier of the interface where the frame was captured.
Flags	1	Flags of the message. In our case, the direction flag will be used to indicate whether the frame was received or transmitted on the interface.
Payload Type	1	Type of the payload following this header.
Payload Length	2	Length of the payload following this header.

Table 2.14: Structure of CMP data message header

Name	Length	Description
Flags	1	Flags of the payload.
Reserved	1	Reserved for future use.
Data Length	2	Length of the data following this header.
Data	N	Content of the Ethernet frame starting with the Ethernet header and ending with the Frame Check Sequence (FCS).

Table 2.15: Structure of CMP Ethernet payload

2.4.3 Status messages

Status messages allow reporting status of the capture module and its interfaces to the data sink. More information about status messages can be found in chapter 5 of the standard [3].

Every status message consists of the following parts:

- The status message header described in section 5.2 of the standard. It has 16 bytes.
- One of the status message payloads described in section 5.3 of the standard. The structure depends on the type of the payload.

Payloads

The following status message payloads should be sent by the capture module periodically:

- Capture Module Status Message (see section 5.3.1.1) should be generated every second. It contains the description of the module including the serial number and hardware and software version and also the current status of the time synchronization.
- Interface Status Message (see section 5.3.1.2) should be generated every second for every interface. It contains the total count of frames received and transmitted on that interface and also the current status of the interface link.
- Configuration Status Message (see section 5.3.1.3) should be generated every 60 seconds. It contains the current configuration of the module in vendor specific format.

There are other status messages payloads which are sent by the capture module when an event such as loss of time synchronization occurs.

It is also possible to create vendor specific status message payloads by using 0xFF as the payload type in the header.

2.4.4 Control messages

Control messages allow setting up the capture module and its interfaces from the data sink. More information about control messages can be found in chapter 6 of the standard [3].

Every control message consists of the following parts:

- The control message header described in section 6.2 of the standard. It has 16 bytes.
- One of the control message payloads described in section 6.3 of the standard. The structure depends on the type of the payload.

Payloads

The following control message payloads should be sent by the data sink periodically:

- Data Sink Ready To Receive Control Message (see section 6.3.1) should be sent every second. It indicates that the data sink is ready to accept data from the capture module.

It is also possible to create vendor specific control message payloads by using 0xFF as the payload type in the header.

Chapter 3

Designing the Hardware

This chapter describes the process of designing all the hardware components needed to accomplish all the goals set by chapter 1. It is divided into several sections where each of them describes one part of the process. Each section first describes the requirements given by the topic, then it shows the solutions considered throughout the design process, and finally it documents the current state of the entities designed according to the chosen solution.

3.1 Defining the clock domains

This section is dedicated to the decision of how the design should be divided into different clock domains. All other signals are synchronous to one of the clocks so I decided to put this section to the very beginning. During the design process, several different approaches were considered, each of them having its advantages and disadvantages.

3.1.1 Requirements

The majority of signals will be transporting Ethernet frames but there will also be signals for management. First, the clocks for GE (gigabit Ethernet) will be discussed. Second, the clocks for XGE (ten gigabit Ethernet) will be added to the game. Finally, the management signals will complete the overview of the clock domains.

GE clocks

On the physical layer, the GE will be handled by an external PHY, so the requirements are given by the interface used between our hardware and the external PHY. As explained in section 2.1.4, the clocks for the RGMII interface are generated as follows:

- The **RX** (receive) clock is generated by the PHY and sent to the MAC along with data. It is usually recovered from the physical medium by the PHY and should be within 100 ppm of the nominal frequency.
- The **TX** (transmit) clock is generated by the MAC and sent to the PHY along with data. It should be within 100 ppm of the nominal frequency.

It means that for building a simple repeater, it is possible to connect the RX clock to the TX clock and it should work. Our case is a bit more complicated though, as we are going to interface with other entities. For example, we are going to transport selected frames to and from the processing system.

XGE clocks

On the physical layer, the XGE will be handled by an internal PHY implemented by Xilinx IP, so the requirements are given by the interface between our hardware and the internal PHY. Both RX and TX clocks are generated within the PHY and accessible to the rest of the design. In the current configuration of the IP, it is not possible to provide any other clock. So without further changes in the IP, there is no other option than using this clock for the XGE part of our design.

Management

Apart from the data path for transporting Ethernet frames, there will be the control path for controlling the components of our design from the processing system. As explained later in section 3.4, there will be two interfaces involved. A custom interface will be used for propagating the commands internally throughout our design. A standard interface called AXI will be used for getting commands from the processing system to our design. According to chapter 35 of the manual [23], it is possible to provide any clock from the programmable logic to the processing system for synchronization of the AXI interface.

It would be therefore beneficient to use the same clock for everything: the Ethernet interfaces, the internal management interface and the external management interface. However, that would be possible only if there was just one Ethernet port which is not our case. Moreover, it should be possible to run each pair of Ethernet ports at different speed (so that the first pair could run at 1 Gb/s and the second pair could run at 100 Mb/s). That means each pair will require separate clock. For those reasons, it will be better to use separate management clock whose frequency will not depend on the current Ethernet speed.

3.1.2 Solutions

In order to meet all the requirements, several solutions were considered, some of which are described in this subsection.

Using the received clocks

The first idea was to use the clocks received from the PHYs to drive all logic. That solution would not require any crossing of clock domains for implementing the bridge. Of course, the CDC would still have to be solved when transporting frames to and from the processing system.

However, this proved to be impractical for cases when only one port of the pair is used. For example, when I was testing a new implementation of DMA, I connected only one port for simplicity. Then I wondered why transmission was not working until I realized that there was no clock coming from the other port as there was no link to recover the clock from.

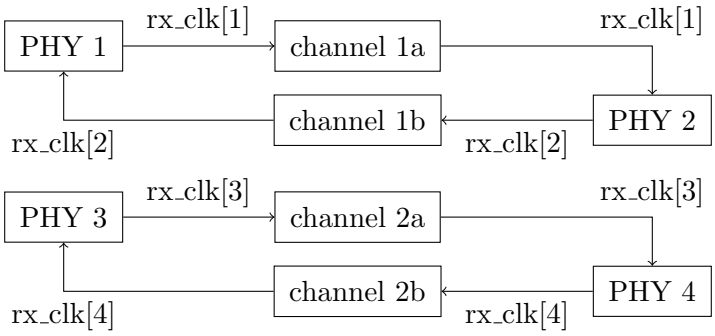

Figure 3.1: Using the received clocks

Using the generated clocks

The second idea was to generate common clock for all entities using the on-board oscillator. That would require the received stream to cross from received clock domain to common clock domain. No crossing would be required for transmission on the other hand. This idea simplified the design as everything except one entity for reception had the same clock domain.

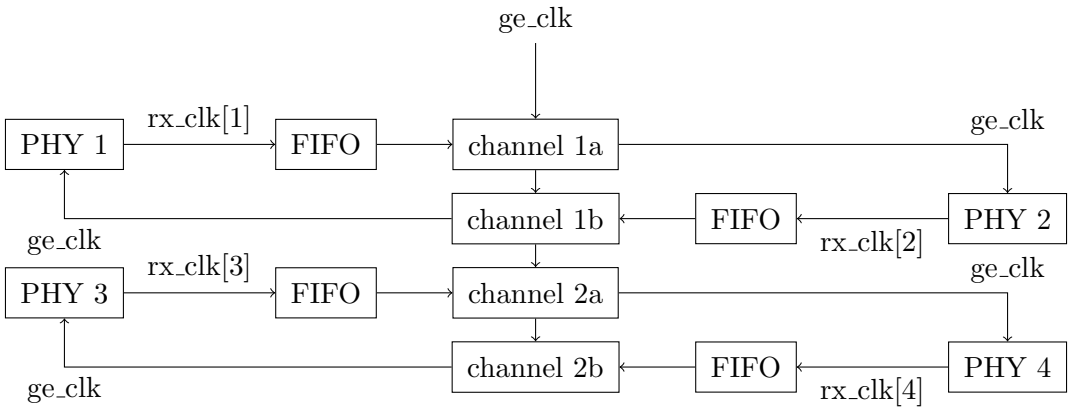

Figure 3.2: Using the generated clocks

Supporting multiple speeds

The clock structure had to be revised when support for multiple Ethernet speeds was desired. The default speed would still be 1 Gb/s Ethernet with 125 MHz clock for both internal and external interfaces. The other speeds to be supported by the design would be 100 Mb/s Ethernet and 10 Mb/s Ethernet with different clock for internal and external interfaces.

For the internal interfaces, the 125 MHz clock would be divided by 10 or 100 respectively to match the data rate and keep the width of the data path. For the external interfaces, the 125 MHz clock would be divided by 5 or 50 respectively and the width of the data path would be changed from 8 bits to 4 bits as required by the specification. In total, that would require generating 5 different clocks in the design so I decided to use a bit simpler solution.

The clock generated by the PHY would be used for both reception and transmission signals of the media independent interface. Thanks to that, it would not be necessary to generate the 25 MHz clock for 100 Mb/s Ethernet and the 2.5 MHz clock for 10 Mb/s Ethernet in our design. There would be a FIFO both in the receiver and the transmitter to transition between that external clock and the internal clock. It would still be necessary to generate the 12.5 MHz clock for 100 Mb/s Ethernet and the 1.25 MHz clock for 10 Mb/s Ethernet. It would also be required to provide a way to switch the clock for each pair of Ethernet ports.

3.2 Crossing the clock domains

As shown in the previous section, crossing clock domains will be required on several places of the design. For transporting data across asynchronous clock domains, FIFO should be used. Therefore, I prepared a few entities to simplify instantiation of all different kinds of FIFO that would be needed for the design.

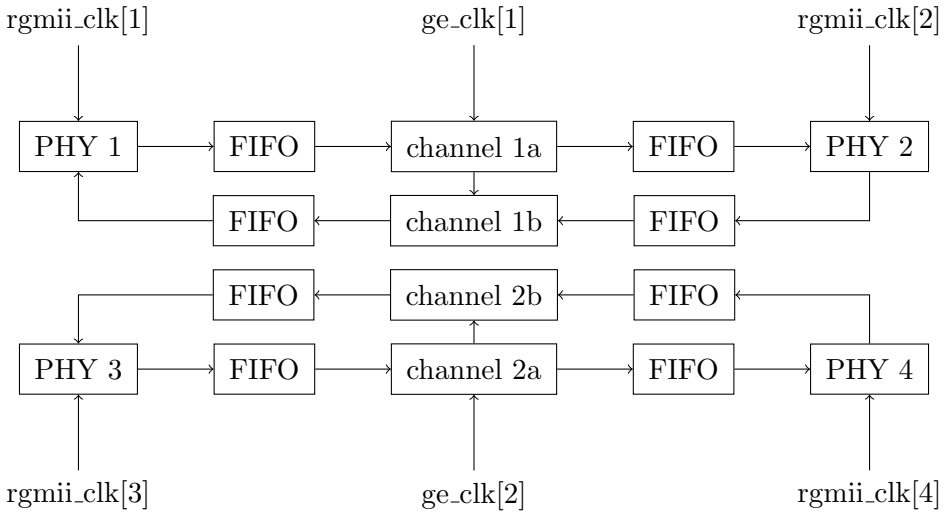

Figure 3.3: Supporting multiple speeds

3.2.1 Solutions

Using the Block RAM

Our kit contains about one hundred large memory blocks called Block RAM or BRAM which can be also used as FIFO. This is especially useful for FIFOs of high depth whose implementation using look-up tables or registers would be inefficient. More information on the Block RAM can be found in chapter 1 of UltraScale Architecture Memory Resources user guide [18].

According to the manual, the Block RAM must be instantiated manually when the internal FIFO logic is to be used. As with other Xilinx primitives, that can be done by using the Unisim package and instantiating one of the available primitives. Given the high amount of inputs which must be tied to zero when not used, instantiating the primitive directly would be inconvenient. For that reason, I decided to create wrappers for all different configurations of the primitive needed in my design.

There are some interesting features I would like to point out. In fact, there are two different primitives which may be used for instantiating a FIFO:

- **FIFO36E2** uses up one whole block. It provides 36K bits of storage and may be therefore used in the following configurations: 36K x 1, 18K x 2, 9K x 4, 4K x 9, 2K x 18, 1K x 36, 512 x 72. The maximal port width is 72 bits.
- **FIFO18E2** uses up only half of the block. It provides 18K bits of storage and may be therefore used in the following configurations: 18K x 1, 9K x 2, 4K x 4, 2K x 9, 1K x 18, 512 x 36. The maximal port width is 36 bits.

The first variant must be used when port width of more than 36 bits is required (which is common in our design due to ten gigabit Ethernet) or depth of more than 18K bits is required (which does not happen in our design).

As visible from the configurations above, there is one control bit for every eight data bits when the port width is more than or equal to 8 bits. This bit is originally meant for error correction but it may be also used to store additional bit of information. This feature will be used several times in our design to store control information.

To provide quick reference, the following table shows my entities which are essentially wrappers for Block RAM with the corresponding configuration:

Entity	Primitive	Clock domains	Write port	Read port
fifo8	FIFO18E2	common	8 + 1	8 + 1
fifo8x	FIFO18E2	independent	8 + 1	8 + 1
fifo8to32	FIFO18E2	independent	8 + 1	32 + 4
fifo8to64	FIFO36E2	independent	8 + 1	64 + 8
fifo64	FIFO36E2	common	64 + 8	64 + 8
fifo64to8	FIFO36E2	independent	64 + 8	8 + 1

Table 3.1: Wrappers of FIFO primitives

Using the LUT RAM

The programmable logic of our chip is built of look-up tables which may also act as memory and be used for storing information. This is especially useful for building FIFOs of low depth and atypical port width whose implementation using Block RAM would be inefficient. More information on the look-up tables may be found in chapter 2 of UltraScale Architecture Configurable Logic Block user guide [16].

3.2.2 Common clock FIFO

The common clock FIFO is implemented in the **fifo** entity. It uses the LUT RAM to create a FIFO of variable width and depth.

When the **wr_en** signal is asserted on the rising edge of the **clk** clock, data from the **wr_data** input is written to the FIFO.

When the **rd_en** signal is asserted on the rising edge of the **clk** clock, data available on the **rd_data** output is consumed and next data is read from the FIFO and put to the **rd_data** output.

The common clock FIFO has the following generic parameters:

- The **data_width** parameter sets the size of every item in the FIFO in bits. The size of wr_data and rd_data ports is given by this parameter.
- The **data_depth** parameter sets the maximal count of items in the FIFO before overflow occurs.

3.2.3 Independent clock FIFO

The independent clock FIFO is implemented in the **fifo_x** entity. It uses the LUT RAM to create a FIFO of variable width and depth.

When the **wr_en** signal is asserted on the rising edge of the **wr_clk** clock, data from the **wr_data** input is written to the FIFO.

When the **rd_en** signal is asserted on the rising edge of the **rd_clk** clock, data available on the **rd_data** output is consumed and next data is read from the FIFO and put to the **rd_data** output.

Three registers are used to prevent metastability and synchronize the current count of items between the write clock domain and the read clock domain. Every write generates an edge which is then detected in the read clock domain. Every read generates an edge which is then detected in the write clock domain.

The **wr_count** is synchronous to the **wr_clk** clock. It reports the current count of items from the write perspective and should be also used to detect whether the FIFO is **full**.

The **rd_count** is synchronous to the **rd_clk** clock. It reports the current count of items from the read perspective and should be also used to detect whether the FIFO is **empty**.

The independent clock FIFO has the same generic parameters as the common clock FIFO.

3.3 Designing the interfaces

Before designing the individual entities, it was necessary to define the interfaces which would be used for communication between them. If interfaces change later in the design process, all entities built with them must be redesigned which is exactly what happened to me. So I would like to stress that this is something what must be carefully decided in the very beginning.

3.3.1 Requirements

Our design is going to work mainly with gigabit Ethernet and ten gigabit Ethernet:

- Gigabit Ethernet (GE) uses a 125 MHz clock with 8 bit data path to provide bandwidth of 1 gigabit per second. For communication between MAC and PHY, the GMII or RGMII interface is used as described in sections 2.1.3 and 2.1.4.
- Ten Gigabit Ethernet (XGE) uses a 156.25 MHz clock with 64 bit data path to provide bandwidth of 10 gigabits per second. For communication between MAC and PHY, the XGMII interface is used as described in section 2.1.6.

Our design should also support slower Ethernet speeds (10 Mb/s, 100 Mb/s) which are still quite common in automotive industry. To simplify the design, I decided to design everything for gigabit Ethernet and slow down the clock for slower Ethernet speeds.

As noted above, there are standard interfaces for Ethernet which could be used between my entities. However, their usage would not be practical. According to the standards, the MAC must be able to receive frames with incomplete preamble. That means every entity would have to search for the SFD (Start of Frame Delimiter) to achieve synchronization. Also, we will need to timestamp frames in order to support the PTP (Precision Time Protocol). We should be able to add this (and possibly other) information to the data stream.

For those reasons, I decided to design my custom interfaces for internal transport of Ethernet frames. At the same time, I will use the standard interfaces for external transport of Ethernet frames from and to the PHY.

3.3.2 GMII interface

For communication with the gigabit Ethernet PHY, the standard GMII interface should be used. The individual signals are described in chapter 2.1.3.

In my design, this interface is implemented using the **gmii_port_t** record with the following members:

- **d** is vector of 8 bits and corresponds to the **RXD** (receive data) or the **TXD** (transmit data) GMII signal depending on the direction.
- **dv** corresponds to the **RX_DV** (receive data valid) or the **TX_EN** (transmit enable) GMII signal depending on the direction.
- **er** corresponds to the **RX_ER** (receive error) or the **TX_ER** (transmit error) GMII signal depending on the direction.

3.3.3 RGMII interface

For communication with the gigabit Ethernet PHY, the RGMII interface may be used instead of the GMII interface. The only difference is that the signals are multiplexed by using both rising and falling edges of the clock. The individual signals are described in chapter 2.1.4.

In my design, this interface is implemented using the **rgmii_port_t** record with the following members:

- **data** is vector of 4 bits and corresponds to the **RXD** (receive data) or the **TXD** (transmit data) RGMII signal depending on the direction.
- **control** corresponds to the **RX_CTL** (receive control) or the **TX_CTL** (transmit control) RGMII signal depending on the direction.

3.3.4 XGMII interface

For communication with the ten gigabit Ethernet PHY, the standard XGMII interface should be used. The individual signals are described in chapter 2.1.6.

In my design, this interface is implemented using the **xgmii_port_t** record with the following members:

- **data** is vector of 64 bits and corresponds to the **RXD** (receive data) or the **TXD** (transmit data) XGMII signal depending on the direction.
- **control** is vector of 8 bits and corresponds to the **RXC** (receive control) or the **TXC** (transmit control) XGMII signal depending on the direction.

3.3.5 GE interface

With the exception of communication between MAC and PHY, this interface will be used for transporting gigabit Ethernet frames between entities.

Initial version

My first idea was to transport data and metadata separately. However, this was not very practical. It required two FIFOs instead of one FIFO in every entity which was buffering the frames and that was almost all of them.

The interface was implemented using a record with the following members:

- **data** of type **ge_data_t** transported the data.
- **data_valid** indicated whether the data was valid. It was therefore used to detect the start of frame and the end of frame conditions.
- **metadata** of type **ge_metadata_t** transported the metadata.
- **metadata_valid** indicated whether the metadata was valid.

The type **ge_data_t** was an alias for a vector of 8 bits. It corresponded to the GMII data signals but the preamble was removed on reception and added on transmission.

The type **ge_metadata_t** was a record with a total width of 64 bits. It included the count of bytes, count of errors, sequence number and timestamp of the Ethernet frame. Although it was convenient to include the length of the frame in the metadata, it was redundant and I decided to remove it later. The sequence number was included for pairing the timestamps of the transmitted frames in software but on reception, it was not used for anything.

The metadata was transported in the following clock cycle after the data transport was completed. I had to admit that there was no point in transporting 64 bits of metadata on the same clock cycle which required wide data path and using FIFOs as mentioned above.

Final version

In the second and final version of the gigabit Ethernet interface, I decided to transport metadata using the same signals as data. It was possible thanks to the inter frame gap which is guaranteed by the Ethernet specification to be at least 12 clock cycles long. The preamble will be also removed on reception and added on transmission, which gives 20 clock cycles in total available for transporting metadata.

I also decided to drop the valid signal and replace it with start of frame and end of frame signals. For designing state machines working with Ethernet frames, it is more convenient to get the signal one clock cycle before the event than to use the valid signal to control the state machine.

The metadata is transported in the so called footer which follows immediately after the frame. There is no start of footer signal as it would be equivalent to the end of frame signal. There is however the end of footer signal so the footer does not have constant length. In practice, the footer added on reception will always have 8 bytes to contain the timestamp and the footer added on transmission will always have 16 bytes to contain the identification and the timestamp. However, the length may change in future depending on the implemenation of the MAC.

It means that the interface basically transitions between three state: the idle state when nothing is transported, the frame state when data transports frame data and the footer state when data transports footer data. The state machine used for reception or transmission on this interface is shown in figure 3.4.

The interface is implemented using the **ge_port_t** record with the following members:

- **data** transports the frame data when the interface is in the frame state and the footer data when the interface is in the footer state.
- **start_of_frame** indicates the interface should transition to the frame state on the following clock cycle.
- **end_of_frame** marks the last byte of the frame and indicates the interface should transition to the footer state on the following clock cycle.
- **end_of_footer** marks the last byte of the footer and indicates the interface should transition to the idle state on the following clock cycle.

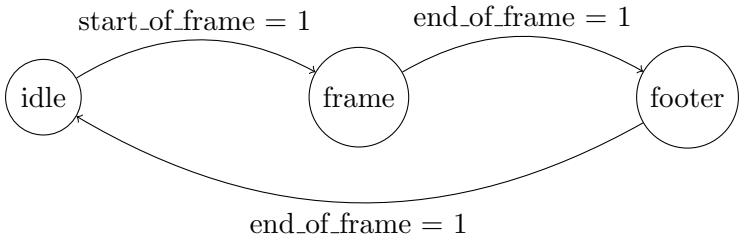

Figure 3.4: State machine of GE interface

3.3.6 XGE interface

With the exception of communication between MAC and PHY, this interface will be used for transporting ten gigabit Ethernet frames between entities.

Initial version

The first and final version of this interface is basically the initial version of the internal gigabit Ethernet interface with wider data path and no metadata signals. Kepping the same data and valid signals allowed simple conversion between gigabit Ethernet and ten gigabit Ethernet. The gigabit Ethernet interface was later changed so the conversion is no longer as simple as passing through a FIFO. No way of transporting metadata through this interface was considered because it was not required by any of the goals.

The interface is implemented using the **xge_port_t** record with the following members:

- **data** is vector of 64 bits divided into 8 lanes of 8 bits and transports the data.
- **valid** is vector of 8 bits and indicates whether the data is valid for each lane.

The frame always starts in the first lane so the start of frame detection is as simple as checking the valid bit of the first lane. The XGMII interface is originally only 32-bit wide and allows start of frame only in the first lane as well. That means in case of 64-bit wide interface, the frame may start in the fifth lane as well. When converting from XGMII to XGE interface, the frame must be aligned to always start in the first lane.

The frame may end in any of the eight lanes so end of frame detection must be performed by checking all eight valid bits. However, in cases when exact frame length does not have to be determined, it is sufficient to check the valid bit of the last lane. Thanks to the start of frame requirement, no more than one frame can be transported in one clock cycle.

3.4 Managing the components

Almost every entity will require a management interface. Some entities will include registers in order to configure different modes of operation. Some entities will include counters in order to check their functionality. It would be impractical to keep all of them in one entity so an interface for propagating read and write operations was needed.

The processing system provides several AXI interfaces for creating custom memory mapped peripherals. One option would be to implement AXI slave in each entity to be managed by the processing system. The AXI interconnect IP blocks would then be used to connect all of them to the processsing system. However, that would be really inefficient with tens of signals required by the AXI interface and hundreds of entities required by this project.

Just like in case of Ethernet interfaces, I decided to create custom interface for internal connections between entities. This interface will be very simple in order to simplify usage and also real device implementation. At the edge of the processing system and the programmable logic, there will be an entity converting the AXI interface to the custom management interface. I called the interface Conf by the word Configuration.

3.4.1 AXI interface

For interfacing between the processing system and the programmable logic, the AXI interface must be used. The individual signals are defined in the specification of the AXI interface [2]. In my design, this interface is implemented using two records to simplify connections.

The signals from master to slave are implemented using the **axi32_m2s_t** record for 32-bit data signals or the **axi64_m2s_t** record for 64-bit data signals.

The signals from slave to master are implemented using the **axi32_s2m_t** record for 32-bit data signals or the **axi64_s2m_t** record for 64-bit data signals.

3.4.2 Conf interface

This interface will be used for management of all entities from the processing system. In this subsection, *master* means the processing system and *slave* means the entity to be managed. The interface supports **read** and **write** transactions.

For read transactions, master puts the address it wants to read to the address signals and asserts the read signal at the same time. On the next rising edge of the clock, slave reads the data from the internal register and puts it to the data output signals.

For write transactions, master puts the address it wants to write to the address signals and asserts the write signal at the same time. It also puts the data to the data input signals. On the next rising edge of the clock, slave writes the data to the internal register.

Unlike in the AXI interface, there is no ready signal for reading or writing so every transaction must complete immediately.

The signals from master to slave are implemented using the **conf_m2s_t** record with the following members:

- **addr** is 32-bit vector representing the address to be read or written.
- **din** is 32-bit vector representing the data to be written to the internal register.
- **rd** starts a read transaction when it is high.
- **wr** starts a write transaction when it is high.

The signals from slave to master are implemented using the **conf_s2m_t** record with the following members:

- **dout** is 32-bit vector representing the data that was read from the internal register.
- **err** indicates there was an error. It is not used at the moment.

3.5 Converting RGMII to GMII

The first step was to convert the RGMII interface described in section 3.3.3 to the GMII interface described in section 3.3.2.

3.5.1 Requirements

The goal is to create entities for interfacing between the PHY which uses the RGMII interface and the MAC which uses the GMII interface. Both interfaces use 125 MHz clock with 100 ppm toleration. The RGMII interface uses DDR (double data rate) signaling with 4 data signals and 1 control signal. The GMII interface uses SDR (single data rate) signaling with 8 data signals and 2 control signals. Collision signals are not considered because only full duplex communication will be supported.

3.5.2 Solutions

Three possible approaches were considered: using simple registers, using special primitives and using dedicated IP blocks. All of them were perfectly functional in simulation but when implementing them on real device, timing problems appeared.

Using simple registers

Converting one signal from DDR to SDR is possible with four registers and the following process:

1. The first register captures the input on the rising edge of the clock.
2. The second register captures the input on the falling edge of the clock.
3. Data is shifted from the first pair of registers to the second pair of registers on the next rising edge of the clock.

For converting from RGMII to GMII interface, there are 5 bits converted to 10 bits in total, so the above process would be instantiated five times.

Converting one signal from SDR to DDR is possible by building a multiplexer controlled by the clock signal as follows:

- When clock is high, put the rising edge input to the output.
- When clock is low, put the falling edge input to the output.

For converting from GMII to RGMII interface, there are 10 bits converted to 5 bits in total, so the above multiplexer would be instantiated five times.

The advantage of this solution is that it does not use any vendor specific primitives so it should be portable to other devices with no problem. Unfortunately, the timing constraints were violated when implemented on real device. On reception (conversion from DDR to SDR), the problem only appeared sometimes depending on circumstances. On transmission (conversion from SDR to DDR), the problem appeared always because using the clock as the control signal for the multiplexer was not able to meet the timing constraints.

Currently, this approach is used for reception in the RGMII receiver where it brings better results than the other solution.

Using special primitives

Xilinx provides special primitives for interfacing with DDR signals: Input Double Data Rate (IDDR) and Output Double Data Rate (ODDR). More information on them can be found in UltraScale Architecture SelectIO Resources user guide [19].

Those primitives allow converting one bit of signal from DDR to SDR in case of IDDR and from SDR to DDR in case of ODDR. For converting between RGMII and GMII interfaces, five primitives should be instantiated.

It is also recommended to use the ODDR primitive to generate the output clock by tying the rising edge data to 1 and the falling edge data to 0 due to better performance.

The advantage of this solution is that thanks to using the recommended primitives, there should be no problem meeting the timing constraints. When implemented on real device, it was not always that case unfortunately. On reception (conversion from DDR to SDR), there was always a slack of 0.25 ns. Later, I found out it was given by the fact that the clock path was always much longer than the data path due to the design of the chip. On transmission (conversion from SDR to DDR), there was no problem when using the primitive to generate the output clock according to the previous paragraph.

Currently, this approach is used for transmission in the RGMII transmitter where it brings better results than the other solution.

Using dedicated IP block

Xilinx provides dedicated IP block for converting GMII to RGMII. More information on it can be found in GMII to RGMII product guide [5].

However, this block seemed too complex compared to the solutions mentioned above. One of the reasons could be that it supports switching the speed of the interface according to the current mode of the PHY which is determined via the management interface. Unfortunately, it is not possible to configure this block only for gigabit Ethernet to simplify the integration.

3.5.3 RGMII receiver

The RGMII receiver is implemented in the **ge_rgmii_rx** entity. It converts the RGMII interface provided by the **rgmii_port** input to the GMII interface provided by the **gmii_port** output. Both interfaces are synchronous to the **rgmii_clk** input. There is also the **gmii_clk** output which is tied to the **rgmii_clk** input at the moment.

Figure 3.5: Usage of RGMII receiver

3.5.4 RGMII transmitter

The RGMII transmitter is implemented in the **ge_rgmii_tx** entity. It converts the GMII interface provided by the **gmii_port** input to the RGMII interface provided by the **rgmii_port** output. The GMII interface is synchronous to the **gmii_clk** input. The RGMII interface is synchronous to the **rgmii_clk** output which is generated by this entity.

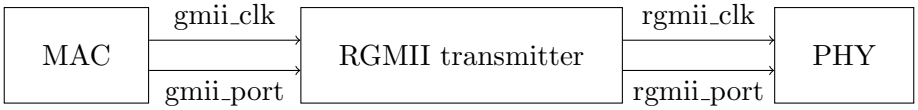

Figure 3.6: Usage of RGMII transmitter

3.5.5 PHY block

The PHY block is implemented in the **ge_phy** entity. It combines the following components:

- RGMII receiver for conversion from RGMII to GMII.
- RGMII transmitter for conversion from GMII to RGMII.
- SMI controller for management of the external PHY.

Verification

The PHY block was verified in simulation by connecting the GMII interfaces of the receiver and the transmitter together and generating and checking Ethernet frames on the RGMII interfaces.

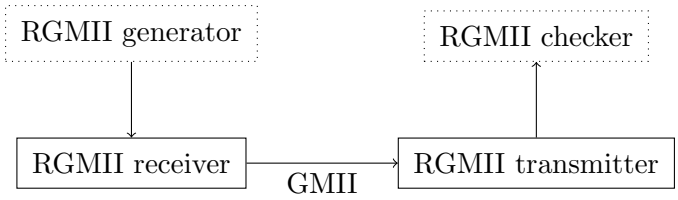

Figure 3.7: Verification of RGMII in simulation

Validation

The PHY block was validated on real device by connecting the GMII interfaces of the receiver and the transmitter together and connecting the RGMII interfaces to the RGMII interfaces of the PHYs on the board.

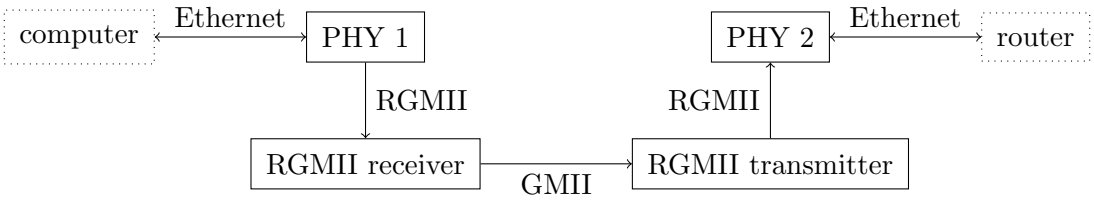

Figure 3.8: Validation of RGMII on real device

3.6 Receiving and transmitting GE

As explained in section 3.3, the GMII interface was not suitable for transporting Ethernet frames between entities. For that purpose, the GE interface was defined and this section shows the design of the entities used for conversion between the GMII interface and the GE interface.

3.6.1 Requirements

The goal is to create entities for interfacing between the PHY which uses the external (GMII) interface and the rest of the system which will use the internal (GE) interface. The entities should support not only the gigabit Ethernet according to the standard, but also the 100 Mb/s Ethernet and possibly the 10 Mb/s Ethernet. When the gigabit Ethernet PHY supports slower Ethernet speeds, it switches the GMII interface to the MII interface as needed.

The solution should also support timestamping of both the received and the transmitted frames for two reasons. First, timestamps are required when using PTP protocol for time synchronization. Second, timestamps are included in frame metadata when using CMP protocol for logging. Timestamps will be transported via the GE interface using the footer as explained in section 3.3.5.

3.6.2 Solutions

The first version of the MAC supports only gigabit Ethernet. It uses the same clock on the GMII side and on the GE side. The second version of the MAC is more complex and supports slower Ethernet speeds which use 4-bit data path instead of 8-bit data path. It uses separate clock on the GMII side and on the GE side which is required due to reasons explained below.

Single speed

The single speed solution is very simple. There is only one clock domain involved so no clock domain crossing is needed. The main goals of the MAC are to add or remove the preamble and to save the timestamp when the frame was received or transmitted.

On reception, the MAC has to detect the start of frame on the GMII interface. It passes the data from the GMII interface after the start of frame delimiter to the GE interface without any other changes. Finally, it generates the GE footer including the timestamp on the GE interface.

On transmission, the MAC has to generate the preamble on the GMII interface. It passes the data from the GE interface to the GMII interface after the start of frame delimiter without any other changes. Finally, it generates the GE footer including the timestamp on the output GE interface.

Dual speed

The dual speed solution required significant changes in the topology of the device. The main goal was to use the same GE interface for 1 Gb/s Ethernet and also for 100 Mb/s or 10 Mb/s Ethernet. Thanks to that, the rest of the components designed for gigabit Ethernet could be also used for slower Ethernet speeds without any changes. Only the clock frequency would be divided by 10 or 100 to match the Ethernet speed.

On the external side of the MAC, the interface is different unfortunately. When used with gigabit Ethernet, the GMII interface uses 8 bit data path according to the specification. When used with slower Ethernet speeds, the GMII interface basically changes to the MII interface and uses only 4 bit data path. For that reason, the frequency of the GMII interface in 1 Gb/s mode is 125 MHz, while the frequency is 25 Mhz in 100 Mb/s mode and 2.5 MHz in 10 Mb/s mode. That is different from the frequency used by the GE interface that will be 12.5 MHz in 100 Mb/s mode and 1.25 MHz in 10 Mb/s mode according to the previous paragraph.

The result is that the MAC must be able to work in two independent clock domains. As always, the clock domain crossing will be performed by using a FIFO with independent clock domains as described in section 3.2. The MAC will be divided into two separate processes. The first process will handle the reception or transmission on the GMII interface and will be driven by the GMII clock. The second process will handle the reception or transmission on the GE interface and will be driven by the GE clock. The MAC will support two different modes of operation. The first mode (called *full mode* in the following sections) will use all 8 bits of the GMII interface to support 1 Gb/s Ethernet. The second mode (called *half mode* in the following sections) will use only 4 bits of the GMII interface to support 100 Mb/s and 10 Mb/s Ethernet.

3.6.3 MAC receiver

The MAC receiver is implemented in the **ge_mac_rx3** entity. It receives Ethernet frames from the GMII interface provided by the **gmii_input** and transmits them to the GE interface provided by the **ge_output**. The GMII interface is synchronous to the **gmii_clk** clock. The GE interface is synchronous to the **ge_clk** clock.

As the GMII and GE clocks are usually asynchronous, the MAC receiver contains an independent clock FIFO for performing the clock domain crossing. It removes the preamble and the SFD (Start of Frame Delimiter) from the frame before passing it to the GE interface. It also adds the footer after the frame is transmitted on the GE interface. The footer consists of the timestamp which is captured from the **timestamp** input when the SFD is received on the GMII interface.

The MAC receiver also provides the management interface described in section 3.4.2 for changing operation mode and reading counters.

Structure

The MAC receiver consists of the following components:

- The **FIFO** component which passes frames from the GMII process to the GE process. It uses the **fifo8x** entity with memory promitive as described in section 3.2.1.
- The **GMII** process which receives frames from the GMII interface and writes them to the FIFO. It is driven by the **gmii_clk** clock and the **gmii_rst** reset.
- The **GE** process which reads frames from the FIFO and transmits them to the GE interface. It is driven by the **ge_clk** clock and the **ge_rst** reset.

The structure is also shown in figure 3.9.

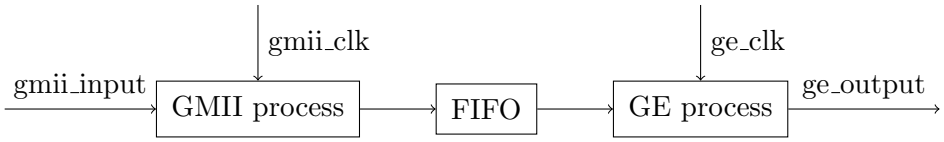

Figure 3.9: Structure of MAC receiver

Management

The MAC receiver is managed by the standard management interface.
The following registers are available for reading:

- The **status** register (0x00) provides information about the current status.
  - Bit 0 reports whether the receiver is enabled.
- The **cnt_frames** register (0x10) counts the frames received when counters were active.
- The **cnt_bytes** register (0x14) counts the bytes received when counters were active.
- The **cnt_errors** register (0x18) counts the bytes received with an error indicated by the PHY when counters were active.

The following registers are available for writing:

- The **control** register (0x00) allows configuring the receiver.
  - Bit 0 disables the receiver when set to 1.
  - Bit 1 enables the receiver when set to 1.
  - Bit 2 activates the counters when set to 1. Frames received after activation influence the values of the counters.
  - Bit 3 deactivates the counters when set to 1. Frames received after deactivation do not influence the values of the counters.
  - Bit 4 resets the counters when set to 1. The values of the counters are set to zeros.
  - Bit 6 selects the full mode when set to 1. The full mode uses 8-bit GMII data path and is used for 1 Gb/s Ethernet.
  - Bit 7 selects the half mode when set to 1. The half mode uses 4-bit MII data path and is used for 100 Mb/s and 10 Mb/s Ethernet.

3.6.4 MAC transmitter

The MAC transmitter is implemented in the **ge_mac_tx3** entity. It receives Ethernet frames from the GE interface provided by the **ge_input** and transmits them to the GMII interface provided by the **gmii_output** and also to the feedback GE interface provided by the **ge_output** with another GE footer added. The GE interfaces are synchronous to the **ge_clk** clock. The GMII interface is synchronous to the **gmii_clk** clock.

As the GE and GMII clocks may be asynchronous in some cases, the MAC transmitter contains an independent clock FIFO for performing the clock domain crossing. It generates the preamble and the SFD (Start of Frame Delimiter) before passing the frame from the GE interface to the GMII interface. It also outputs the frame to the feedback GE interface with another GE footer added. The footer consists of the timestamp which is captured from the **timestamp** input when the SFD is transmitted on the GMII interface.

The MAC transmitter also provides the management interface described in section 3.4.2 for changing operation mode and reading counters.

Structure

The MAC transmitter consists of the following components:

- The **FIFO** component which passes frames from the GE process to the GMII process. It uses the **fifo8x** entity with memory primitive as described in section 3.2.1.
- The **GE** process which receives frames from the GE interface and writes them to the FIFO. It is driven by the **ge_clk** clock and the **ge_rst** reset. It also transmits the frame to the feedback GE interface with another GE footer added. The timestamp included in the footer is received from the GMII process.
- The **GMII** process which reads frames from the FIFO and transmits them to the GMII interface. It is driven by the **gmii_clk** clock and the **gmii_rst** reset. It also generates the 8 bytes of the preamble with the SFD (Start of Frame Delimiter) before the frame and the 12 bytes of the IPG (Inter-Packet Gap) after the frame. It captures the timestamp when the SFD is transmitted and passes it to the GE process.

The structure is also shown in figure 3.10.

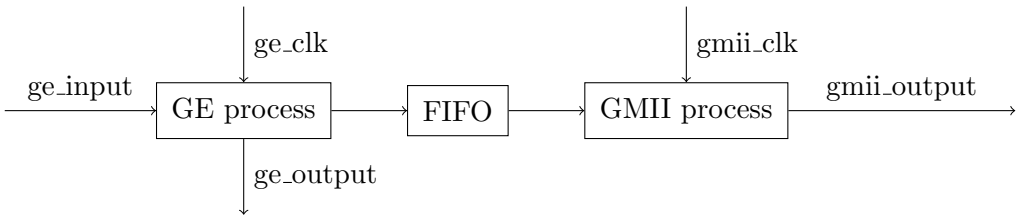

Figure 3.10: Structure of MAC transmitter

Management

The MAC transmitter is managed by the standard management interface.

The following registers are available for reading:

- The **status** register (0x00) provides information about the current status.
  - Bit 0 reports whether the transmitter is enabled.
- The **cnt_frames** register (0x10) counts the frames transmitted when counters were active.

- The **cnt_bytes** register (0x14) counts the bytes transmitted when counters were active.

The following registers are available for writing:

- The **control** register (0x00) allows configuring the transmitter.
  - Bit 0 disables the transmitter when set to 1.
  - Bit 1 enables the transmitter when set to 1.
  - Bit 2 activates the counters when set to 1. Frames transmitted after activation influence the values of the counters.
  - Bit 3 deactivates the counters when set to 1. Frames transmitted after deactivation do not influence the values of the counters.
  - Bit 4 resets the counters when set to 1. The values of the counters are set to zeros.
  - Bit 6 selects the full mode when set to 1. The full mode uses 8-bit GMII data path and is used for 1 Gb/s Ethernet.
  - Bit 7 selects the half mode when set to 1. The half mode uses 4-bit MII data path and is used for 100 Mb/s and 10 Mb/s Ethernet.

3.6.5 MAC block

The MAC block is implemented in the **ge_mac2** entity. It combines the following components:

- MAC receiver for receiving GE frames from GMII interface.
- MAC transmitter for transmitting GE frames to GMII interface.

Integration

The structure and typical usage of the MAC block is shown in figure 3.11.

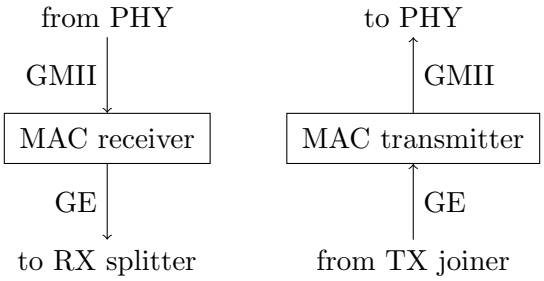

Figure 3.11: Usage of MAC block

Verification

The MAC block was verified in simulation as described in section 4.4.1.

Validation

The MAC block was validated on real device later as part of the passthrough test described in section 8.1 and the loopback test described in section 8.2.

3.7 Filtering and splitting GE stream

Once we had the GE frames received from the PHY and converted to the GE interface, the next step was to build components for filtering the frames in hardware. The need for implementing the filter in hardware was given by the high amount and high speed of Ethernet interfaces planned for the final device. If we are able to implement eight Ethernet ports as planned, the total throughput will be 16 Gb/s in the theoretical case when all the ports run in full duplex mode and their capacity is fully utilized. That is far beyond the total power of the processing system available on the chip.

3.7.1 Requirements

Each frame received on one of the GE interfaces should be examined and treated according to its headers. Some frames should be logged via the CMP interface. Some frames should be passed to the paired GE interface. Some frames should be processed by the processing system. Most frames will have multiple actions applied like logging and passing the frame at the same time. The following criteria including their combinations should be used for decision:

- Link layer header (always Ethernet)
  - Destination MAC address (6 bytes)
  - Source MAC address (6 bytes)
  - Ethernet type (2 bytes)
- Network layer header (mostly IPv6)
  - Next header (1 byte)
  - Source IP address (16 bytes)
  - Destination IP address (16 bytes)
- Transport layer header (mostly UDP)
  - Source port (2 bytes)
  - Destination port (2 bytes)

Each frame transmitted on one of the GE interfaces should be examined as well. Some frames should be logged via the CMP interface, especially those created by the processing system which would be otherwise missing in the log. Some frames should be processed by the processing system, especially those for which TX timestamp is required.

3.7.2 Solutions

During the development, three different solutions were implemented. The first idea was to create specific filters for each protocol layer separately. That was simple for the link layer which is always Ethernet in our case. However, it would be more difficult to support more different network layers or transport layers because their headers are very different. The second idea which came from my supervisor was to make one generic filter which would support any protocol up to predefined header length. The third idea changed filter to splitter to save resources and simplify configuration.

Specific filters

The idea was to create an array of matchers which would all examine the frame coming to the input and assert a signal if there was a match. Each matcher would handle one specific protocol. That would require creating at least the following matchers:

- MAC matcher for matching the link layer header against configured source and destination addresses and Ethernet types.
- IPv4 and IPv6 matchers for matching the network layer header against configured source and destination addresses.
- TCP and UDP (and potentially more) matchers for matching the transport layer header against configured source and destination ports.

There is however a catch. Due to various reasons, the network layer headers and the transport layer headers are not always at the same position. Here are a few examples:

- Link layer header may include the 802.1Q tag with VLAN identifier which adds 4 more bytes to the header and thus shifts upper layer headers.
- Network layer header differs depending on the protocol that is used. The basic IPv4 header has 20 bytes. The basic IPv6 header has 40 bytes.
- Network layer header may be extended by adding options to the header and thus shifting upper layer headers.

In order to support all possible combinations of the mentioned options, each matcher would have to examine both the layer it was intended for and the lower layers so that it knows where the header it is waiting for starts.

Moreover, it was not clear how the outputs of specific matchers should be put together in order to support any possible combination of filters.

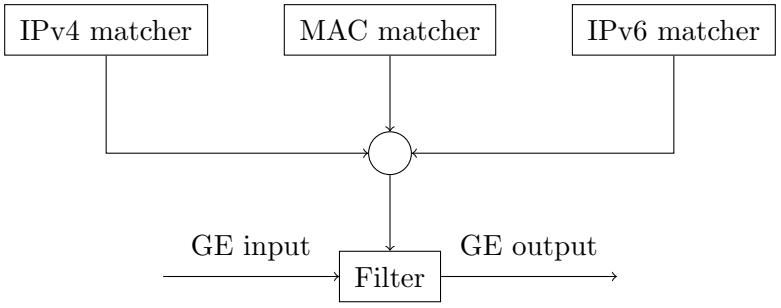

Figure 3.12: Structure of specific filter

Generic filters

To simplify the design, a generic filter supporting virtually any protocol could be created. It would take each frame as a sequence of bytes and bits and match it against the following arrays:

- The **value** array where the bits corresponding to the header fields used for the filtering would be set to the required values. Other bits would be insignificant.
- The **mask** array where the bits corresponding to the header fields used for the filtering would be set to ones. Other bits would be set to zeros.

The arrays would be configured by software depending on the current needs. It would be possible to set up filtering by the link layer, the network layer, the transport layer or any possible combination of them.

The length of both arrays would be set in order to fit all common protocols up to the transport layer header. In the worst case, the header lengths are as follows:

- The link layer header is 18 bytes long when it includes the 802.1Q tag.
- The network layer header is 40 bytes long in case of IPv6 with no additional options.
- The transport layer header is 20 bytes long in case of TCP with no additional options.

That means the arrays should have at least 80 bytes to support filtering by the second, the third and the fourth layer. In practice, it probably will not be necessary to filter by anything else than the port numbers on the fourth layer. So I will set the default length of the arrays to 64 bytes which should be enough.

This pair of arrays would be called a matcher. One matcher would correspond to one type of frames which could be either passed or dropped by the filter. For example, there could be a matcher set up to match one specific MAC address or all multicast addresses. As another example, there could be a matcher set up to match a combination of source and destination IP addresses and source and destination UDP ports at the same time.

There would be many matchers connected to the filter which could work in one of the following operating modes:

- It would pass all frames or drop all frames irrespective of the content. In this case, the filter would be called **disabled**.
- It would **pass** the frame if and only if at least one of the matchers reported a match. In other cases, specifically when all matchers reported a mismatch or no decision had been made due to the frame being too short, the filter would **drop** the frame. This mode would be therefore called **whitelist** mode.
- It would **drop** the frame if and only if at least one of the matchers reported a match. In other cases, specifically when all matchers reported a mismatch or no decision had been made due to the frame being too short, the filter would **pass** the frame. This mode would be therefore called **blacklist** mode.

The filter itself would contain a FIFO to save the frame before the decision was made. If a decision was made to **pass** the frame, the filter would then read the frame from the FIFO and transmit it on the output interface. If a decision was made to **drop** the frame, the filter would then read the frame from the FIFO as well but it would not transmit it anywhere.

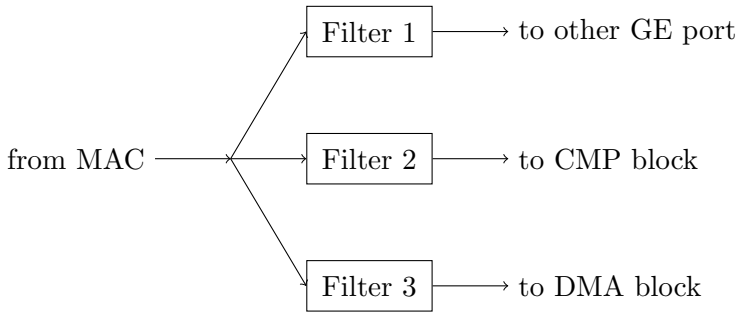

Figure 3.13: Usage of generic filter

Generic splitter

To simplify the design even more, a single generic splitter with multiple outputs could replace multiple generic filters with single output. This change of topology will provide some advantages:

- There will be a single FIFO instead of multiple FIFOs for buffering the frame before decision is made. That means it will require less memory resources.
- The matchers will be common for all outputs from one input. That will avoid having multiple matchers doing the same job in cases when they would have the same configuration. The saved resources can be used to implement more matchers or increase their length.
- The matchers will be configured together for all outputs from one input. That will simplify programming and also help to avoid mistakes.

The matcher part will remain the same as in the generic filter approach. There will be a fixed amount of matchers whose signals will be evaluated as described in the previous subsection.

The splitter part will add an action register for each matcher to specify which outputs should the frame go to in case of a match. The action register will consist of one bit for each output controlling whether the frame should appear on that output or not. There will also be a default action register to specify the action for cases when none of the matchers report a match.

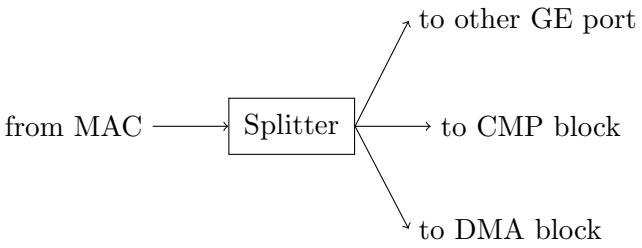

Figure 3.14: Usage of generic splitter

3.7.3 Matcher

The matcher is implemented in the **ge_matcher** entity. It matches Ethernet frames from the **ge_input** against configured value and mask arrays. It contains two memories accessible from the management interface for the value and mask arrays. The size of the memories is given by the **max_length** generic parameter.

In addition to configuring the memories, it is necessary to enable the matcher and set the length of the matcher. When the matcher is enabled, one of the following happens for each frame:

1. Each byte of the frame between the first byte and the Nth byte, where N is the length of the matcher, does match. Then the **match** output is asserted after the last matching byte is present on the input.
2. At least one byte of the frame between the first byte and the Nth byte, where N is the length of the matcher, does not match. Then the **mismatch** output is asserted after the first mismatching byte is present on the input.
3. None of the previous happens because the frame is shorter than the matcher. In that case, none of the outputs is asserted.

Both match and mismatch outputs are deasserted when the frame which caused their assertion ends. Assertion of both signals at the same time is not possible.

Structure

The matcher consists of the match process driven by the **ge_clk** clock which transitions between the following states:

- The **idle** state corresponds to the idle state of the GE input interface. On the start of frame and when enabled, the matcher transitions to the listen state.
- The **listen** state indicates that no decision has been made yet and the matcher is examining the frame. When decision is made, the matcher transitions either to the match state or to the mismatch state.
- The **match** state indicates that the current frame does match. On the end of frame, the matcher transitions to the idle state.
- The **mismatch** state indicates that the current frame does not match. On the end of frame, the matcher transitions to the idle state.

The matcher also contains the value RAM and the mask RAM which are optimized to be implemented using the LUT RAM.

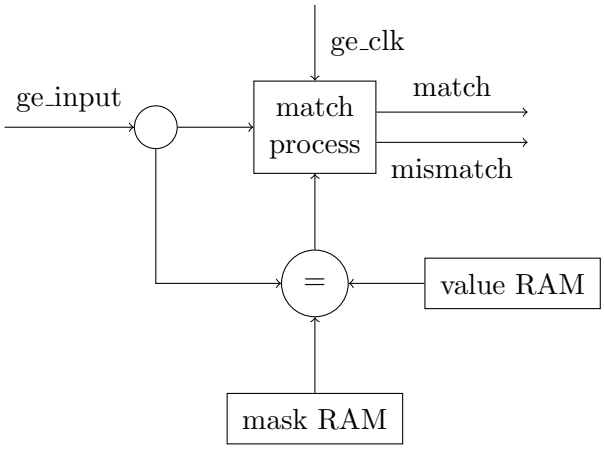

Figure 3.15: Structure of matcher

Management

The matcher is managed by the standard management interface.

The following registers are available for reading and writing:

- The **control** register (0x00) allows enabling and disabling the matcher:
  - Bit 0 enables the matcher when set to 1 and disables the matcher when set to 0.
- The **length** register (0x04) allows setting the length of the matcher. The minimal length is 1. The maximal length is given by max_length parameter.

The following memories are available for writing only:

- The **value** memory (0x100-0x1FF) serves as the value array of the matcher. Byte access is possible to this memory. The actual size of the memory is given by the max_length parameter.
- The **mask** memory (0x200-0x2FF) serves as the mask array of the matcher. Byte access is possible to this memory. The actual size of the memory is given by the max_length parameter.

3.7.4 Splitter

The splitter is implemented in the **ge_splitter** entity. It examines each frame from the input and passes it to zero, one or more outputs depending on the results of the embedded matchers. Each matcher is connected to an action which specifies the outputs where the frame should go in case of a match.

The splitter has the following generic parameters:

- The **output_count** parameter sets the number of outputs from the splitter. It also influences the size of each action register as that reflects the output count.
- The **matcher_count** parameter sets the count of the matchers to be instantiated inside the splitter.
- The **matcher_length** parameter sets the length of the matchers to be instantiated inside the splitter.

Structure

The splitter consists of the following components:

- The **Matchers** which are instantiated according to the matcher_count generic parameter. They examine the frame according to their settings and provide match and mismatch signals to the Input Process.
- The **Input Buffer** which holds the frame received from the input port until decision is made by the Input Process.
- The **Input Process** which selects the action according to the match and mismatch signals from the Matchers and writes it to the Action FIFO. It is driven by the **ge_clk** clock. It transitions between the following states:
  - The **idle** state when the splitter is waiting for the frame. It transitions to the **listen** state when the start of frame signal is asserted.
  - The **listen** state when the splitter is waiting for the the signals from the Matchers. It transitions to the **idle** state when either some of the match signals is asserted or all of the mismatch signals are asserted or the end of frame signal is asserted. In the first case, the action corresponding to that Matcher is written to the Action FIFO. In other cases, the default action is written to the Action FIFO.
- The **Action FIFO** which passes the selected action from the Input Process to the Output Process.
- The **Output Process** which reads the action from the Action FIFO and controls the Demultiplexer accordingly. It is driven by the **ge_clk** clock.
- The **Demultiplexer** which connects the output of the Input Buffer to zero, one or more output ports according to the current action. It is controlled by the Output Process.

The structure is also shown in figure 3.16.

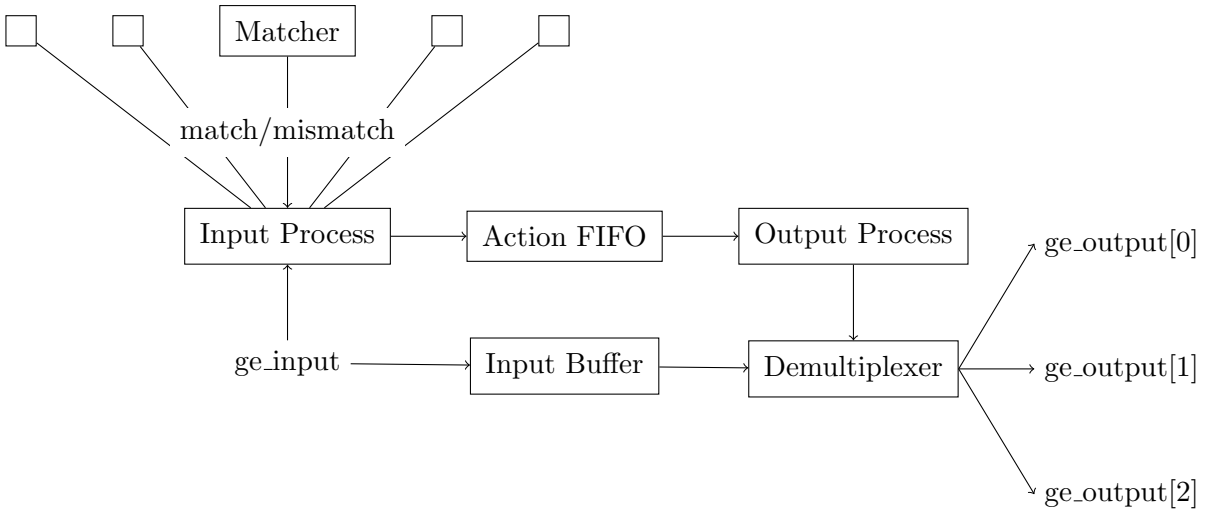

Figure 3.16: Structure of splitter

Management

The splitter is managed by the standard management interface. It provides access to the following components:

- The **Splitter** (see below) is mapped from **0x0000** to 0x03FF. It contains multiple sets of registers depending on the count of the matchers.
- The **Matchers** (see section 3.7.3) are mapped from **0x0400** to 0x7FFF. A region of 0x400 is assigned to every matcher up to the count of the matchers.
  - The 1st Matcher is mapped from **0x0400** to 0x07FF.
  - The 2nd Matcher is mapped from **0x0800** to 0x0BFF.
  - The 3rd Matcher is mapped from **0x0C00** to 0x0FFF.
  - ...

There is always one set of registers with index 0 for managing the splitter itself. The following registers are available for reading and writing in that set:

- The **control** register (0x00) allows enabling and disabling the splitter.
  - Bit 0 enables the splitter when set to 1 and disables the splitter when set to 0.
- The **action** register (0x04) sets the default action of the splitter. It is applied when all matchers report a mismatch or no matcher reports a match before the frame ends.

Then there are N sets of registers with indexes 1 to N for managing the matchers where N equals the count of the matchers. The following registers are available for reading and writing in each set:

- The **control** register (0x10, 0x20...) allows enabling and disabling the matcher from the perspective of the splitter. The matcher itself still needs to be enabled separately.
  - Bit 0 enables the matcher when set to 1 and disables the matcher when set to 0.
- The **action** register (0x14, 0x24...) sets the action of the matcher. It is applied when the matcher reports a match.
  - Bit N determines whether the frame is passed to output N where N is between 0 and output_count - 1.

Verification

The splitter was verified in simulation as described in section 4.4.4.

Validation

The splitter was validated on real device as part of the passthrough test described in section 8.1.

3.8 Processing GE stream

One of the greatest advantages our device will have compared to commercial solutions is the ability to process frames in real time. The first level of the processing is the filtering described in the previous section. The second level of the processing will be done by software in order to make it highly flexible.

One example which will be used for testing the software processing is handling the PTP frames. As mentioned earliner, our device will have to actively participate and respond to some of the PTP messages in order to allow accurate time synchronization across the device.

3.8.1 Requirements

To make the software processing possible, it is necessary to transport selected frames between hardware and software. The goal is therefore to transport the frames from our custom hardware peripheral to the memory accessible by the processing system and vice versa.

There will be multiple gigabit Ethernet ports on our device so the transport should be as efficient as possible so that the introduced delay is the lowest possible and the total throughput is the highest possible.

It is also necessary to transport the hardware timestamp for each frame received or transmitted. At the moment, the timestamp will be used only by the PTP protocol, but it should be possible to get timestamp for any frame.

3.8.2 Solutions

Using CPU to transport data

The simplest solution would be to make the CPU do all the job and transport the frames between hardware and software word by word. From the hardware perspective, it would require building an entity with AXI slave interface on one side and FIFO interface on the other side. It would probably be possible to find such an entity in the Vivado IP catalog. That is why I consider this solution the simplest.

The advantage of this solution is that there would be no problem with caching. Just like every modern processor, the CPU on our chip contains an instruction cache and a data cache for each core. The data cache works well when the core is the only device to use the memory. However, when other devices start pushing data into the memory, the data cache must be either disabled for that memory region or invalidated each time the content of that memory region changes.

The disadvantage of this solution is that it is probably going to be slower than other solutions which use DMA (direct memory access). It does not make sense to use the CPU for transporting data between two different memories. It would be better to use the CPU only for processing the frame once it had been transported to the memory by another entity, some kind of DMA controller.

Using DMA controller to transport data

The better solution would be to use the DMA controller available on our chip for transporting the frames between the custom hardware peripheral located in the programmable logic and the main memory located in the processing system. From the hadrware perspective, it would require building an entity with AXI slave interface just like in the previous solution. From the software perspective, it would require studying the manual of the DMA controller and configuring it to make the transport when needed for reception or transmission.

The advantage of this solution is that the CPU will no longer spend clock cycles transporting data between peripherals so the whole processing power will be available to the application. Another advantage of using the standard DMA controller could be that the software driver for this controller is already created by someone else and also the controller hardware itself is already verified by someone else.

The disadvantage of this solution is that the cache will have to be managed so that the CPU does not work with invalid data. As mentioned earlier, one of the solutions to this problem is to invalidate the cache after every transfer. Another disadvantage is that the DMA controller will not be available for other uses which could become a problem when switching from bare metal to Linux environment.

However, the main reason for not going this way was different. It was not clear how to control the DMA controller from hardware. When transmitting Ethernet frames, the DMA transfer is started by software once the frame is ready in the main memory. When receiving Ethernet frames, the DMA transfer should be started by hadrware once the frame is ready in the peripheral.

One solution would be to interrupt the processor which would then setup the DMA transfer from the peripheral. That would be inefficient and possibly slower than using the CPU to transport the whole frame though. The correct solution would be to somehow start the transfer directly from the peripheral. As pointed out by my supervisor, the DMA controller probably supports that by using so called credits. However, I decided to try developing a custom DMA controller which could be integrated into the rest of the design.

Developing custom DMA controller

The most complex solution would be to build a custom DMA controller in the programmable logic for transporting the frames between the custom hardware peripheral located there and the main memory located in the processing system. From the hardware perspective, it would require building an entity with AXI master interface on one side and the FIFO interface on the other side. It would probably be possible to find such an entity in the Vivado IP catalog. However, I wanted to try building a custom solution before using a proprietary one.

The advantage of this solution is that the whole processing power will be available to the application like in the previous solution. Another advantage of using a custom DMA controller is that the standard DMA controller will be fully available to the operating system and the custom DMA controller can be fully customized according to the requirements of our application.

The disadvantage of this solution is that the cache will have to be managed like in the previous solution. Another disadvantage of developing a custom DMA controller is that it will probably require more time than the other solutions.

The idea was to use part of the main memory as a ring buffer. The frames would be stored there one after another and when the end of the buffer was reached, the frame would wrap to the start of the buffer. This design had a significant disadvantage I was not able to see at first. When the frame was located in the middle of the buffer, a structure in C could be conveniently mapped to the frame in order to extract individual fields of the frame. When the frame wrapped from the end to the start of the buffer, that would not be possible anymore. In fact, it would require copying the frame to another part of memory in order to assemble the whole frame.

This problem is usually solved by allocating an array of buffers where each buffer may contain only one frame and the size of the buffer is given by the maximal length of the frame. I did not like this solution because most frames we will be transporting to the main memory will be much shorter than the maximal length of the frame is. So I decided to modify the solution with a ring buffer to solve the wrapping problem. I added a rule to return to the start of the buffer when the frame to receive or transmit is too long to fit between the current position in the buffer and the end of the buffer.

Another thing to solve was how to transport the timestamps which were part of metadata in the first version of the design and part of footer in the second version of the design as described in section 3.3.5. In the first version, the metadata used a separate ring buffer in the main memory. For each frame, there was a separate data transport and a separate metadata transport, both done by the DMA controller. The metadata also included the length of the frame. Both hardware and software maintained its own pointer into the data buffer and into the metadata buffer. In the second version, the footer was stored after the frame in the same buffer. Only one transfer was required which transported both the frame and the footer.

That brought another thing to solve which was how to transport the length of the frame and also synchronize the offset into the buffer. In the first version, it often happened that the two pointers into the data buffer used by hardware and software were not properly synchronized. It could have been caused by an error in the implementation or a bit error during clock domain crossing which should however happen very rarely. In the second version, I used a much better and reliable approach. I created a simple descriptor with the length of the frame and the offset into the buffer. This descriptor would be written to the controller for each transmitted frame and read from the controller for each received frame. In case of any error, it would only compromise one frame because the next frame would also include the offset for proper synchronization.

3.8.3 DMA receiver

The DMA receiver is implemented in the **ge_dma_rx2** entity. It receives Ethernet frames from the GE interface provided by the **ge_port** input and transports them to external memory by becoming the master of the AXI interface provided by the **axi_m2s** and **axi_s2m** signals.

The GE interface is synchronous to the **ge_clk** clock which is expected to run at 125 MHz (or 12.5 MHz for 100 Mb/s Ethernet). The AXI interface is synchronous to the **axi_clk** clock which is expected to run at 250 MHz (or any other frequency). The DMA receiver must therefore perform clock domain crossing between GE and AXI clock domains.

The location of the buffer in the memory is given by the **buffer_start** and **buffer_size** registers. The DMA receiver keeps an internal pointer into the buffer. When the frame to be transported to the memory is short enough to fit between the current position in the buffer and the end of the buffer, the frame is written to the current position in the buffer. When the frame to be transported to the memory is too long to fit between the current position in the buffer and the end of the buffer, the current position in the buffer is reset so that the frame is written to the start of the buffer.

Structure

The DMA receiver consists of the following components:

- The **GE process** which receives frames from the GE interface and writes them to the **data FIFO**. It aligns the count of the bytes written to the FIFO to the nearest multiple of eight to allow the data width conversion to be done in the FIFO. It also writes the computed length of the frame to the **metadata FIFO** for next processing. It is driven by the **ge_clk** clock.

- The **data FIFO** which holds the frames received by the GE process to be transported to the external memory by the AXI process.
- The **metadata FIFO** which holds the length of the frames received by the GE process to be used by the AXI process.
- The **AXI process** which reads frames from the **data FIFO** and transports them to the external memory. It keeps the current offset into the data buffer. For each frame, it decides whether it is going to fit between the current offset and the end of the data buffer. If not, the current offset is reset to zero. Once the frame is transported, it builds a descriptor for the frame and writes it into the **descriptor FIFO**. It is driven by the **axi_clk** clock.
- The **descriptor FIFO** which transports descriptors from the AXI process to the management interface. Each descriptor describes one frame that was transported from the data FIFO to the external memory. The descriptor consists of the length of the frame and the offset to be added to the base address of the data buffer in order to get the start address of the frame.

The structure is also shown in figure 3.17.

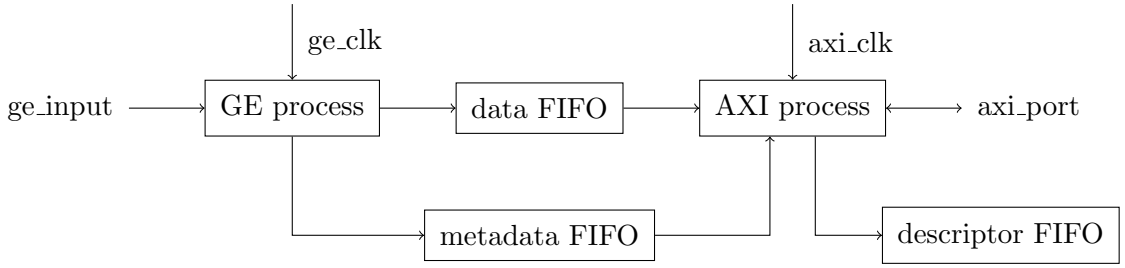

Figure 3.17: Structure of DMA receiver

Management

The DMA receiver is managed by the standard management interface.

The following registers are available for writing:

- The **control** register (0x00) allows configuration of the receiver:
  - Bit 0 disables the receiver when set to 0 and enables it when set to 1.
  - Bit 1 resets the current offset into the data buffer.
- The **buffer_start** register (0x10) sets the first address of the data buffer in the external memory. It must be aligned to 8 bytes.
- The **buffer_size** register (0x14) sets the size (in bytes) of the data buffer in the external memory. It must be aligned to 8 bytes.

The following registers are available for reading:

- The **status** register (0x00) provides information about the current status:
  - Bit 0 reports whether the receiver is enabled.
  - Bit 1 reports whether the reset of the current offset is in progress.

- The **descriptor_data** register (0x04) provides read access to the descriptor FIFO. Each descriptor describes one frame that has been transported to memory. It has the following structure:
  - Upper 16 bits are the offset into the data buffer where the first byte of the frame is located. It must be aligned to 8 bytes.
  - Lower 16 bits are the length of the frame.
- The **descriptor_count** register (0x08) reports the current count of descriptors in the descriptor FIFO. Only the lower 8 bits are used.

3.8.4 DMA transmitter

The DMA transmitter is implemented in the **ge_dma_tx2** entity. It transports Ethernet frames from external memory by becoming the master of the AXI interface provided by the **axi_m2s** and **axi_s2m** signals and transmits them to the GE interface provided by the **ge_port** output.

The AXI interface is synchronous to the **axi_clk** clock which is expected to run at 250 MHz (or any other frequency). The GE interface is synchronous to the **ge_clk** clock which is expected to run at 125 MHz (or 12.5 MHz for 100 Mb/s Ethernet). The DMA transmitter must therefore perform clock domain crossing between AXI and GE clock domains.

The location of the buffer in the memory is given by the **buffer_start** and **buffer_size** registers.

Structure

The DMA transmitter consists of the following components:

- The **descriptor FIFO** which transports descriptors from the management interface to the AXI process. Each descriptor describes one frame that was prepared to be transported from the external memory to the data FIFO. The descriptor consists of the length of the frame and the offset to be added to the base address of the data buffer in order to get the start address of the frame.
- The **AXI process** which reads descriptors from the **descriptor FIFO** and transports frames from the external memory to the **data FIFO** according to the descriptors. Once the frame is transported, it writes the length of the frame to the **metadata FIFO**. It is driven by the **axi_clk** clock.
- The **data FIFO** which holds the frames transported from the external memory by the AXI process to be transmitted by the GE process.
- The **metadata FIFO** which holds the length of the frames transported from the external memory by the AXI process to be used by the GE process.
- The **GE process** which reads frames from the **data FIFO** and transmits them to the GE interface. It waits until the length of the frame becomes available in the **metadata FIFO** which also means that the frame has been transported completely from the external memory. It aligns the count of the bytes read from the FIFO to the nearest multiple of eight to remove the effect of data width conversion done in the FIFO. It is driven by the **ge_clk** clock.

The structure is also shown in figure 3.18.

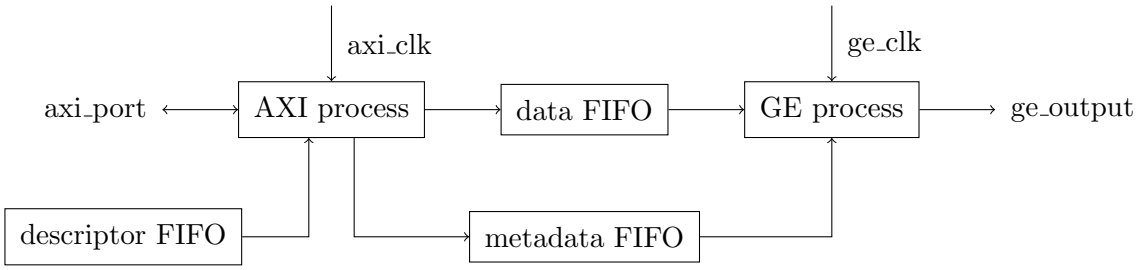

Figure 3.18: Structure of DMA transmitter

Management

The DMA transmitter is managed by the standard management interface.

The following registers are available for writing:

- The **control** register (0x00) allows configuration of the transmitter:
  - Bit 0 disables the transmitter when set to 0 and enables it when set to 1.
- The **descriptor_data** register (0x04) provides write access to the descriptor FIFO. Each descriptor describes one frame that has been prepared to be transported from memory. It has the following structure:
  - Upper 16 bits are the offset into the data buffer where the first byte of the frame is located. It must be aligned to 8 bytes.
  - Lower 16 bits are the length of the frame.
- The **buffer_start** register (0x10) sets the first address of the data buffer in the external memory. It must be aligned to 8 bytes.
- The **buffer_size** register (0x14) sets the size (in bytes) of the data buffer in the external memory. It must be aligned to 8 bytes.

The following registers are available for reading:

- The **status** register (0x00) provides information about the current status:
  - Bit 0 reports whether the transmitter is enabled.
- The **descriptor_count** register (0x08) reports the current count of descriptors in the descriptor FIFO. Only the lower 8 bits are used.

3.8.5 DMA feedback

The DMA feedback is implemented in the **ge_dma_fb** entity. It receives footers from the GE interface provided by the **ge_port** input and writes them into an internal FIFO where they can be read by software. The GE interface is synchronous to the **ge_clk** clock.

The typical usage of the DMA feedback is to make the TX timestamp which is added to the footer of each frame by the MAC transmitter available to software.

Structure

The DMA feedback consists of the following components:

- The **GE process** which receives footers from the GE interface and writes them to the **data FIFO**. It aligns the count of the bytes written to the FIFO to the nearest multiple of four to allow the data width conversion to be done in the FIFO. It is driven by the **ge_clk** length.
- The **data FIFO** which holds the footers received by the GE process. Read access to this FIFO is provided by the management interface.

Management

The DMA feedback is managed by the standard management interface.
The following registers are available for writing:

- The **control** register (0x00) allows configuration of the feedback:
  - Bit 0 disables the feedback when set to 1.
  - Bit 1 enables the feedback when set to 1.

The following registers are available for reading:

- The **status** register (0x00) provides information about the current status:
  - Bit 0 reports whether the feedback is enabled.
- The **data** register (0x04) provides read access to the data FIFO. The footer is split into 32-bit words so it is necessary to read multiple words in order to get the whole footer.
- The **count** register (0x08) reports the current count of 32-bit words in the data FIFO.

3.8.6 DMA block

The DMA block is implemented in the **ge_dma2** entity. It combines the following components:

- DMA receiver for receiving GE frames in the processing system.
- DMA transmitter for transmitting GE frames in the processing system.
- DMA feedback for acquiring timestamps of transmitted GE frames.

Integration

The structure and typical usage of the DMA block is shown in figure 3.19.

Verification

The DMA block was verified in simulation as described in section 4.4.2.

Validation

The DMA block was validated on real device later as part of the loopback test described in section 8.2.

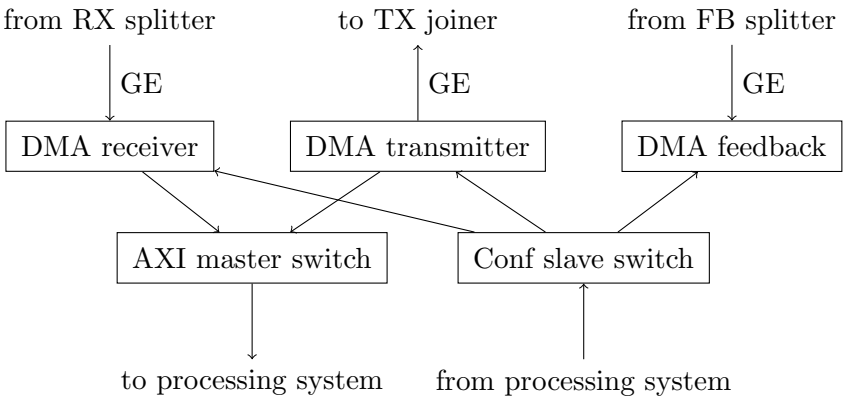

Figure 3.19: Usage of DMA block

3.9 Joining GE streams

After we designed the components for software processing of frames as shown in the last section, it was clear that there were two different GE streams that had to be combined together and transmitted on the same GE port. The first GE stream is passed from the other GE port of the pair. The second GE stream is injected from the processing system by the DMA block. These GE streams must be joined to create one GE stream to be transmitted by the MAC block.

3.9.1 Requirements

The joiner should be made generic in order to support any number of input ports. At the moment, only two input ports are required. In the future, the requirements might be different though. The joiner should use a fair algorithm to ensure that all input ports have equal priority when all buffers contain a frame.

3.9.2 Solutions

In this case, only one solution was considered because the requirements are very simple. Other solutions would be possible if priority of ports or even individual frames was given.

Round robin

The first idea was there would be a counter running from one to port count indicating which input buffer is allowed to take turn. If the current buffer had some frame available, it would be selected. After the transmission ended, the counter would be incremented. If the current buffer did not have any frame available, the counter would be incremented right away.

The only problem is that incrementing the counter only be one in each clock cycle is inefficient when frames are actually coming only from one of the input ports. In that case, there would be a delay of one clock cycle for each input port with no frames available. For that reason, the counter should be implemented in a way that it is allowed to increment by more than one in each clock cycle when the joiner is selecting the buffer for the next frame.

3.9.3 Buffer

The buffer is implemented in the **ge_buffer** entity. It allows buffering of Ethernet frames. When frame is received on the **ge_input** interface, the **available** output is asserted. When the **ready** input is asserted, the frame is transmitted on the **ge_output** interface. When there are no more frames in the buffer, the **available** output is deasserted.

Structure

The buffer consists of the following components:

- The **FIFO** component buffers the frames.
- The **WR** (write) process receives the frames from the GE interface and writes them to the FIFO. It is driven by the **ge_clk** clock.
- The **RD** (read) process reads the frames from the FIFO and transmits them to the GE interface. It is driven by the **ge_clk** clock.

The structure is also shown in figure 3.20.

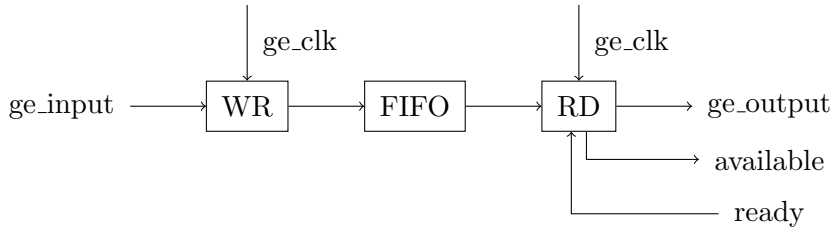

Figure 3.20: Structure of buffer

3.9.4 Joiner

The joiner is implemented in the **ge_joiner** entity. It allows joining multiple Ethernet streams together. It passes Ethernet frames from multiple Ethernet inputs provided by **ge_input**s to single Ethernet output provided by **ge_output**.

The joiner has the following generic parameters:

- The **input_count** parameter sets the number of inputs to the joiner. It also influences the number of buffers which are to be instantiated inside the joiner.

Structure

The joiner consists of the following components:

- The **Input Buffer** which is instantiated for every input port and holds the frames received on that input port until the output port becomes available.
- The **Control Process** which selects the current Input Buffer and controls the Output Multiplexer. It is driven by the **ge_clk** clock. It transitions between the following states:
  - The **idle** state when the joiner is waiting until one of the buffers reports that it contains a frame by asserting the available signal. The joiner transitions to the **start** state after selecting the buffer which first reported that it contains a frame.
  - The **start** state when the joiner is waiting until the selected buffer starts outputting the frame. The joiner transitions to the **frame** state after the selected buffer asserts the start of frame signal.
  - The **frame** state when the joiner is waiting until the current frame ends. The joiner transitions to the **footer** state after the end of frame signal is asserted.
  - The **footer** state when the joiner is waiting until the current footer ends. The joiner transitions to the **idle** state after the end of footer signal is asserted.

- The **Output Multiplexer** which connects the output of the current Input Buffer to the output port. It is controlled by the Control Process.

The structure is also shown in figure 3.21.

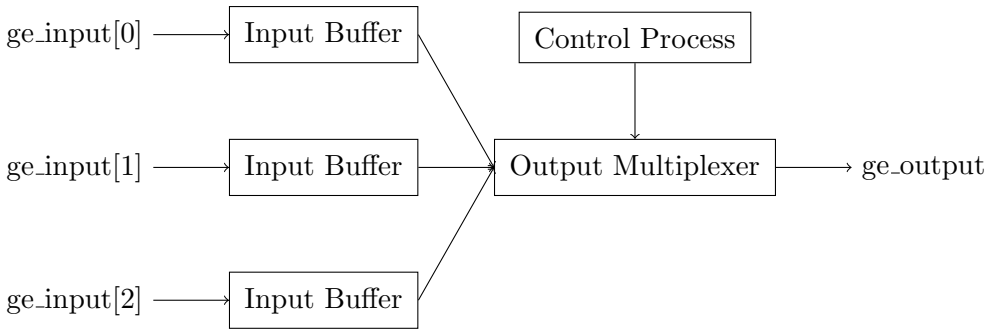

Figure 3.21: Structure of joiner

Management

The joiner is not currently managed in any way.

Verification

The joiner was verified in simulation as described in section 4.4.5.

Validation

The joiner was validated on real device as part of the passthrough test described in section 8.1.

3.10 Receiving and transmitting XGE

This section describes the first components created for ten gigabit Ethernet (XGE). As explained in section 3.3, the XGMII interface was not suitable for transporting Ethernet frames between entities. For that purpose, the XGE interface was defined and this section shows the design of the entities used for conversion between the XGMII interface and the XGE interface.

3.10.1 Requirements

The goal of this section is to build everything between the convenient internal XGE interface as described in section 3.3.6 and the Ethernet physical layer transceiver provided by the SFP+ module.

3.10.2 Solutions

The solution is divided into two parts. First, the internal XGE interface will be converted to the standard XGMII interface. Then, the XGMII interface will be connected to the GT (gigabit transceiver) of the FPGA by instantiating Xilinx IP for ten gigabit Ethernet.

Converting from XGE to XGMII

To convert the frames from the XGE interface to the XGMII interface, a receiver and a transmitter similar to the ones used for gigabit Ethernet will be designed.

The receiver will receive the XGMII stream on one side and analyze the control characters in order to detect the start of frame and the end of frame. On the other side, it will transmit the frame to the XGE interface with the preamble removed and control characters converted to valid signals.

The transmitter will receive the XGE stream on one side and watch the valid signals in order to detect the start of frame and the end of frame. On the other side, it will transmit the frame to the XGMII interface with the preamble added and valid signals converted to control characters. It will also transmit idle control characters when the XGE interface is idle.

Interfacing to SFP+ module

For interfacing to the high speed interface of the SFP+ module as described in section 2.1.9, the **10G/25G High Speed Ethernet Subsystem** IP provided by Xilinx will be used. It includes many components out of which some are provided for free and some are paid. In our case, the only component needed is the 10GBASE-R PCS/PMA which configures the transceivers and also performs encoding and decoding as required by this physical layer. Fortunately, that component is provided by Xilinx for free.

The IP itself has many ports, some of which will not be used in our design, so I will create a wrapper for the IP. The wrapper will have only the GT (gigabit transceiver) interface on one side and the XGMII interface on the other side.

3.10.3 XGE MAC receiver

The XGE MAC receiver is implemented in the **xge_mac_rx** entity. It receives Ethernet frames from the XGMII interface provided by the **xgmii_input** and transmits them to the XGE interface provided by the **xge_output**. Both interfaces are synchronous to the **clk** clock.

Structure

The XGE MAC receiver consists of one process which handles the conversion. The frame may start either in lane 0 or in lane 4 on the 64-bit XGMII interface while it may start only in lane 0 on the XGE interface. The process must therefore align the frame to lane 0 on the XGE interface when it starts in lane 4 on the XGMII interface. The data signals of both interfaces are connected together. The valid signals of the XGE interface are generated simply by inverting the control signals of the XGMII interface.

3.10.4 XGE MAC transmitter

The XGE MAC transmitter is implemented in the **xge_mac_tx** entity. It receives Ethernet frames from the XGE interface provided by the **xge_input** and transmits them to the XGMII interface provided by the **xge_output**. Both interfaces are synchronous to the **clk** clock.

Structure

The XGE MAC transmitter consists of one process which handles the conversion. When the XGE interface is idle, the IDLE control characters are transmitted on the XGMII interface. When the first valid bit of the XGE interface is asserted, the START control character and the preamble are transmitted on the XGMII interface. The data is then passed between the interfaces without any change. When any valid bit of the XGE interface is deasserted, the TERMINATE control character is transmitted on the XGMII interface.

3.10.5 XGE MAC block

The XGE MAC block is implemented in the **xge_mac** entity. It combines the following components:

- XGE MAC receiver for receiving XGE frames from XGMII interface.
- XGE MAC transmitter for transmitting XGE frames to XGMII interface.

Verification

The XGE MAC block was verified in simulation as described in section 4.4.6 and 4.4.7.

Validation

The XGE MAC block was validated on real device as part of the passthrough test described in section 8.1.

3.10.6 XGE PCS block

The XGE PCS block is implemented in the **xge_pcs** entity. It is a wrapper of the **10G/25G High Speed Ethernet Subsystem** IP provided by Xilinx. It includes both receiver and transmitter which use the gigabit transceivers of the FPGA to convert the XGMII interface provided by **xgmii_rx** and **xgmii_tx** to the 10GBASE-R physical layer supported by the SFP+ modules. The **xgmii_rx** interface is synchronous to the **xgmii_rx_clk**. The **xgmii_tx** interface is synchronous to the **xgmii_tx_clk**. Both clocks are generated inside the IP so they are provided as outputs of this entity.

3.11 Converting GE to XGE

This section explains what was done to pass Ethernet frames from GE (gigabit Ethernet) domain to XGE (ten gigabit Ethernet) domain. It was the first step towards the final goal of creating CMP packages from Ethernet frames and outputting them to the XGE port. So the components described in this section will be used together with the components described in the next section in order to implement the CMP protocol. At the same time, I decided to use the same components for direct communication between the GE ports and the XGE port. The idea was that sometimes, it would be useful to use the XGE port for direct connection to one of the GE ports for communication between a computer and an automotive unit.

3.11.1 Requirements

We need to transition from the GE interface described in section 3.3.5 to the XGE interface described in section 3.3.6. The GE interface transports one byte or 8 bits on each clock cycle and uses 125 MHz clock. The XGE interface transports eight bytes or 64 bits on each clock cycle and uses 156.25 MHz clock.

The first goal is to make an entity which performs clock domain crossing (because the frequency of the clock differs between GE and XGE) and also data width crossing (because XGE uses eight times wider data path than GE). The interfaces also differ in start of frame and end of frame signals. While newer GE interface uses explicit signals, older XGE interface uses valid bits to indicate those events.

The second goal is to make an entity which joins multiple XGE streams into single XGE stream. It should have the same functions and interfaces as the GE variant described in section 3.9.4. The implementation will be slightly different though due to the differences between the interfaces mentioned above.

3.11.2 GE–XGE buffer

The buffer is implemented in the **ge2xge_buffer2** entity. It converts frames from the GE interface to the XGE interface. It contains an independent clock FIFO to perform clock domain crossing and data width conversion at the same time.

When frame is received on the **ge_input** interface, the **available** output is asserted. When the **ready** input is asserted, the frame is transmitted on the **xge_output** interface. When there are no more frames in the buffer, the **available** signal is deasserted.

Structure

The buffer consists of the following components:

- The **FIFO** component buffers the frames. It contains a FIFO primitive as described in section 3.2.1. The write port uses the **ge_clk** clock and is 8 bits wide. The read port uses the **xge_clk** clock and is 64 bits wide.
- The **WR** (write) process receives the frames from the GE interface and writes them to the FIFO. It also aligns the number of bytes written to the FIFO to the nearest higher multiple of eight so that the last word can be read from the FIFO when it is not complete. It is driven by the **ge_clk** clock.
- The **RD** (read) process reads the frames from the FIFO and transmits them to the XGE interface. It is driven by the **xge_clk** clock.
- The **SEM** (semaphore) component synchronizes the frame count between the write process and the read process. The write process gives the semaphore when it finishes writing a frame to the FIFO. The read process takes the semaphore when it starts reading a frame from the FIFO.

The structure is also shown in figure 3.22.

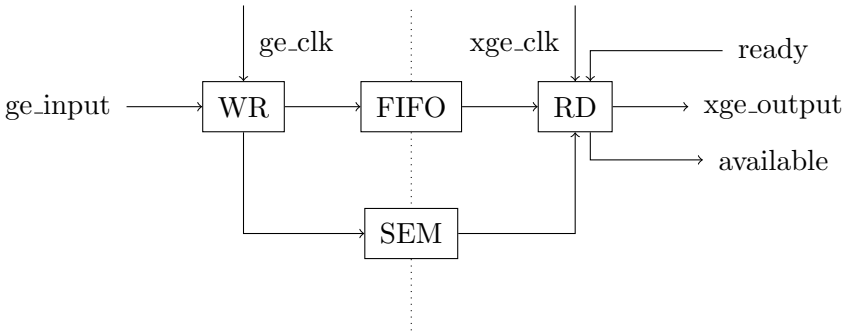

Figure 3.22: Structure of buffer

3.11.3 GE–XGE joiner

The joiner is implemented in the **ge2xge_joiner** entity. It joins multiple gigabit Ethernet streams into single ten gigabit Ethernet stream. It passes Ethernet frames from the **ge_input**s to the **xge_output** and performs conversion from GE to XGE interface on the way.

The joiner has the following generic parameters:

- The **input_count** parameter sets the number of inputs of the joiner. It also influences the number of buffers which are instantiated inside the joiner.

Structure

The joiner consists of the following components:

- The **GE–XGE buffer** which is instantiated for every input port. It performs the conversion from GE to XGE. It also holds the frame until the output port becomes available.
- The **Control** process which selects the current input port according to the available signals and then waits until the buffer outputs the frame with the footer.
- The **Multiplexer** which connects the output of the buffer selected by the **Control** process to the output port of the joiner.

The structure is also shown in figure 3.23.

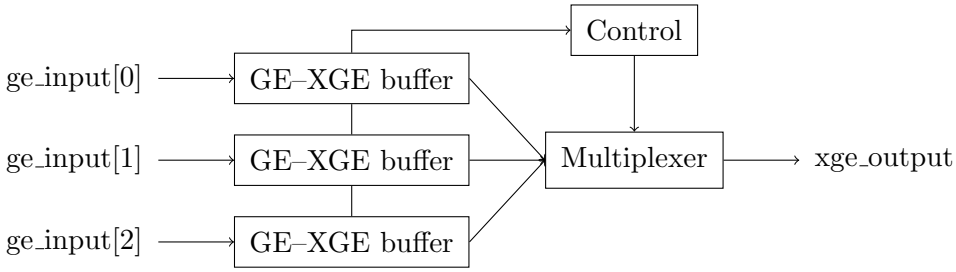

Figure 3.23: Structure of joiner

3.12 Converting XGE to GE

This section explains what was done to pass Ethernet frames from XGE (ten gigabit Ethernet) domain to GE (gigabit Ethernet) domain. It was not explicitly required by any of the goals given by the topic but I thought it would be useful to allow using the XGE port not only for logging, but also for communication between the computer connected to that port and the automotive units connected to the GE ports. The first part of that which is the transition from the GE port to the XGE port was already made possible in the previous section. The second part of that which is the transition from the XGE port to the GE port is described in this section.

3.12.1 Requirements

We need to transition from the XGE interface described in section 3.3.6 to the GE interface described in section 3.3.5. The XGE interface transports eight bytes or 64 bits on each clock cycle and uses 156.25 MHz clock. The GE interface transports one byte or 8 bits on each clock cycle and uses 125 MHz clock.

The first goal is to make an entity which performs clock domain crossing (because the frequency of the clock differs between XGE and GE) and also data width crossing (because GE uses eight times narrower data path than GE). The interfaces also differ in start of frame and end of frame signals. While older XGE interface uses valid bits, newer GE interface uses explicit signals to indicate those events.

The second goal is to make an entity which splits single XGE stream into multiple XGE streams. It should have the same functions and interfaces as the GE variant described in section 3.7.4. The implementation will be slightly different though due to the differences between the interfaces mentioned above.

3.12.2 XGE–GE buffer

The buffer is implemented in the **xge2ge_buffer2** entity. It converts frames from the XGE interface to the GE interface. It contains an independent clock FIFO to perform clock domain crossing and data width conversion at the same time.

When a frame is received on the **xge_input** interface, the **available** output is asserted. When the **ready** input is asserted, the frame is transmitted on the **ge_output** interface. When there are no more frames in the buffer, the **available** signal is deasserted.

Structure

The buffer consists of the following components:

- The **FIFO** component buffers the frames. It contains a FIFO primitive as described in section 3.2.1. The write port uses the **xge_clk** clock and is 64 bits wide. The read port uses the **ge_clk** clock and is 8 bits wide.
- The **WR** (write) process receives the frames from the XGE interface and writes them to the FIFO. It is driven by the **xge_clk** clock.
- The **RD** (read) process reads the frames from the FIFO and transmits them to the GE interface. It also aligns the number of bytes read from the FIFO to the nearest higher multiple of eight so that the next frame can be read correctly from the FIFO. It also adds an empty footer to the frame since frames without footer are not allowed in the GE interface. It is driven by the **ge_clk** clock.

The structure is also shown in figure 3.24.

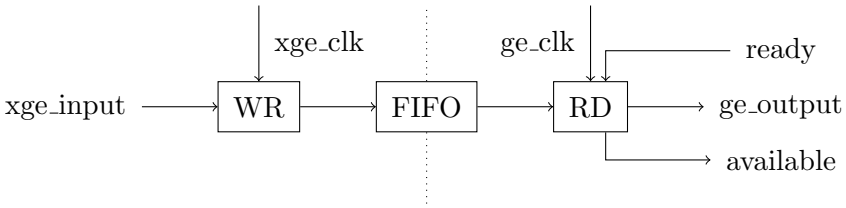

Figure 3.24: Structure of buffer

3.12.3 XGE–GE splitter

The splitter is implemented in the **xge2ge_splitter** entity. It splits single ten gigabit Ethernet stream into multiple gigabit Ethernet streams. It passes Ethernet frames from the **xge_input** to the **ge_output**s and performs conversion from XGE to GE interface on the way.

The splitter has the following generic parameters:

- The **output_count** parameter sets the number of outputs of the splitter. It also influences the number of buffers which are instantiated inside the splitter.

At the moment, the splitter transmits all the frames received from the input port to all the output ports. This should however change in the future.

Structure

The splitter consists of the following components:

- The **XGE–GE buffer** which is instantiated for every output port. It performs the conversion from XGE to GE.

3.13 Packaging GE to CMP

Once it became possible to transport frames from the GE domain to the XGE domain, it was desirable to add some metadata to each frame.

The most significant problem was that it was impossible to know on which interface the frame was captured. So the metadata should include some kind of identification of the interface and also information whether the frame was received or transmitted on that interface.

Another common feature of devices similar to ours is that each frame is timestamped and the timestamp is included in the metadata. That is especially useful when multiple devices are used to capture on many interfaces at the same time. For that case, however, precise time synchronization would be also required.

For this task, the CMP protocol described in section 2.4 was chosen. It has been specifically designed for devices capturing data from various automotive networks. It also provides fields for the metadata mentioned in previous paragraphs. Finally, it is a standard solution so it could be supported by existing software. For example, a packet dissector for this protocol already exists for Wireshark, so I will be able to check my frames in a very simple manner.

3.13.1 Requirements

The goal is to make an entity which packages GE frames into CMP frames. That means adding a few headers according to the specification of the CMP protocol. Apart from static fields, the headers include the length of the frame (which can be measured by buffering the whole frame) and the timestamp of the frame (which can be obtained from the footer following the frame).

In order to reduce the overhead introduced by the CMP protocol and also reduce the worload of the network through which CMP frames will have to pass after being transmitted from our device, the CMP packager should also support aggregation as described in the specification. That means there will be single CMP header added for multiple CMP data messages (where one message corresponds to one Ethernet frame). The aggregation should consider the maximal length of an Ethernet frame (which could be set either to 1500 bytes according to the standard or more when jumbo frames are considered) and the maximal delay introduced by the aggregation (which will matter when the utilization of the network is low).

3.13.2 Solutions

For creating CMP packages, three different solutions of increasing complexity were considered during the development.

Adding headers to each frame

The simplest solution would be adding all required CMP headers to each frame. In order to form a valid frame according to the specification, the following headers would have to be added:

- The Ethernet header with MAC addresses and CMP EtherType = 14 bytes.
- The CMP header with device ID and stream counter = 8 bytes.
- The CMP data message header with timestamp and interface ID = 16 bytes.
- The CMP Ethernet payload header with frame length = 6 bytes.

That makes a total of 44 bytes added to each frame. With long frames, there would be no problem. With short frames, the overhead would be extremely high though. The shortest frames on Ethernet have 64 bytes. So the resulting CMP package would have 108 bytes, out of which the original frame would make about 60% only.

This solution would be the simplest because it would require only one entity running in the GE domain for adding the headers. On the other hand, it would limit the maximal throughput from each GE port to XGE port, which could become a problem especially in case of a sequence of many short frames going after each other.

Aggregation in GE domain

To address the problem and reduce the overhead, the CMP protocol supports aggregation. That means putting multiple payloads, each with one Ethernet frame, into single CMP package. In that case, the Ethernet header and the CMP header are added only once for each CMP package.

The total size of the CMP package is limited only by the maximal size of an Ethernet frame. That was originally set to 1500 bytes but may be increased up to 9000 bytes with jumbo frames.

The following headers would still be added to each frame:

- The CMP data message header with timestamp and interface ID = 16 bytes.
- The CMP Ethernet payload header with frame length = 6 bytes.

That makes a total of 22 bytes added to each frame. Thanks to aggregation, the overhead would be reduced almost to one half. The common headers would still have to be added from time to time. With long frames, the situation would be the same as with no aggregation. With short frames, however, the overhead would drop significantly.

Let us consider the case of a sequence of many short frames. The CMP package could contain 16 payloads. Each of them would consist of 22 bytes of header and 64 bytes of data. Also, there would be 22 bytes of the common headers. The total length of the package would be 1398 bytes, out of which 1024 bytes would be the original frames, making about 75% of the package.

The question was when and where to make the aggregation happen. The most logical choice would be to aggregate the frames in the XGE domain. That would allow adding other types of payloads (like CAN and LIN messages) to the CMP packages later. However, it would be difficult to implement due to reasons described in the next solution. So it was first considered whether aggregation in the GE domain would be sufficient.

It would mean that aggregation would occur only for frames received or transmitted on the same interface. It would still solve the problem of a sequence of many short frames by using single CMP header for multiple Ethernet payloads. It all sounded like this solution would be a good compromise between functionality and complexity.

As always, there was a catch. It lies in the fields of the first CMP header, the stream ID and the stream counter. The specification states that for each stream ID, the stream counter should monotonically increase between subsequent CMP packages. Our device should produce only one stream with one stream ID so it should have only one stream counter. On the other hand, with this solution, there would be an entity performing aggregation for each GE port. It would be necessary to implement a lock for sharing the counter among all GE ports but that significantly increases the complexity of this solution. Another workaround would be to use a different stream ID for each GE port but I believe it does not conform to the specification of the protocol. For those reasons, the third solution was considered and chosen as the final solution.

Aggregation in XGE domain

The final solution was to split the packaging into two phases:

1. The frame headers (the CMP data message header and the CMP Ethernet payload header) would be added in the GE domain.
2. The package headers (the Ethernet header and the CMP header) would be added in the XGE domain where payloads would be also combined together.

This solution would be definitely the most flexible one and the most complex one at the same time. As mentioned earlier, the CMP payload could come not only from Ethernet but also from other types of interfaces supported by the CMP protocol like CAN and LIN. The problem of stream counter would be solved as there would be only one entity adding the CMP header.

In order to support the full throughput of eight GE ports which are planned for the final device, this would be almost sufficient. The package headers of 22 bytes for each package would be added in the XGE domain so that would not limit the throughput in any way. The frame headers of 22 bytes for each frame would still be added in the GE domain. When frames are transmitted on the GE link, there are 8 bytes of preamble before each frame and 12 bytes of inter-frame gap after each frame. That makes a total of 20 bytes which are replaced by 22 bytes in the CMP payload. So there is still a slack of 2 bytes which limits the maximal throughput. However, it could be solved by simply increasing the clock and creating a faster clock domain for transporting CMP payloads between the GE domain and the XGE domain.

The greatest challenge of this solution will be combining the CMP payloads together and also computing the CRC of the CMP package. The problem is that in the XGE domain, the data path is 8 bytes wide so that the clock can have an acceptable frequency. That would be fine if the length of each payload was a multiple of eight bytes. Unfortunately, it is not the case of the CMP protocol. Even the package headers have 22 bytes together which means the first payload starts in the middle of the eight byte word. Each payload may have any length between 86 and 1544 bytes because it consists of the frame headers which have 22 bytes together and the original Ethernet frame which may have any length between 64 and 1522 bytes. Combining the CMP package headers with an arbitrary count of CMP payloads and the CRC checksum of the package will require aligning them properly so that there is no gap between them.

3.13.3 CMP packager

The CMP packager is implemented in the **ge_cmp_packager** entity. It receives Ethernet frames from the **ge_input** interface, packages them into CMP payload and transmits them to the **ge_output** interface.

It sets the following fields of the CMP data message header:

- The Timestamp is set to the timestamp from the GE footer.
- The Interface ID is set according to the interface ID register described below.
- The Payload Type is set to 0x08 which corresponds to Ethernet.
- The Payload Length is set to the length of the GE frame plus 6 bytes (the length of the CMP Ethernet payload header).

It sets the following fields of the CMP Ethernet payload header:

- The Data Length is set to the length of the GE frame.

All remaining fields are set to zero.

Structure

The CMP packager consists of the following components:

- The **RX process** which receives GE frames from the input interface and writes them to the frame FIFO. It also measures the length of the frame and extracts the timestamp of the frame from the footer. It passes this information to the CMP process. It is driven by the **ge_clk** clock.

- The **CMP process** which creates CMP headers (CMP data message header and CMP Ethernet payload header to be correct) and writes them to the header FIFO. It gets the metadata of the frame from the RX process. It is driven by the **ge_clk** clock.
- The **frame FIFO** which stores the frames received by the RX process to be transmitted by the TX process. It is 8 bits wide and uses the **fifo8** entity described in section 3.2.1.
- The **header FIFO** which stores the headers created by the CMP process to be transmitted by the TX process. It is 8 bits wide and uses the **fifo8** entity described in section 3.2.1.
- The **TX process** which reads GE frames from the frame FIFO and CMP headers from the header FIFO and transmits CMP payloads to the output interface. Each CMP payload consists of the CMP header and the GE frame. It also adds a gap between the payloads so that the next entity handles them correctly. It is driven by the **ge_clk** clock.

The structure is also shown in figure 3.25.

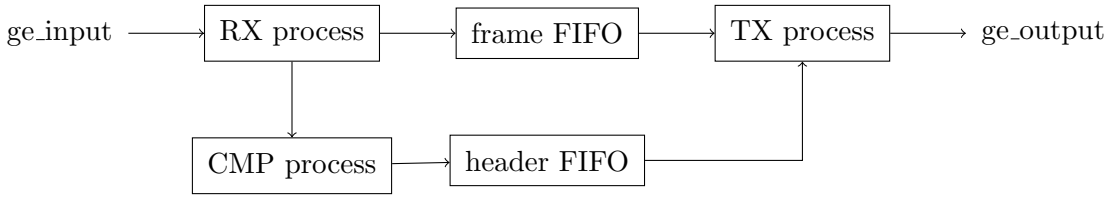

Figure 3.25: Structure of CMP packager

Management

The CMP packager is managed by the standard management interface.

The following registers are available for writing:

- The **control** register (0x00) allows enabling and disabling the packager:
  - Bit 0 disables the packager when set to 1.
  - Bit 1 enables the packager when set to 1.
- The **interface ID** register (0x04) sets the 32-bit interface ID used in the CMP data message header.

The following registers are available for reading:

- The **status** register (0x00) provides information about the current status:
  - Bit 0 reports whether the packager is enabled.

3.13.4 CMP aggregator

The CMP aggregator is implemented in the **xge_cmp_aggregator** entity. It receives CMP payloads from the **xge_input** interface and combines them into CMP packages. It generates the Ethernet header and the CMP header at the start of every package. It also generates the CRC checksum at the end of every package. Finally, it passes the package through the XGE combinator and transmits it to the **xge_output** interface.

It sets the following fields of the Ethernet header:

- The destination and source addresses are set according to the memory described below.

- The EtherType is set to 0x99FE which corresponds to CMP.

It sets the following fields of the CMP header:

- The Device ID is set according to the device ID register described below.
- The Message Type is set to 0x01 which corresponds to data message.
- The Stream ID is set according to the stream ID register described below.
- The Stream Counter is set to the current value of the internal counter which starts at zero and increments after every CMP package.

All remaining fields are set to zero.

Structure

The CMP aggregator consists of the following components:

- The **input process** which receives CMP payloads from the input XGE interface and writes them to the payload FIFO. It also measures the length of the payload and passes it to the output process via the length FIFO. It is driven by the **xge_clk** clock.
- The **payload FIFO** which stores the payload received by the input process before it is processed by the output process. It is 64 bits wide and uses a Block RAM as described in section 3.2.1.
- The **length FIFO** which stores the length of the payload received by the input process before it is processed by the output process. It is 12 bits wide to support payloads of length up to 4095 bytes and uses a LUT RAM as described in section 3.2.1.
- The **output process** which reads CMP payloads from the payload FIFO and passes them to the XGE combinator. It decides when CMP packages are started and ended according to the maximal length and delay set by the management registers. It also generates the appropriate headers (Ethernet header and CMP header) and footers (CRC checksum) before and after every CMP package.
- The **XGE combinator** which combines the CMP payloads with the header and footer created by the output process and transmits them to the output XGE interface.

The structure is also shown in figure 3.26.

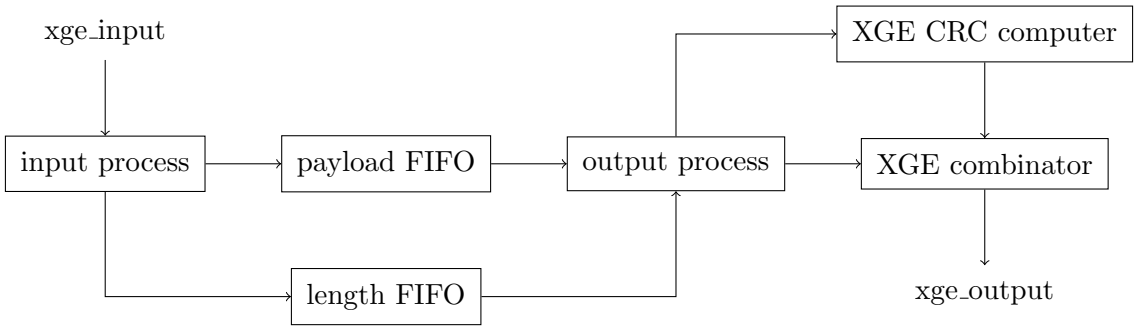

Figure 3.26: Structure of CMP aggregator

Management

The CMP aggregator is managed by the standard management interface.

The following registers are available for writing:

- The **control** register (0x00) allows enabling and disabling the aggregator:
  - Bit 0 disables the aggregator when set to 1.
  - Bit 1 enables the aggregator when set to 1.
- The **max length** register (0x04) sets the maximal length (expressed in bytes) of the CMP package. The aggregator terminates the current CMP package when this length would be exceeded by adding the next CMP payload to the package. It is set to 1500 bytes by default.
- The **max delay** register (0x08) sets the maximal delay (expressed in clock cycles) of the CMP package. The aggregator terminates the current CMP package when this delay is exceeded and there is no CMP payload available in the FIFO. It is set to 1500 clock cycles by default which corresponds approximately to 10 microseconds.
- The **device ID** register (0x20) sets the 16-bit device ID used in the CMP header. It is set to zero by default.
- The **stream ID** register (0x24) sets the 8-bit stream ID used in the CMP header. It is set to zero by default.

The following memories are available for writing by bytes or words:

- The **destination MAC address** memory (0x10-0x15) sets the destination MAC address used in the Ethernet header. It is set to FF:FF:FF:FF:FF:FF by default.
- The **source MAC address** memory (0x18-0x1D) sets the source MAC address used in the Ethernet header. It is set to 00:00:00:00:00:00 by default.

The following registers are available for reading:

- The **status** register (0x00) provides information about the current status:
  - Bit 0 reports whether the aggregator is enabled.

3.13.5 XGE combinator

The XGE combinator is implemented in the **xge_combinator** entity. It combines shorter frames received from the **xge_input** interface into longer frames and transmits them to the **xge_output** interface.

It ensures that there are no gaps in the longer frames between the shorter frames by aligning them to proper lanes. The first frame is always aligned to start in the first lane. Every following frame is then aligned to start in the lane next to the lane where the previous frame ended. For example, when the first frame ends in lane 1, the second frame is aligned to start in lane 2, when the second frame ends in lane 7, the third frame is aligned to start in lane 0, and so on.

The combination process is terminated when the **terminate** signal is asserted. Only then, the longer frame is transmitted to the output interface. The next shorter frame received after the termination becomes part of the next longer frame.

Structure

The XGE combinator consists of the following components:

- The **control process** which combines the frames received from the input XGE interface and writes them into the FIFO. It is driven by the **xge_clk** clock.
- The **FIFO** which stores the combined frame created by the control process before it is terminated by the terminate signal and transmitted by the transmit process. It uses the **fifo64** entity with memory primitive as described in section 3.2.1.
- The **semaphore** which counts the terminated frames stored in the FIFO. It is given by the terminate signal. It is taken by the transmit process when a frame transmission starts.
- The **transmit process** which reads the frames from the FIFO and transmits them to the output XGE interface once the combination is finished. It is driven by the **xge_clk** clock.

The structure is also shown in figure 3.27.

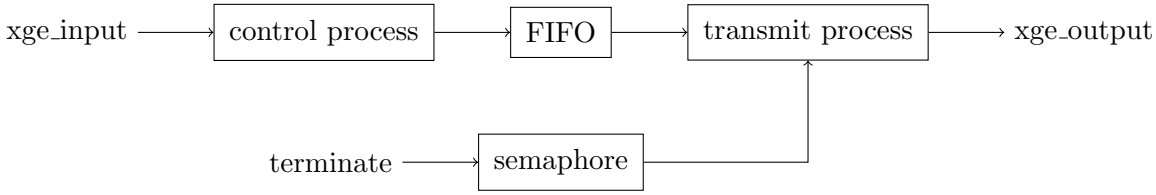

Figure 3.27: Structure of XGE combinator

3.13.6 XGE CRC computer

The XGE CRC computer is implemented in the **xge_crc** entity. It receives Ethernet frames from the **xge_input** interface and computes CRC checksum for each of them as required by Ethernet and described in section 2.1.2. The checksum appears on the **crc_output** signal and is updated with each word received. Before each frame, the **crc_rst** signal must be asserted in order to reset the CRC computation.

3.13.7 CMP block

The CMP block is implemented in the **ge_cmp** entity. It combines the following components:

- CMP packager for packaging received GE frames into CMP payloads.
- CMP packager for packaging transmitted GE frames into CMP payloads.

Integration

The structure and typical usage of the CMP block is shown in figure 3.28.

Verification

The CMP block was verified in simulation as described in section 4.5.3.

Validation

The CMP block was validated on real device later as part of the tests described in section 8.

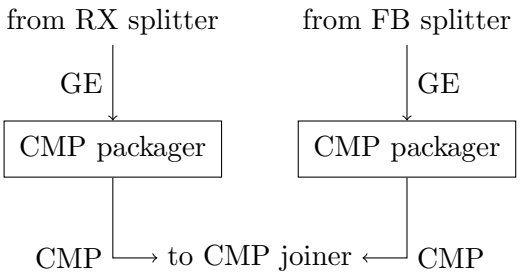

Figure 3.28: Structure of CMP block

3.14 Managing the transceivers

Apart from the data interface for transporting Ethernet frames, the PHY has the management interface for monitoring and configuration. It is not necessary to use this interface in order to get Ethernet working. However, it brings some advantages like access to the loopback modes of the PHY or ability to change the behavior of the LEDs connected to the PHY.

3.14.1 Requirements

In our device, there will be two different kinds of Ethernet transceivers.

First, there will be gigabit (or slower) Ethernet PHYs for connecting the automotive units. The management interface is described in section 2.1.8 and specified in clauses 22 and 45 of the IEEE 802.3 standard [6]. It is similar to the I2C bus but the frame format is different.

Second, there will be ten gigabit Ethernet PHYs integrated in SFP+ modules for logging the captured data. The management interface is described in section 2.1.9 and specified in chapter 5 of the SFF-8419 specification [14]. It is compatible with the I2C bus.

In both cases, the management interface should be accessible from software so we need to create a converter from the internal management interface described in section 3.4 to the management interface described above.

3.14.2 Solutions

Using the I2C controller

The peripherals available on the chip we are using include two I2C controllers. They could be used with the management interface of the SFP+ modules which is compatible with the I2C bus. The clock and data signals of each I2C controller can be routed from the PS (processing system) to the PL (programmable logic) via the EMIO interface. Then, they can be mapped to any pin of the FPGA as needed by the schematic of the board.

The advantage of this solution is that it would be very simple to implement because the controller has already been designed and verified by the manufacturer. Moreover, the driver for this controller has already been written for both bare metal and Linux environment.

The disadvantage of this solution is that it does not support the standard management interface of the Ethernet PHYs. Although similar to the I2C bus, the differences are substantial enough to prevent generic I2C controllers from being used.

Designing a custom controller

Both interfaces are simple enough that developing a custom controller for them should not take much time. The advantage is that it will be seamlessly integrated into the rest of my design. It will provide the same internal management interface as the rest of my components.

3.14.3 SMI controller

The SMI controller is implemented in the **conf_smi** entity. It provides access to the standard management interface (SMI) of Ethernet PHYs as specified in clause 22 of the IEEE 802.3 standard [6]. The internal management interface is provided by the **conf_m2s** and **conf_s2m** signals. The standard management interface (SMI) is provided by the **smi_mdc** output and the **smi_mdio** bidirectional signal. All interfaces are synchronous to the **clk** clock.

Structure

The SMI controller consists of the following components:

- The **MDC** process drives the MDC signal. It performs the division of the clock according to the set MDC divisor so that the maximum MDC frequency of the interface is respected. It also generates *mdc_rising* and *mdc_falling* signals which are used by the MDIO process for correct timing.
- The **MDIO** process drives the MDIO signal. It reads commands from the CMD FIFO. For each command, it first generates the 16 bit header according to the specification. For write commands, it then reads the associated 16 bits of data from the TX FIFO and transmits them via the MDIO signal. For read commands, it then receives 16 bits of data via the MDIO signal and writes them to the RX FIFO.
- The **CMD** (command) FIFO stores the commands to be executed by the controller. Each command includes the device address, the register address and the operation (read/write). Write access to this FIFO is provided by the management interface.
- The **TX** (transmit) FIFO stores the data to be written to the peripheral by the controller. Write access to this FIFO is provided by the management interface.
- The **RX** (receive) FIFO stores the data to be read from the peripheral by the controller. Read access to this FIFO is provided by the management interface.

Management

The SMI controller is managed by the standard management interface.

The following registers are available for writing:

- The **control** register (0x00) allows enabling and disabling the controller:
  - Bit 0 disables the controller when set to 1.
  - Bit 1 enables the controller when set to 1.
- The **mdc_divisor** register (0x08) sets the divisor used to generate the **smi_mdc** clock from the **clk** clock.
- The **cmd_data** register (0x10) provides write access to the CMD FIFO. Each command has the following structure:
  - Bits 4 to 0 set the register address of the transaction.
  - Bits 9 to 5 set the device address of the transaction.
  - Bit 10 sets the type of the transaction (0 = write / 1 = read).
- The **tx_data** register (0x14) provides write access to the TX FIFO. Only the lower 16 bits are used.

The following registers are available for reading:

- The **status** register (0x00) provides information about the current status:
  - Bit 0 reports whether the controller is enabled.
- The **rx_data** register (0x18) provides read access to the RX FIFO. Only the lower 16 bits are used.
- The **cmd_count** register (0x20) reports the current count of the commands stored in the CMD FIFO.
- The **tx_count** register (0x24) reports the current count of the 16 bit words stored in the TX FIFO.
- The **rx_count** register (0x28) reports the current count of the 16 bit words stored in the RX FIFO.

3.14.4 I2C controller

The I2C controller is implemented in the **conf_i2c** entity. It provides access to the two wire interface (TWI) of SFP+ modules as specified in chapter 5 of the SFF-8419 specification [14]. The internal management interface is provided by the **conf_m2s** and **conf_s2m** signals. The two wire interface (TWI) is provided by the **i2c_scl** output and the **i2c_sda** bidirectional signal. All interfaces are synchronous to the **clk** clock.

Structure

The I2C controller consists of the following components:

- The **SCL** process drives the SCL signal. It performs the division of the clock according to the set SCL divisor so that the maximum SCL frequency of the interface is respected. It also generates *scl_mid_low* and *scl_mid_high* signals which are used by the SDA process for correct timing.
- The **SDA** process drives the SDA signal. It reads commands from the CMD FIFO. For each command, it first generates the START condition followed by the 7 bit address. For write commands, it then reads N bytes of data from the TX FIFO and transmits them via the SDA signal. For read commands, it then receives N bytes of data via the SDA signal and writes them to the RX FIFO. Finally, it generates the STOP condition. When positive ACK (acknowledge) is not received after transmitting address or data, the STOP condition is generated immediately.
- The **CMD** (command) FIFO stores the commands to be executed by the controller. Each command includes the address, the length and the type (read/write) of the transaction. Write access to this FIFO is provided by the management interface.
- The **TX** (transmit) FIFO stores the data to be written to the peripheral by the controller. Write access to this FIFO is provided by the management interface.
- The **RX** (receive) FIFO stores the data to be read from the peripheral by the controller. Read access to this FIFO is provided by the management interface.

Management

The I2C controller is managed by the standard management interface.

The following registers are available for writing:

- The **control** register (0x00) allows enabling and disabling the controller:
  - Bit 0 disables the controller when set to 1.
  - Bit 1 enables the controller when set to 1.
- The **scl_divisor** register (0x08) sets the divisor used to generate the **i2c_scl** clock from the **clk** clock.
- The **cmd_data** register (0x10) provides write access to the CMD FIFO. Each command has the following structure:
  - Bits 6 to 0 set the address of the transaction.
  - Bit 7 sets the type of the transaction (0 = write / 1 = read).
  - Bits 15 to 8 set the length of the transaction (in bytes).
- The **tx_data** register (0x14) provides write access to the TX FIFO. Only the lower 8 bits are used.

The following registers are available for reading:

- The **status** register (0x00) provides information about the current status:
  - Bit 0 reports whether the controller is enabled.
- The **rx_data** register (0x18) provides read access to the RX FIFO. Only the lower 8 bits are used.
- The **cmd_count** register (0x20) reports the current count of the commands stored in the CMD FIFO.
- The **tx_count** register (0x24) reports the current count of the 8 bit words stored in the TX FIFO.
- The **rx_count** register (0x28) reports the current count of the 8 bit words stored in the RX FIFO.

3.15 Putting it all together

After designing all entities, the last step was connecting them together into greater and greater entities until there was only one entity called the top level entity that would be later used for synthesis and implementation on the real device. There will be two kinds of connections between the entities actually:

- The **Ethernet** interfaces described in section 3.3 will form the data path between gigabit and ten gigabit Ethernet ports.
- The **Management** interfaces described in section 3.4 will form the control path between the processing system and all entities.

3.15.1 Requirements

The goal is to create an entity called Ethernet system which could be implemented on the real device. It should be also possible to simulate this entity for testing the whole data path and control path. So there will be no Xilinx IP in this entity because it is not possible to simulate it outside Xilinx tools. It should be also possible to integrate this entity into larger designs. For example, it would be nice to add CAN and LIN interfaces which are commonly used alongside with Automotive Ethernet.

The Ethernet system should also support building with variable port count. At the moment, we are using the AMD Kria starter kit which provides only two GE ports and one XGE port because the other GE ports are directly connected to the processing system. In the near future, we will design a custom motherboard which will provide between six and eight GE ports and two XGE ports. For that reason, I decided to create an entity called GE port for GE related components and an entity called XGE port for XGE related components. The Ethernet system can then instantiate any number of them.

3.15.2 Solutions

Two different ways of grouping the entities were considered: thinking about one pair of ports either as two **channels** (channel A goes from port 1 to port 2 and channel B goes from port 2 to port 1) or as two **ports** with two **links** (from port 1 to port 2 and from port 2 to port 1) between them.

Using channels

The first idea was to work with channels instead of ports as shown in figure 3.29. The initial motivation was that each channel would be driven by the clock recovered from the received stream as described in section 3.1. Each pair of ports would then be composed of two channels:

- Channel A would receive frames on port 1, filter them, optionally process them and finally transmit them on port 2.
- Channel B would receive frames on port 2, filter them, optionally process them and finally transmit them on port 1.

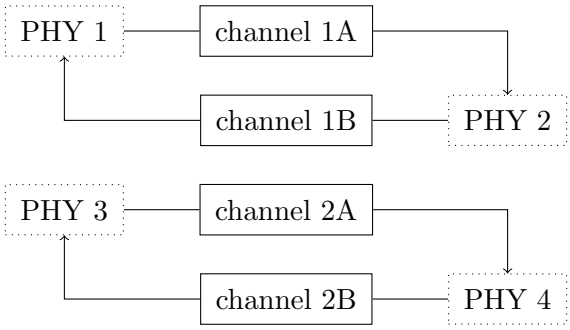

Figure 3.29: Each pair is composed of two channels

This approach worked well in the initial phase of the design when filtering GE stream (see section 3.7) and converting GE stream to XGE stream (see section 3.11) was considered. In the later phase when processing GE stream (see section 3.8) was considered, it started being less practical. Thinking about two channels instead of thinking about two ports was difficult when some frames were removed from the stream by the processor and some frames were introduced to the stream by the processor.

Using ports

The second idea was to work directly with ports as shown in figure 3.30. The motivation was that it would be more intuitive to use. From software perspective, it would make more sense if responding on the same network interface caused a frame to be generated on the same physical port and not on the other physical port of the pair. From human perspective, it would be more convenient to assign numbers to ports rather than to use a more complex system of numbers and letters for channels.

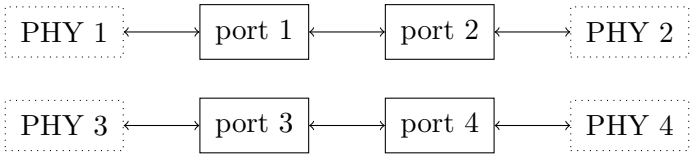

Figure 3.30: Each pair is composed of two ports

3.15.3 GE port

The GE port is implemented in the **ge_port** entity. It instantiates all entities required for one gigabit Ethernet port. It is expected that GE ports are instantiated in pairs and both ports of the pair are connected to each other.

Interfaces

The GE port has the following interfaces that should be connected to the external PHY:

- The **rgmii_rx** interface consists of 5 signals as described in section 3.3.3. It is synchronous to the **rgmii_rx_clk** clock which goes from the PHY to this entity.
- The **rgmii_tx** interface consists of 5 signals as described in section 3.3.3. It is synchronous to the **rgmii_tx_clk** clock which goes from this entity to the PHY.
- The **smi** interface consists of **mdc** output and **mdio** bidirectional signal.

The GE port has the following interfaces that should be connected to other GE/XGE ports:

- The **ge_pass** input and output interfaces should be connected to the other GE port of the pair. They are synchronous to the **ge_clk** clock. The GE interface consists of 8 data bits and 3 control signals as described in section 3.3.5.
- The **ge_divert** input and output interfaces should be connected to the GE/XGE port dedicated for diverting GE streams. They are synchronous to the **ge_clk** clock. The GE interface consists of 8 data bits and 3 control signals as described in section 3.3.5.
- The two **cmp_log** output interfaces should be connected to the GE/XGE port dedicated for logging GE streams. They are synchronous to the **cmp_clk** clock. The CMP interface is the same as the GE interface at the moment.

The GE port has the following interfaces that should be connected to the processing system:

- The **axi** master interface should be connected to the AXI slave port of the processing system. It is used to transport data to and from the memory of the processing system.
- The **conf** slave interface should be converted to AXI interface and connected to the AXI master port of the processing system. It is described in section 3.4.2.

Components

The GE port consists of the following components:

- The **PHY block** (see section 3.5.5) which converts the RGMII interface to the GMII interface in both directions and provides access to the management interface of the PHY.
- The **MAC block** (see section 3.6.5) which converts the GMII interface to the GE interface in both directions.
- The **DMA block** (see section 3.8.6) which allows processing of the GE stream in the processing system by transporting frames to and from the external memory.
- The **CMP block** (see section 3.13.7) which converts the GE stream to the CMP stream.
- The **RX splitter** (see section 3.7.4) which splits the GE stream of received frames coming from the MAC block into the following streams:
  1. The CMP RX stream which continues to the CMP block to be converted from GE frames to CMP payloads and then leaves the port through the **cmp_log_rx** interface.
  2. The DMA RX stream which continues to the DMA block to be transported to the external memory and then processed by the processing system.
  3. The pass output stream which leaves the port through the **ge_pass_out** interface.
  4. The divert output stream which leaves the port through the **ge_divert_out** interface.
- The **TX joiner** (see section 3.9.4) which joins the following streams into the GE stream of transmitted frames going to the MAC block:
  1. The DMA TX stream which comes from the DMA block after it was processed by the processing system and then transported from the external memory.
  2. The pass input stream which enters the port through the **ge_pass_in** interface.
  3. The divert input stream which enters the port through the **ge_divert_in** interface.
- The **FB splitter** (see section 3.7.4) which splits the GE stream of transmitted frames coming from the MAC block into the following streams:
  1. The CMP TX stream which continues to the CMP block to be converted from GE frames to CMP payloads and then leaves the port through the **cmp_log_tx** interface.
  2. The DMA FB stream which continues to the DMA block to be processed by the processing system in order to extract TX timestamps.

The structure is also shown in figure 3.31.

Management

The GE port provides access to the following components via the management interface:

- The **RX splitter** (see section 3.7.4) is mapped from **0x0000** to 0x7FFF.
- The **FB splitter** (see section 3.7.4) is mapped from **0x8000** to 0xBFFF.
- The **CMP block** is mapped from **0xC000** to 0xCFFF.
  - The **CMP RX packager** (see section 3.13.3) is mapped from **0xC000** to 0xC0FF.
  - The **CMP TX packager** (see section 3.13.3) is mapped from **0xC100** to 0xC1FF.

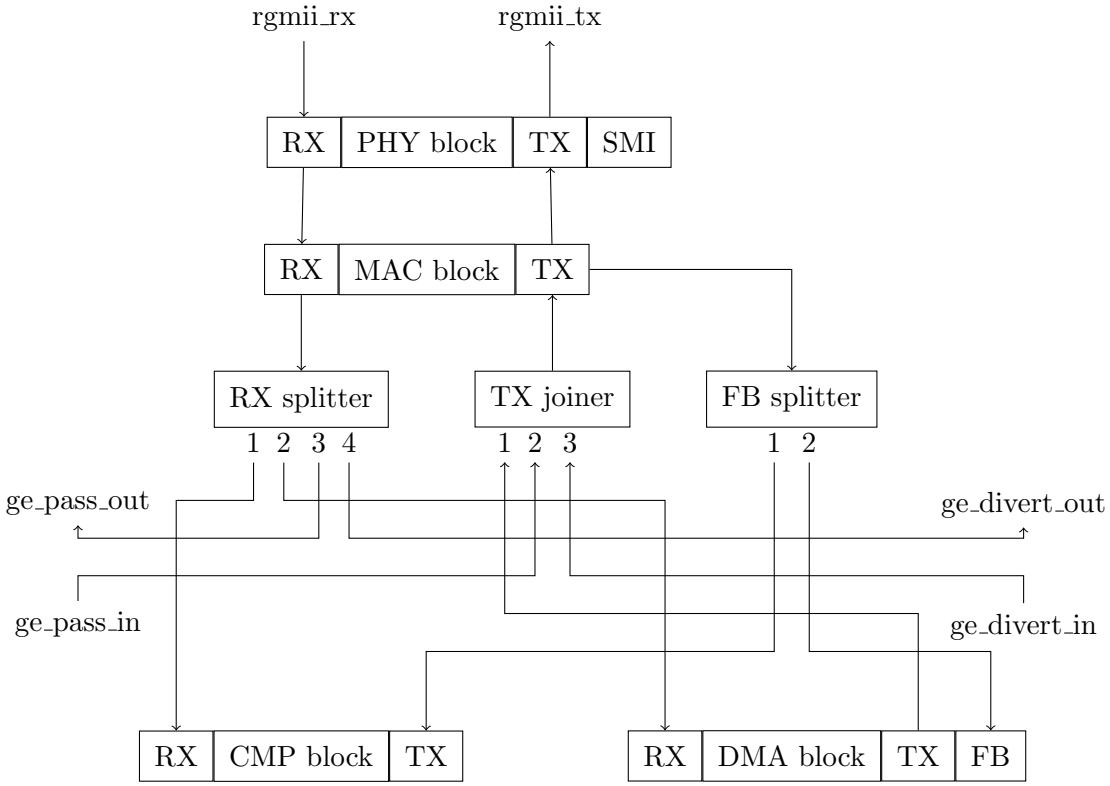

Figure 3.31: Structure of GE port

- The **DMA block** is mapped from **0xD000** to 0xDFFF.
  - The **DMA receiver** (see section 3.8.3) is mapped from **0xD000** to 0xD0FF.
  - The **DMA transmitter** (see section 3.8.4) is mapped from **0xD100** to 0xD1FF.
  - The **DMA feedback** (see section 3.8.5) is mapped from **0xD200** to 0xD2FF.
- The **MAC block** is mapped from **0xE000** to 0xEFFF.
  - The **MAC receiver** (see section 3.6.3) is mapped from **0xE000** to 0xE0FF.
  - The **MAC transmitter** (see section 3.6.4) is mapped from **0xE100** to 0xE1FF.
- The **PHY block** is mapped from **0xF000** to 0xFFFF.
  - The **SMI controller** (see section 3.14.3) is mapped from **0xF000** to 0xF0FF.

3.15.4 XGE port

The XGE port is implemented in the **xge_port** entity. It instantiates all entities required for one ten gigabit Ethernet port. It is expected that single XGE port is connected to multiple GE ports for logging and diverting frames.

Parameters

The XGE port has the following generic parameters:

- The **cmp_input_count** determines the count of the CMP input interfaces. It is usually set to the count of the GE ports from which logging should be possible multiplied by two (because each port generates two CMP streams).

- The **ge_input_count** determines the count of the GE input interfaces. It is usually set to the count of the GE ports from which diverting should be possible.
- The **ge_output_count** determines the count of the GE output interfaces. It is usually set to the count of the GE ports to which diverting should be possible.

Interfaces

The XGE port has the following interfaces that should be connected to the external PHY:

- The **i2c** interface consists of **scl** output and **sda** bidirectional signal.

The XGE port has the following interfaces that should be connected to the internal PHY:

- The **xgmii_rx** interface consists of 64 data bits and 8 control bits as described in section 3.3.4. It is synchronous to the **xge_rx_clk** clock which goes from the PHY to this entity.
- The **xgmii_tx** interface consists of 64 data bits and 8 control bits as described in section 3.3.4. It is synchronous to the **xge_tx_clk** clock which goes from the PHY to this entity.

The XGE port has the following interfaces that should be connected to other GE/XGE ports:

- The **ge** input and output interfaces should be connected to all GE ports for diverting GE streams. They are synchronous to the **ge_clk** clock. The GE interface consists of 8 data bits and 3 control signals as described in section 3.3.5.
- The **cmp** input interfaces should be connected to all GE ports for logging GE streams. They are synchronous to the **ge_clk** clock. The CMP interface is the same as the GE interface at the moment.

The XGE port has the following interfaces that should be connected to the processing system:

- The **conf** slave interface should be converted to AXI interface and connected to the AXI master port of the processing system. It is described in section 3.4.2.

Components

The XGE port consists of the following components:

- The **PHY block** (see section 3.14.4) which provides access to the management interface of the external PHY.
- The **MAC block** (see section 3.10.5) which converts the XGMII interface to the XGE interface in both directions.
- The **GE splitter** (see section 3.12.3) which splits the XGE stream of received frames coming from the MAC block into the GE streams leaving the XGE port through the **ge_output** interfaces. It also performs performs the conversion from the XGE interface to the GE interface.
- The **GE joiner** (see section 3.11.3) which joins the GE streams entering the XGE port through the **ge_input** interfaces into the first XGE stream going to the TX joiner. It also performs the conversion from the GE interface to the XGE interface.
- The **CMP joiner** (see section 3.11.3) which joins the CMP streams entering the XGE port through the **cmp_input** interfaces into the CMP stream going to the CMP aggregator. It also performs the conversion from the GE interface to the XGE interface.

- The **CMP aggregator** (see section 3.13.4) which combines CMP payloads into CMP packages and generates the second XGE stream going to the TX joiner.
- The **TX joiner** which joins the following streams into the XGE stream of transmitted frames going to the MAC block:
  1. The GE input stream which comes from the GE joiner.
  2. The CMP input stream which comes from the CMP joiner after going through the CMP aggregator.

The structure is also shown in figure 3.32.

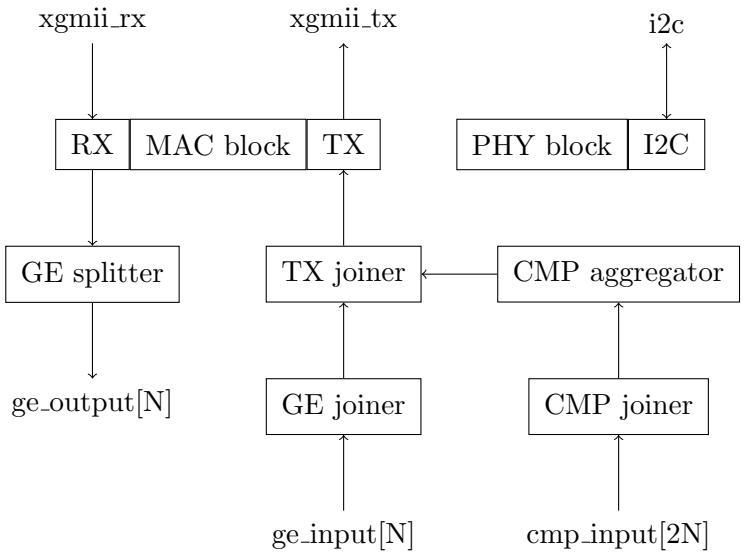

Figure 3.32: Structure of XGE port

Management

The XGE port provides access to the following components via the management interface:

- The **CMP block** is mapped from **0xC000** to 0xCFFF.
  - The **CMP aggregator** (see section 3.13.4) is mapped from **0xC000** to 0xC0FF.
- The **DMA block** is mapped from **0xD000** to 0xDFFF. It is reserved for future use.
- The **MAC block** is mapped from **0xE000** to 0xEFFF.
  - The **MAC receiver** (see section 3.10.3) is mapped from **0xE000** to 0xE0FF.
  - The **MAC transmitter** (see section 3.10.4) is mapped from **0xE100** to 0xE1FF.
- The **PHY block** is mapped from **0xF000** to 0xFFFF.
  - The **I2C controller** (see section 3.14.4) is mapped from **0xF000** to 0xF0FF.

3.15.5 Ethernet system

The Ethernet system is implemented in the **eth_system** entity. It instantiates multiple GE ports described in section 3.15.3 and single XGE port described in section 3.15.4 in order to provide all the functions described in this chapter. The count of GE ports is given by the **port_count** generic parameter and must be even because they are instantiated and connected in pairs (port 1 is connected to port 2, port 3 is connected to port 4 and so on).

For simulation, this should be used as the top level entity. It does not include any Xilinx IP so it can be simulated without any problems. It does however include clock and memory primitives which can be simulated after building them as shown in section 4.1. The simulation of the Ethernet system is described in chapter 4.

For implementation, this cannot be used as the top level entity. It requires Xilinx IP for interfacing with the XGE transceiver and interfacing with the processing system as described in section 5.2.2. The implementation of the Ethernet system is described in chapter 5.

Clocks

The Ethernet system has the following clocks which must be generated by the parent entity:

- The **clk_125** clock must have frequency of 125 MHz. It is used for 1000 Mb/s Ethernet.
- The **clk_12_5** clock must have frequency of 12.5 MHz. It is used for 100 Mb/s Ethernet.
- The **axi_clk** clock may have any viable frequency. It is used for AXI master and slave interfaces.

Interfaces

The Ethernet system has the following GE interfaces which should be connected to the GE external transceivers (Ethernet PHYs in our case):

- The array of **rgmii_rx** interfaces. Each of them consists of 5 signals as described in section 3.3.3. Each of them is synchronous to the respective element of the **rgmii_rx_clk** array which goes from the transceiver to this entity.
- The array of **rgmii_tx** interfaces. Each of them consists of 5 signals as described in section 3.3.3. Each of them is synchronous to the respective element of the **rgmii_tx_clk** array which goes from this entity to the transceiver.
- The array of **smi** interfaces. Each of them consists of **mdc** output and **mdio** bidirectional signal.

The Ethernet system has the following XGE interfaces which should be connected to the XGE external transceivers (SFP+ modules in our case):

- The **i2c** interface consists of **scl** output and **sda** bidirectional signal.

The Ethernet system has the following XGE interfaces which should be connected to the XGE internal transceivers (Xilins IPs in our case):

- The **xgmii_rx** interface consists of 64 data bits and 8 control bits as described in section 3.3.4. It is synchronous to the **xgmii_rx_clk** clock.
- The **xgmii_tx** interface consists of 64 data bits and 8 control bits as described in section 3.3.4. It is synchronous to the **xgmii_tx_clk** clock.

The Ethernet system has the following management interfaces which should be connected to the processing system:

- The **axi_s_m2s** and **axi_s_s2m** slave interface should be connected to the AXI master port of the processing system. It is used to configure the Ethernet system from the processing system. It is synchronous to the **axi_clk** clock.
- The **axi_m_m2s** and **axi_m_s2m** master interface should be connected to the AXI slave port of the processing system. It is used to transport data to and from the memory of the processing system. It is synchronous to the **axi_clk** clock.

Components

The Ethernet system consists of the following components:

- The **GE port** (see section 3.15.3) which instantiates all entities required for one gigabit Ethernet port. The pass interfaces are connected together (port 1 with port 2, port 3 with port 4 and so on). The divert interfaces are connected to the XGE port. The CMP interfaces are also connected to the XGE port. The AXI master interfaces are connected to the AXI master switch.
- The **XGE port** (see section 3.15.4) which instantiates all entities required for one ten gigabit Ethernet port. The GE interfaces are connected to all the GE ports. The CMP interfaces are also connected to all the GE ports.
- The **64 bit counter** which provides common time base for timestamping received and transmitted frames in all the GE ports.
- The **clock multiplexer** which switches between 125 MHz and 12.5 MHz clocks to support either 1000 Mb/s or 100 Mb/s Ethernet.
- The **AXI master switch** which performs simple arbitration between AXI masters of the GE ports which all share one AXI master interface.
- The **AXI slave converter** which performs conversion between AXI slave interface and internal configuration interface as described in section 3.4.2.

The structure is also shown in figure 3.33.

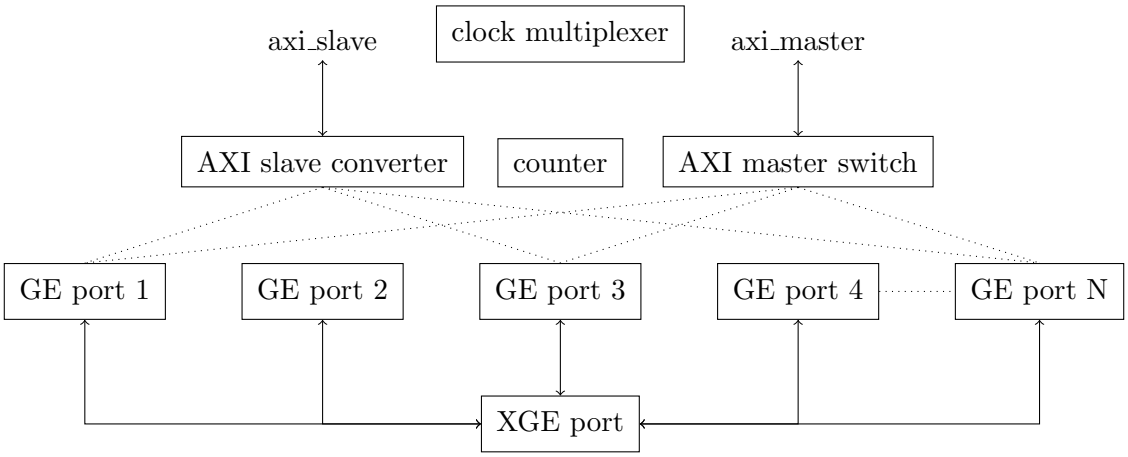

Figure 3.33: Structure of Ethernet system

Management

The Ethernet system provides management access to the following components by the standard management interface:

- The **Ethernet system** (see below) is mapped from **0x00000** to 0x0FFFF.
- The **GE port** (see section 3.15.3) is mapped from **0xN0000** to 0xNFFFF where N refers to the number of the port (the first port has number 1 and the last port has number equal to the count of the GE ports).
- The **XGE port** (see section 3.15.4) is mapped from **0xM0000** to 0xMFFFF where M is equal to the count of the GE ports plus 1.

The following registers are available for reading in the Ethernet system:

- The **system ID** register (0x00) identifies the system. It can be used to check whether the Ethernet system is correctly loaded and accessible via the management interface. It should be always read as 0xAEAEAEAE.
- The **build date** register (0x04) reports the date of the hardware build. It can be used to check whether the correct Ethernet system is loaded.
- The **port count** register (0x08) reports the port count of the hardware build.
- The **timestamp lower** register (0x10) allows reading the lower part of the counter used for timestamping.
- The **timestamp upper** register (0x14) allows reading the upper part of the counter used for timestamping.

The following registers are available for writing in the Ethernet system:

- The **clock control** register (0x20) allows setting the clock multiplexer:
  - Bit 1 enables the clock when set to 1 and disables it when set to 0.
  - Bit 2 selects the frequency of the clock. The 125 MHz clock is used when set to 0. The 12.5 MHz clock is used when set to 1.

Verification

The Ethernet system was verified in simulation as described in section 4.5.

Validation

The Ethernet system was validated on real device as part of the tests described in section 8.

Chapter 4

Simulating the Hardware

Developing our device would not be possible without simulation. This chapter is devoted to the simulation of the hardware. It is divided into the following sections:

- In section 4.1, I show how to set up the software environment for the simulation.
- In section 4.2, I show how to generate the clocks with static or dynamic frequency.
- In section 4.3, I show how to simulate the interfaces (AXI, GE/XGE, GMII/XGMII...).
- In section 4.4, the simulations of individual components and blocks are described.
- In section 4.5, the simulations of the whole Ethernet system are described.

4.1 Preparing the environment

For the simulation of the hardware, the following software components are needed:

- A **simulator** which understands VHDL, simulates the testbench and generates the waveform.
- A **waveform viewer** which presents the waveform in human readable format.

4.1.1 GHDL

As for the simulator, I decided to use GHDL (available from `https://github.com/ghdl/ghdl`). It is an open source simulator of VHDL which compiles the design into machine code instead of interpreting it for better performance. It is easy to use thanks to its command line interface.

Installing

On Linux, it can be either installed from the standard package repository or built from sources hosted on GitHub. I used the following command which can be used on Debian or Ubuntu based systems:

```
apt install ghdl
```

When I wanted to try a different version or a different backend, I had to perform a custom build. After installing all required prerequisities, I used the following commands according to the manual:

```
./configure --with-llvm-config --prefix=/usr/local
make
make install
```

On Windows, it can be downloaded from GitHub. The ZIP archive includes a compiled binary which can be used immediately after extracting the archive to any directory. I recommend adding the directory to PATH so that it can be always used from the command line.

Compiling vendor primitives

Our design uses several vendor specific primitives to use all different features of the FPGA. For example, they are used for clock management and memory entities. Fortunately, it is possible to simulate them using GHDL and other simulators. Due to licensing issues, their sources cannot be distributed with the simulators so they must be compiled by the end user.

I performed the following steps according to the manual:

1. I located the `vendors` directory with scripts for building the primitives. When performing a custom build, it was located in the `/usr/local/lib/ghdl` directory. When using the standard package repository, I had to download the scripts from GitHub because I could not find them on my system.
2. I edited the `config.sh` script located in the `vendors` directory to provide the installation path of Xilinx Vivado which includes the sources of the Xilinx specific primitives.
3. I changed to the `sim` directory inside my project where I will run the simulations later.
4. I started the `compile-xilinx-vivado.sh` script located in the `vendors` directory which took several minutes to compile the primitives.
5. I checked there was the `xilinx-vivado` directory created by the script in the `sim` directory inside my project.

Running simulations

The first step is to make GHDL aware of all files that will be necessary for the simulation. For this purpose, the `-i` command shall be used. Sometimes, I ran this command with a wildcard so that I did not have to specify all the files individually. In most cases, there were only a few new files which had to be added. It should be noted that GHDL remembers the files which have been already added before so this command must be used for new files only. For example, the following command shall be used to add the `ethernet_system.vhdl` file located in the root of the `src` directory:

```
ghdl -i ../src/ethernet_system.vhdl
```

The second step is to compile the source code into machine code which will be run to perform the simulation. For this purpose, the `-m` command shall be used. It builds a dependency tree and compiles all the files in the correct order. It is therefore much more convenient than using other commands which require running them in correct order. It is necessary to run this command after every change in the source code. It automatically detects the changes which have been made since the last build and recompiles only the changed files and the files depending on them. For example, the following command shall be used to compile the `ethernet_system_tb` entity with all its dependencies:

```
ghdl -m ethernet_system_tb
```

The third step is to run the simulation. For this purpose, the `-r` command shall be used. When no other options are given, the simulation runs as long as some signal of the design changes its value. That means the simulation runs indefinitely whenever there is some clock which never stops. To solve this problem, I often used the `--stop-time` option to run the simulation for given time. Later, I started generating the clocks in a more clever way so that they could be stopped when the simulation was complete. I also used the `--wave` option to save the waveform whenever I wanted to analyse it. However, I preferred creating automated testbenches which not only generated the inputs for the unit but also checked the outputs of the unit. For example, the following command shall be used to simulate the `ethernet_system_tb` entity for 10 microseconds and save the waveform:

```
ghdl -r ethernet_system_tb --stop-time=10us --wave=ethernet_system_tb.ghw
```

4.1.2 GTKWave

As for the waveform viewer, I decided to use GTKWave (available from `https://github.com/gtkwave/gtkwave`). It is a GTK+ based waveform viewer which reads GHW files generated by GHDL and allows their viewing.

Installing

On Linux, it can be either installed from the standard package repository or built from sources hosted on GitHub. I used the following command which can be used on Debian or Ubuntu based systems:

```
apt install gtkwave
```

On Windows, it can be downloaded from SourceForge. The ZIP archive includes a compiled binary which can be used immediately after extracting the archive to any directory. I recommend adding the directory to PATH so that it can be always used from the command line.

4.2 Generating the clocks

To make a simple testbench, the UUT (unit under test) must be instantiated and all the clocks must be generated. This section shows a simple and a more complex way to generate a clock for simulation.

The simple way

The most simple way which became my favorite to generate a clock is to use the following assignment:

```
clk <= not clk after 10 ns;
```

For this to work, the clock signal must have the initial value set to `'0'` or `'1'`. The assignment causes the signal to flip its value every 10 nanoseconds. The period of the clock is therefore 20 nanoseconds and the frequency is 50 MHz.

The more complex way

A more complex way to generate a clock which could be enabled, disabled, speeded up or slowed down throughout the simulation was required by more advanced testbenches. Another advantage of stopping the clock when the testbench finished was that the simulation was stopped automatically without prior setting of the simulation time. When using the simple way to generate a clock, the simulation never stops because the clock keeps running indefinitely. With those requirements in mind, I put together the following process:

```
loop
    if enable then
        wait for 1000 ns / frequency / 2;
        clock <= '1';
        wait for 1000 ns / frequency / 2;
        clock <= '0';
    else
        wait until enable;
    end if;
end loop;
```

This process could be either put directly into every testbench file which required more complex clock generation or better put into a procedure which could be instantiated in every testbench file. So I created the `clk_generate` procedure which gets the following signals:

- The **clock** output signal where the procedure shall generate the clock.
- The **enable** input signal which controls whether the clock shall be running. When this signal is zero, the clock is stopped and stays at zero. That can be used to stop the testbench by any other process of the testbench.
- The **frequency** input signal which controls the frequency of the clock in megahertz. When this signal changes, the frequency of the clock changes at the next period. That can be used to vary the frequency by any other process of the testbench.

4.3 Simulating the interfaces

Most components of our design have one or more input interfaces which must be driven by the testbench and one or more output interfaces which should be checked by the testbench. Each interface consists of several signals having from five to hundred bits in total. Each interface also uses a specific protocol for transporting frames, reading from or writing to memory and so on. It would be painful to implement and maintain the protocol in every testbench separately.

It turned out that VHDL functions and procedures could be used very conventiently to simplify the development of testbenches significantly. The difference between a function and a procedure in VHDL is as follows:

- A function in VHDL returns a value given a set of input constants. It may not use wait statements or change values of any signals or variables.
- A procedure in VHDL gets a set of input and output signals and variables and does not return any value. It may use wait statements and change values of the output signals and variables.

4.3.1 Ethernet interfaces

Procedures can be used to transmit and receive frames on Ethernet interfaces in order to create automated testbenches in a very simple manner.

To transmit a frame on one of the Ethernet interfaces, the respective procedure gets the following signals and constants:

- The **clock** input signal. The procedure drives the Ethernet interface synchronous to this clock.
- The **port** output signal which represents the Ethernet interface and whose type reflects the type of the interface.
- The **data** input constant. The transmitted frame will contain this data.

The transmit procedure first generates the start of frame condition which depends on the type of the interface. Then, it transmits all bytes of the data input constant. Finally, it generates the end of frame condition which again depends on the type of the interface.

To receive a frame on one of the Ethernet interfaces, the respective procudere gets the following signals and constants:

- The **clock** input signal. The procedure samples the Ethernet interface synchronous to this clock.
- The **port** input signal which represents the Ethernet interface and whose type reflects the type of the interface.
- The **data** input constant. The received frame must contain exactly this data in order to pass all assertions.

The receive procedure first waits until the start of frame condition which depends on the type of the interface. Then, it checks that every byte of the frame received on the interface matches the corresponding byte of the expected frame given by the data input constant. Finally, it checks that the end of frame condition comes right after the last byte of the expected frame. It reports any mismatch of data and also when the received frame is shorter or longer than the expected frame.

For each type of Ethernet interface used in our design, transmit and receive procedures were implemented to allow automated testbench development. They are described in the following list:

- The `gmii_transmit` and `gmii_receive` procedures can be used to simulate the GMII interface as described in section 3.3.2.
- The `mii_transmit` and `mii_receive` procedures can be used to simulate the GMII interface in MII mode (for 100 Mb/s Ethernet) as described in section 3.3.2.
- The `rgmii_transmit` and `rgmii_receive` procedures can be used to simulate the RGMII interface as described in section 3.3.3.
- The `rgmii_mii_transmit` and `rgmii_mii_receive` procedures can be used to simulate the RGMII interface in MII mode (for 100 Mb/s Ethernet) as described in section 3.3.3.
- The `ge_transmit` and `ge_receive` procedures can be used to simulate the GE interface as described in section 3.3.5. Instead of the data input constant, there is the frame input constant and the footer input constant.

- The `xgmii_transmit` and `xgmii_receive` procedures can be used to simulate the XGMII interface as described in section 3.3.4. In addition to the data input constant, there is also the lane input constant which specifies the lane where the frame starts.
- The `xge_transmit` and `xge_receive` procedures can be used to simulate the XGE interface as described in section 3.3.6.

4.3.2 Management interfaces

Procedures can be used to perform read or write transactions on management interfaces in order to configure the units under test in a very simple manner.

To perform a read transaction on one of the management interfaces, the read procedure needs to get the following items:

- The **clk** (clock) input signal. The procedure drives and samples the management interface synchronous to this clock.
- The **m2s** (master to slave) output signal and **s2m** (slave to master) input signal which represent the management interface and whose type reflects the type of the interface.
- The **address** input constant which sets the address of the first word read by the transaction. For each word, it is incremented by the size of the word.
- The **value** output variable which is set to the value of the word (or words when it is an array) read by the transaction.

To perform a write transaction on one of the management interfaces, the write procedure needs to get the following items:

- The **clk** (clock) input signal. The procedure drives and samples the management interface synchronous to this clock.
- The **m2s** (master to slave) output signal and **s2m** (slave to master) input signal which represent the management interface and whose type reflects the type of the interface.
- The **address** input constant which sets the address of the first word written by the transaction. For each word, it is incremented by the size of the word.
- The **value** input constant whish sets the value of the word (or words when it is an array) written by the transaction.

For each type of management interface used in our design, read and write procedures were implemented to allow automated testbench development. They are described in the following list:

- The `conf_read` and `conf_write` procedures can be used to simulate the custom management interface described in section 3.4.2.
- The `conf_read_bytes` and `conf_write_bytes` procedures can be used to simulate byte access on the custom management interface described in section 3.4.2.
- The `axi32_read` and `axi32_write` procedures can be used to simulate transactions with multiple transfers on the AXI interface with 32 bit data width described in section 3.4.1.
- The `axi32_read1` and `axi32_write1` procedures can be used to simulate transactions with single transfer on the AXI interface with 32 bit data width described in section 3.4.1.

The name of the procedures must be different because VHDL does not differentiate between a word and an array of words.

4.4 Simulating the components

Every component of the design was verified in simulation either one by one or in combination with other components. This section describes the testbenches used to verify all key components of the design.

4.4.1 GE MAC block

The MAC block designed in section 3.6.5 was verified in two phases. First, the receiver and the transmitter were simulated in separate testbenches as shown in figure 4.1. Second, the receiver and the transmitter were put into a common testbench and connected together via the GE interface as shown in figure 4.2.

It was necessary to verify operation under extreme conditions which are represented by the 100 ppm difference in clock frequencies in this case. It was also necessary to verify both functional modes of the receiver and the transceiver. For that reason, the testbenches used the following pattern:

1. The clock frequencies were set to simulate the 1 Gb/s Ethernet.
   - The `ge_clk` frequency was set to 125 MHz.
   - The `gmii_clk` frequency was set to 125 MHz minus 100 ppm.
2. The receiver/transmitter was enabled and switched to the full mode.
3. The frame generator transmitted N frames of 100 bytes. The gap between every two frames was set to simulate the minimal inter frame gap.
4. The frame checker received N frames of 100 bytes and reported any problems.
5. The clock frequencies were set to simulate the 100 Mb/s Ethernet.
   - The `ge_clk` frequency was set to 12.5 MHz.
   - The `gmii_clk` frequency was set to 25 MHz minus 100 ppm.
6. The receiver/transmitter was switched to the half mode.
7. The frame generator transmitted N frames of 100 bytes. The gap between every two frames was set to simulate the minimal inter frame gap.
8. The frame checker received N frames of 100 bytes and reported any problems.
9. The testbench reported OK and stopped the clocks.

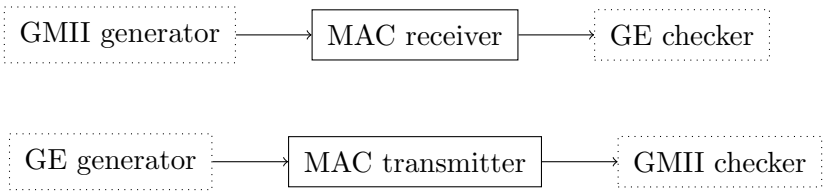

Figure 4.1: Separate testbenches for the MAC block

Figure 4.2: Common testbench for the MAC block

4.4.2 GE DMA block

The DMA block designed in section 3.8.6 was also verified in two phases. First, the receiver and the transmitter were simulated in separate testbenches with an empty memory as shown in figure 4.3. Second, the receiver and the transmitter were put into a common testbench with a shared memory and configured to use the shared memory as the data buffer as shown in figure 4.4.

DMA receiver

The separate testbench for the DMA receiver was driven by the following process:

1. The receiver was enabled and the data buffer was set to start at 0x2000 with size of 0x400.
2. The frame generator transmitted N frames of various length ranging from 100 to 300 bytes.
3. The receiver generated AXI write transactions which had to be checked manually.

DMA transmitter

The separate testbench for the DMA transmitter was driven by the following process:

1. The transmitter was enabled and the data buffer was set to start at 0x2000 with size of 0x400.
2. The transmitter was commanded via the management interface to transmit N frames of various length ranging from 100 to 300 bytes.
3. The transmitter generated AXI read transactions which could be checked manually.
4. The frame checker received N frames of various length ranging from 100 to 300 bytes and reported any problems. The data of the received frames was expected to be zero because the memory was empty. The length of the received frames was expected to match the transmitted frames.

DMA block

The common testbench for the DMA block was driven by the following process:

1. The receiver and the transmitter were both enabled and their data buffer was set to start at 0x2000 with size of 0x400.
2. The frame generator transmitted N frames of various length ranging from 100 to 300 bytes.
3. The controller polled the descriptor count register of the receiver every 100 nanoseconds. When a descriptor was available, it was read from the receiver and written to the transmitter.
4. The frame checker received N frames of various length ranging from 100 to 300 bytes and reported any problems. The received frames were expected to match the transmitted frames.

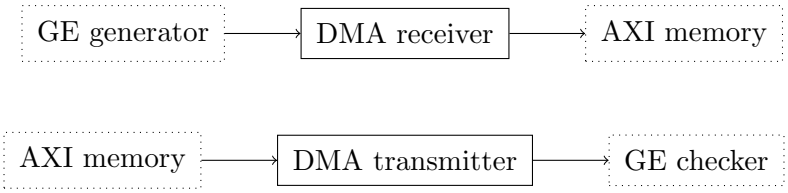

Figure 4.3: Separate testbenches for the DMA block

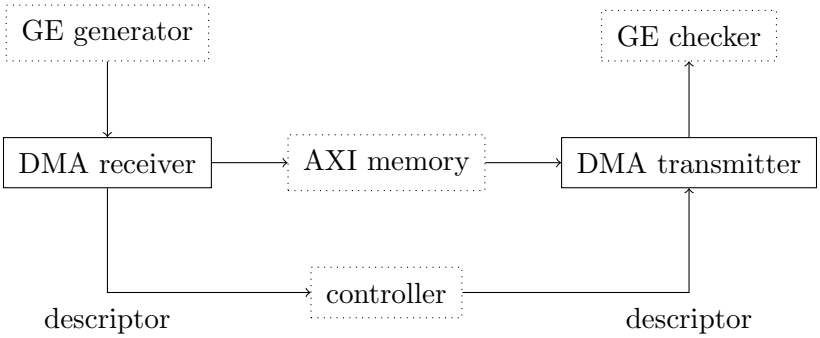

Figure 4.4: Common testbench for the DMA block

4.4.3 GE matcher

The matcher designed in section 3.7.3 was verified by setting up its configuration via the management interface, generating frames on the input and checking the match and mismatch outputs. The configuration and the content of the frames were chosen carefully in order the check correct function in several possible situations.

Setup

The matcher was first set up with the following commands:

1. It was enabled and the length was set to 3 bytes.
2. The value array was set to [0x11, 0x22, 0x33].
3. The mask array was set to [0x0f, 0xff, 0xf0].

That means only the following parts of the frame should have effect on the result: the second nibble of the 1st byte, the whole 2nd byte, the first nibble of the 3rd byte.

Input

The frame generator then generated the following frames:

1. Frame starting with [0x11, 0x22, 0x33] was expected to **match**.
2. Frame starting with [0x12, 0x22, 0x33] was expected to **mismatch**. The second nibble of the 1st byte does not match.
3. Frame starting with [0x11, 0x23, 0x33] was expected to **mismatch**. The second nibble of the 2nd byte does not match.
4. Frame starting with [0x11, 0x22, 0x34] was expected to **match**. The second nibble of the 3rd byte is not significant.
5. Frame starting with [0x01, 0x22, 0x33] was expected to **match**. The first nibble of the 1st byte is not significant.

Output

The match and mismatch outputs were finally checked manually.

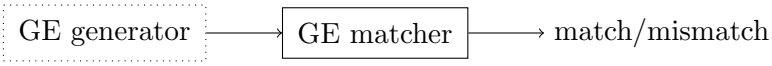

Figure 4.5: Testbench for the GE matcher

4.4.4 GE splitter

The splitter designed in section 3.7.4 was verified by setting up its configuration via the management interface, generating frames on the input and checking frames on the outputs. Given the fact that the matcher had been already verified by the previous testbench, the goal was to verify the action setup which is handled by the splitter and not the value and mask setup which is handled by the matcher.

Setup

The splitter was first set up with the following rules:

1. Frames with first 16 bytes matching the sequence [**0x10**, 0x11, 0x12, ...] should be passed to **output 1** (action is equal to 0b01);
2. Frames with first 32 bytes matching the sequence [**0x20**, 0x21, 0x22, ...] should be passed to **output 2** (action is equal to 0b10).
3. Frames with first 48 bytes matching the sequence [**0x30**, 0x31, 0x32, ...] should be passed to **both outputs** (action is equal 0b11).
4. Other frames should be dropped (default action is equal to 0b00).

Input

The frame generator then generated the following frames in an infinite loop:

1. Frame of 100 bytes starting with the sequence [**0x10**, 0x11, 0x12, ...] which was expected to appear on output 1 only.
2. Frame of 200 bytes starting with the sequence [**0x20**, 0x21, 0x22, ...] which was expected to appear on output 2 only.
3. Frame of 300 bytes starting with the sequence [**0x30**, 0x31, 0x32, ...] which was expected to appear on both outputs.
4. Frame of 400 bytes starting with the sequence [**0x40**, 0x41, 0x42, ...] which was not expected on any output.

Output

The frame checkers received the frames and reported any problems.

4.4.5 GE joiner

The joiner designed in section 3.9.4 was verified by generating frames on the inputs and checking frames on the output. The gap between the frames was set to the minimum so that the full throughput of both the inputs and the output was used. The count and length of the frames were chosen low enough so that the internal buffer did not overflow.

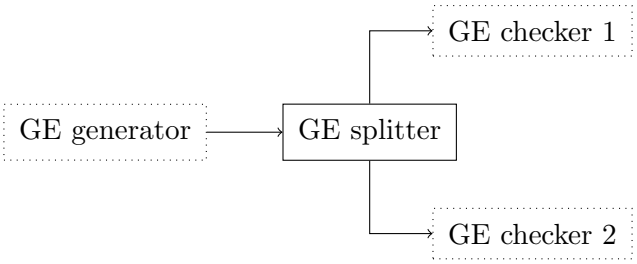

Figure 4.6: Testbench for the GE splitter

Input

The frame generators generated the following frames in a loop:

1. Frames of 100 bytes starting with the sequence [**0x10**, 0x11, 0x12, ...] were transmitted on input 1 with gap of 12 bytes between them.
2. Frames of 200 bytes starting with the sequence [**0x20**, 0x21, 0x22, ...] were transmitted on input 2 with gap of 12 bytes between them.
3. Frames of 300 bytes starting with the sequence [**0x30**, 0x31, 0x32, ...] were transmitted on input 3 with gap of 12 bytes between them.

Output

The frame checker received the frames and reported any problems.

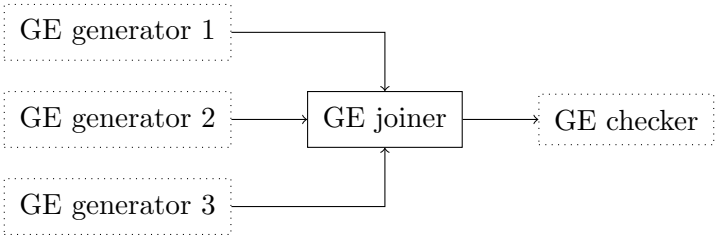

Figure 4.7: Testbench for the GE joiner

4.4.6 XGE MAC transmitter

The MAC transmitter designed in section 3.10.4 was verified by generating XGE frames on the XGE port and checking XGMII frames on the XGMII port. The testbench is shown in figure 4.8.

For every length from 81 to 90 bytes, the following was done:

1. The frame generator transmitted a frame of that length on the XGE port.
2. The frame checker received a frame of that length on the XGMII port and reported any problem.

Figure 4.8: Testbench for the XGE MAC transmitter

4.4.7 XGE MAC receiver

The MAC receiver designed in section 3.10.3 was verified by generating XGMII frames on the XGMII port and checking XGE frames on the XGE port. The testbench is shown in figure 4.9.

For every length from 81 to 90 bytes, the following was done:

1. The frame generator transmitted a frame of that length on the XGMII port beginning in lane 0.
2. The frame generator transmitted a frame of that length on the XGMII port beginning in lane 4.
3. The frame checker received two identical frames of that length on the XGE port and reported any problems.

Figure 4.9: Testbench for the XGE MAC receiver

4.4.8 GE–XGE buffer

The buffer designed in section 3.11.2 was verified by generating GE frames on the GE input port and checking XGE frames on the XGE output port. The testbench is shown in figure 4.10.

Input

The frame generator generated the following frames on the GE input port in a loop:

1. Frame of 100 bytes starting with **0x10** was transmitted with gap of 12 bytes after it.
2. Frame of 200 bytes starting with **0x20** was transmitted with gap of 12 bytes after it.

Output

The frame checker received the frames on the XGE output port and reported any problems.

Figure 4.10: Testbench for the GE–XGE buffer

4.4.9 XGE–GE buffer

The buffer designed in section 3.12.2 was verified by generating XGE frames on the XGE input port and checking GE frames on the GE output port. The testbench is shown in figure 4.11.

Input

The frame generator generated the following frames on the XGE input port in a loop:

1. Frame of 100 bytes starting with **0x10** was transmitted with gap of 3 words (24 bytes) after it.
2. Frame of 200 bytes starting with **0x20** was transmitted with gap of 3 words (24 bytes) after it.

Output

The frame checker received the frames on the GE output port and reported any problems.

Figure 4.11: Testbench for the XGE–GE buffer

4.4.10 GE–XGE joiner

The joiner designed in section 3.11.3 was verified by generating GE frames on the GE input ports and checking XGE frames on the XGE output port. The gap between the GE frames was set to the minimum so that the full throughput of the GE input ports was used. The throughput of the XGE output port is much higher than the total throughput of the GE input ports so the internal buffers can never overflow.

Input

The total length of the frames was chosen the same on both inputs so that the order of the frames on the output was always the same.

The first frame generator generated the following frames in a loop:

1. Frame of 100 bytes starting with **0x10** was transmitted with gap of 12 bytes after it.
2. Frame of 200 bytes starting with **0x20** was transmitted with gap of 12 bytes after it.

The second frame generator generated the following frames in a loop:

1. Frame of 200 bytes starting with **0x20** was transmitted with gap of 12 bytes after it.
2. Frame of 100 bytes starting with **0x10** was transmitted with gap of 12 bytes after it.

Output

The frame checker received the frames on the output and reported any problems.

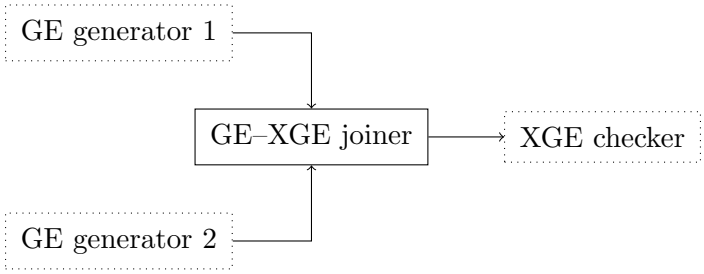

Figure 4.12: Testbench for the GE–XGE joiner

4.4.11 GE CMP packager

The CMP packager designed in section 3.13.3 was verified by setting up its configuration via the management interface, generating GE frames on the input port and checking CMP payloads on the output port. The gap between the frames was set to the minimum so that the full throughput of the input port was used. The testbench is shown in figure 4.13.

Setup

The CMP packager was first set up with the following commands:

1. It was enabled and the interface ID was set to 0x12345678.

Input

The frame generator then transmitted the following frames in a loop:

1. Frame of 100 bytes starting with **0x10** was transmitted with gap of 12 bytes after it.
2. Frame of 200 bytes starting with **0x20** was transmitted with gap of 12 bytes after it.

Output

The payload checker finally received the payloads and reported any problems. The expected payload was composed according to the specification of the CMP protocol and the configuration above for every generated frame. The timestamp was set to a known constant by the frame generator.

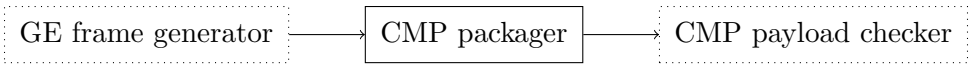

Figure 4.13: Testbench for the CMP packager

4.4.12 XGE CMP aggregator

The CMP aggregator designed in section 3.13.4 was verified by setting up its configuration via the management interface, generating CMP payloads on the input port and checking CMP packages on the output port. The testbench is shown in figure 4.14.

Setup

The CMP aggregator was first set up with the following commands:

1. It was enabled, the maximal length was set to 1000 bytes (see below for reason) and the maximal delay was set to 100 microseconds (does not matter).
2. The destination MAC address were set to A1:A2:A3:A4:A5:A6 (to check byte order) via byte access to the respective memory.
3. The source MAC address was set to B1:B2:B3:B4:B5:B6 (to check byte order) via byte access to the respective memory.
4. The device ID was set to 0x1234 and the stream ID was set to 0xFF.

Input

The payload generator then transmitted payloads of 100 bytes with gaps of 5 words (40 bytes) in a loop. Having set the maximal length to 1000 bytes, every package should contain exactly 10 payloads of 100 bytes.

Output

The package checker finally received the packages and reported any problems. The expected package was composed according to the specification of the CMP protocol and the configuration above for every generated group of payloads. The stream counter was incremented for every expected package.

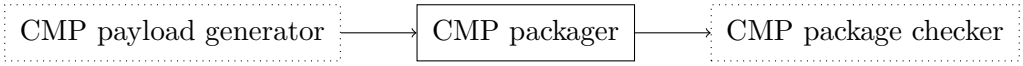

Figure 4.14: Testbench for the CMP aggregator

4.4.13 XGE combinator

The XGE combinator designed in section 3.13.5 was verified by generating short XGE frames of various length on the input port and checking long XGE frames on the output port. The testbench is also shown in figure 4.15.

For every length from 21 to 31 bytes, the following was done:

1. The frame generator transmitted 11 frames of that length on the input port and asserted the terminate signal for one clock cycle.
2. The frame checker received 1 frame created by combining 11 frames of that length on the output port and reported any problems.

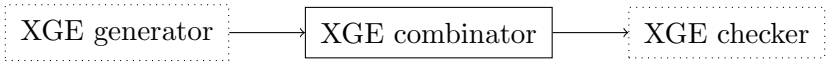

Figure 4.15: Testbench for the XGE combinator

4.5 Simulating the whole system

Sometimes, it happened that the components were working correctly when they were simulated separately but they were behaving incorrectly when they were connected together due to various reasons. It was therefore convenient to simulate them all together in order to debug the problem. This section describes the testbenches used to verify the whole Ethernet system designed in section 3.15.5.

4.5.1 GE testbench

The GE testbench has been designed to test whether passing frames from one GE port to the other GE port of the pair works correctly in both 1 Gb/s and 100 Mb/s modes. After setting up the system via the management interface, the frame generator transmits N frames of 100 bytes on the GE1 port and the frame checker receives the same N frames of 100 bytes on the GE2 port. The frame count is set to 10 frames by default but can be changed by setting the `frames` generic parameter when running the simulation.

Coverage

Every frame must pass through the following entities in order to get from one GE port to the other GE port of the pair so their correct function and cooperation is verified by this testbench:

- RGMII receiver
- MAC receiver
- Splitter
- Joiner
- MAC transmitter
- RGMII transmitter

Setup (1 Gb/s)

The Ethernet system is configured by the testbench as follows for 1 Gb/s Ethernet:

- The frequency of the RGMII clock generated by the testbench is set to 125 MHz minus 100 ppm.
- The clock mulxiplexer of the Ethernet system is set to use the 125 MHz clock as the GE clock.
- The MAC receiver of the GE port 1 is enabled and switched to full mode.
- The MAC transmitter of the GE port 2 is enabled and switched to full mode.
- The RX splitter of the GE port 1 is set to pass all frames to the other GE port by default.

Setup (100 Mb/s)

The Ethernet system is configured by the testbench as follows for 100 Mb/s Ethernet:

- The frequency of the RGMII clock generated by the testbench is set to 25 MHz plus 100 ppm.
- The clock mulxiplexer of the Ethernet system is set to use the 12.5 MHz clock as the GE clock.
- The MAC receiver of the GE port 1 is enabled and switched to half mode.
- The MAC transmitter of the GE port 2 is enabled and switched to half mode.
- The RX splitter of the GE port 1 is set to pass all frames to the other GE port by default.

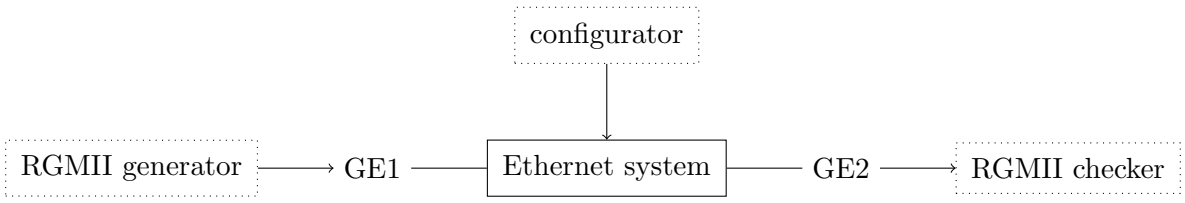

Figure 4.16: GE testbench of the Ethernet system

4.5.2 GE–XGE testbench

The GE–XGE testbench has been designed to test whether diverting frames from the GE ports to the XGE port works correctly. After setting up the system via the management interface, the frame generator transmits N frames of 100 bytes on the GE1 port and the frame checker receives the same N frames of 100 bytes on the XGE port.

Coverage

Every frame must pass through the following entities in order to get from one of the GE ports to the XGE port so their correct function and cooperation is verified by this testbench:

- RGMII receiver
- MAC receiver
- Splitter
- GE–XGE joiner
- XGE joiner
- XGE MAC transmitter

Setup

The Ethernet system is configured by the testbench as follows:

- The MAC receiver of the GE port 1 is enabled.
- The MAC transmitter of the XGE port is enabled.
- The RX splitter of the GE port 1 is set to divert all frames to the XGE port by default.

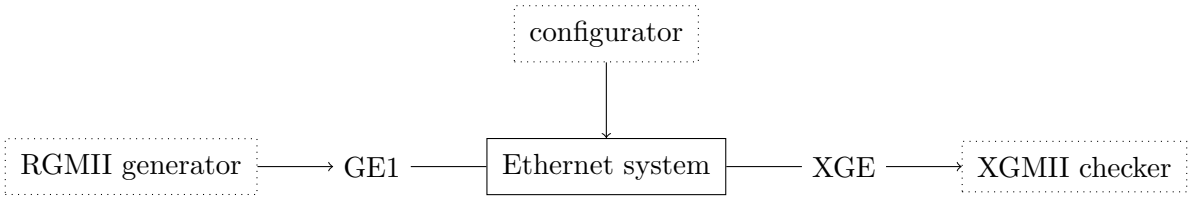

Figure 4.17: GE–XGE testbench of the Ethernet system

4.5.3 CMP testbench

The CMP testbench has been designed to test whether logging frames from the GE ports to the XGE port works correctly. Specifically, it checks whether packaging received and transmitted frames into CMP payloads and also aggregating the payloads into CMP packages works correctly. After setting up the system via the management interface, the frame generator transmits N frames of 100 bytes on the GE1 port. The checker has not been implemented yet so the CMP packages must be inspected manually.

Coverage

Every frame must pass through the following entities in order to get from one of the GE ports to the XGE port so their correct function and cooperation is verified by this testbench:

- RGMII receiver
- MAC receiver
- Splitter

- CMP packager
- GE–XGE joiner
- CMP aggregator
- XGE joiner
- XGE MAC transmitter

Setup

The Ethernet system is configured by the testbench as follows:

- The CMP aggregator of the XGE port is enabled.
- The MAC receiver of the GE port 1 is enabled.
- The RX splitter of the GE port 1 is set to log and pass all frames by default.
- The CMP RX packager of the GE port 1 is enabled.
- The MAC transmitter of the GE port 2 is enabled.
- The FB splitter of the GE port 2 is set to log all frames by default.
- The CMP TX packager of the GE port 2 is enabled.

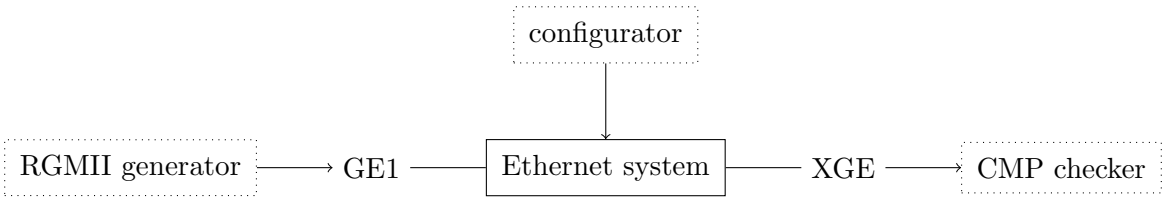

Figure 4.18: CMP testbench of the Ethernet system

Chapter 5

Implementing the Hardware

After testing all hardware components in simulation, the next step was to load them into the the programmable logic. This chapter is devoted to the implementation of the hardware. It is divided into the following sections:

- In section 5.1, I show how to set up the software environment for the implementation.
- In section 5.2, I show how to create the build script for using Vivado in non-project mode.
- In section 5.3, I show how to define the physical constraints for pin assignment.
- In section 5.4, I show how to define the timing constraints for timing analysis.

The process

In general, the implementation process consists of the following steps:

1. **Synthesis** translates from the entities described in VHDL to the primitives (look up tables, registers...) available in the FPGA.
2. **Placing** decides on the location of all the primitives instantiated by the previous step.
3. **Routing** builds signal paths connecting the primitives placed by the previous step.
4. **Writing bitstream** embeds the place and route information into a single binary file.

The term implementation sometimes refers only to steps 2 and 3 but I will be using it to describe the whole process.

5.1 Preparing the environment

The software toolchain used for implementation is usually given by the manufacturer of the FPGA for which the implementation is going to be done. There have been some attempts to create open source synthesis and/or implementation tools though. As I have not tried any of them, I will focus on using the official tools provided by Xilinx in this section.

5.1.1 Vivado

The software kit provided by Xilinx for synthesis and implementation is called Vivado. It provides both GUI (graphical user interface) and CLI (command line interface) for all relevant tasks.

Installing

The current version of Vivado can be downloaded from the following link: `https://www.xilinx.com/support/download.html`. It is available for Windows and Linux operating systems.

The installer provides a simple way to install all different software provided by Xilinx including Vivado and Vitis. When installing all available components, the total size exceeds 100 GB, so I recommend carefully choosing only the required components. For our project, the following components will be used:

- **Vivado Design Suite** is the main and required component composing of Vivado and Vitis HLS (which will not be used but cannot be removed from the installation).
- **Kria SOMs and Starter Kits** adds support for the KR260 starter kit we will use for implementation before we design our own motherboard.
- **Zynq UltraScale+ MPSoC** adds support for other Zynq UltraScale+ devices (which should not be required but some parts of Vivado did not work correctly without it).

In my case, the total size of the installation is around 35 GB.

Installing drivers

On Linux, it was then necessary to install the drivers for communication with the board. On Windows, it was done automatically by the installer. This difference is described only in the release notes.

Fortunately, it was as simple as running the following script in the installation directory:

```
cd data/xicom/cable_drivers/lin64/install_script/install_drivers/
sudo ./install_drivers
```

Using the project mode

The first and more common way of using Vivado is the project mode. A project contains the build configuration and also references all files used by the build. The first step in this case is creating the project.

When creating the project, is is possible to choose a board instead of a device. A board consists of not only the device (the Zynq UltraScale+ chip in our case) but also other peripherals available on the board (the Ethernet PHYs in our case). That can be used to simplify the project setup in the following ways:

- When configuring the processing system, the board provides the default configuration for using all the peripherals available on the board. Without that, it would be necessary to configure everything by hand which includes configuring all the parameters of the DDR memory and mapping all the pins of other peripherals.
- When instantiating other subsystems (the Ethernet subsystem for example), the board provides the pin mapping of the peripherals (the Ethernet transceivers in this case) available on the board. The instantiation is then as simple as choosing one of the transceivers to which the controller should be connected. Unfortunately, this works only with IP provided by Xilinx so I will not be able to use it.

The disadvantage of using the project mode is that the project consists of many temporary files which makes it incompatible with version control systems like Git. One solution would be to recreate the project on every build using the command line interface. However, using the non-project mode seemed as the correct solution.

More information on the project mode can be found in chapter 3 of the user guide [21].

Using the non-project mode

The second way of using Vivado is the non-project mode. In that case, the project is only created in memory and no files are written to disk unless otherwise requested. The build process is controlled by a script which executes commands provided by Vivado.

The advantage is that everything including the build script may be put into a version control system like Git. Thanks to that, the design may be built on any computer or any version of Vivado right after checking out the Git repository.

The disadvantage is that it is obviously more complicated to create a build script when using the non-project mode than it is to create a project when using the project mode. The usage of IP provided by Xilinx is also not very convenient in the non-project mode.

More information on the non-project mode can be found in chapter 4 of the user guide [21].

5.2 Creating the build script

This section shows how to create a build script for synthesis and implementation of hardware using Vivado in the non-project mode. Vivado uses Tcl (Tool Command Language) to control the build process from the command line. It also uses the language internally because every action in the user interface is mapped to a Tcl command. The complete list of Tcl commands is available in the reference guide [20].

5.2.1 Loading files

The first part of the build script is loading all files to be used during the build process. That includes VHDL files with description of hardware entities and XDC files with board specific constraints. Using the Xilinx IP is described in the next subsection.

In order to load a VHDL file, the following Tcl command shall be used:

```
read_vhdl src/hdl/ethernet_system.vhdl
```

In order to load a XDC file, the following Tcl command shall be used:

```
read_xdc src/xdc/kr260/pinout.xdc
```

It would be time consuming to add files one by one and update the build script every time a new file is created. Fortunately, it is possible to use the `glob` command in order to get a list of files matching a pattern. The following command may be used to load all VHDL files from a directory:

```
read_vhdl [glob src/hdl/axi/*.vhdl]
```

5.2.2 Using the Xilinx IP

The most difficult part of the non-project mode is using the Xilinx IP. In our project, the following components will be implemented using the Xilinx IP:

- The configuration of the processing system is done by instantiating the **Zynq UltraScale+ Processing System** IP and setting its parameters using the GUI wizard. This IP then generates a script which is later used to initialize the processing system by software. This IP also provides the only way to communicate between the PS and the PL via AXI or other interfaces. More information on this IP can be found in the respective product guide [24].

- The conversion from XGMII to 10GBASE-R for ten gigabit Ethernet will be done by instantiating the **10G/25G Ethernet Subsystem** IP as decribed in chapter 3.10. This IP supports multiple modes of operation including the 10GBASE-R PCS/PMA mode which does exactly what we need. More information on this IP can be found in the respective product guide [1].

The IP must be first created with proper configuration and synthesised before it can be loaded by the build script to be integrated into the design. The simplest way of getting the configuration is using the GUI. The parameters of the IP are listed at the end of the product guide but they are often not well described.

After using the GUI to get the configuration, the following Tcl command shall be used to create and configure the IP:

```
create_ip -vlnv xilinx.com:ip:xxv_ethernet:4.1 -dir ip -module_name xge_pcs
set_property -dict {
    CONFIG.BASE_R_KR {BASE-R}
    CONFIG.CORE {Ethernet PCS/PMA 64-bit}
    ...
} [get_ips xge_pcs]
```

The `dir` option creates the IP in the `ip` directory instead of the default directory. The `module_name` option sets the name of the IP instead of the generated name.

After creating all the IPs using that command, the following Tcl command shall be used to synthesise the IPs:

```
set_part xck26-sfvc784-2lv-c
synth_ip [get_ips]
```

It is necessary to set the part for which the synthesis will be done first. The part number above corresponds to the device used on the Kria KR260 and KV260 starter kits.

All the previous commands must be run only once before the first build. Having the IPs synthesised separately speeds up the build process significantly. Those commands must be also run when the configuration of the IP changes.

Finally, the following Tcl command must be added to the build script for each IP to be integrated into the design:

```
read_ip ip/ip_xge_pcs/ip_xge_pcs.xci
```

5.2.3 Building the design

The last part of the build script is running all the steps required to produce a bitstream which can be later loaded into the FPGA.

In order to run the synthesis, the following Tcl command shall be used:

```
synth_design -top ethernet_system_top
```

The `top` option specifies the top level entity where the synthesis starts.

In order to run the implementation, the following Tcl commands shall be used:

```
opt_design
place_design
route_design
```

Finally, the following Tcl command shall be used to write the final bitstream to disk:

```
write_bitstream -force ethernet_system.bit
```

The `force` option allows overwriting the bitstream created by the previous build.

5.2.4 Generating reports

An optional part of the build script is generating various reports to gain more information about the result of the implementation. Especially the timing report should be reviewed each time to see whether the design meets the timing constraints or not. The utilization report provides useful information on the resources used by the design.

In order to generate the timing report, the following Tcl command shall be used:

```
report_timing_summary -file build/timing.rpt
```

In order to generate the utilization report, the following Tcl command shall be used:

```
report_utilization -file build/utilization.rpt
```

It is possible to add the `hierarchical` option in order to obtain a utilization tree of the design.

5.2.5 Running the script

When the build script was complete, the last step was to run it every time I wanted to try the latest changes on real hardware. I used the following command which expects Vivado binaries added to the PATH environment variable:

```
vivado -mode batch -source src/tcl/build.tcl
       -log build/vivado.log -nojournal
```

The `source` option specifies the build script. The `log` option saves the log to the build directory. The `nojournal` option disables saving the journal because its content is subset of the log.

5.3 Adding physical constraints

For successful implementation, the physical constraints which assign the signals of the top level entity to the pins of the chip need to be defined. In our case, it is necessary to combine several sources in order to get the whole chain of the pin assignment. For each signal, the following steps had to be performed:

1. First, the schematic of the board had to be examined. After finding the peripheral to which the signal had to be connected (the Ethernet transceiver in our case), it was necessary to note the **net name** assigned to the signal by the designer of the schematic.
2. Second, the data sheet of the module had to be examined. On pages 13 to 28, there are two tables showing the pinout of the two connectors which are used to connect the module to the board. From there, it was necessary to note the **connector pin** for every important signal. The same information could be also obtained from the schematic.
3. Third, the XDC file of the module had to be examined. It includes the mapping from the connector pin to the package pin for every available signal in the form of XDC constraints. From there, it was necessary to note the **package pin** for every important signal. I was not able to obtain this information from any other source.

During that process, I created a table for all the signals required in our project which included the mapping all the way from the peripheral pin to the package pin.

- The pin assignment for GEM2 (the upper left Ethernet of the board) is shown in table 5.1. All pins are part of the **som240_1** connector.
- The pin assignment for GEM3 (the lower left Ethernet of the board) is shown in table 5.2. All pins are part of the **som240_2** connector.

Device pin	Net name	Connector pin	Package pin
TX_CLK	HPA06P	A3	A2
TX_CTRL	HPA00N	C4	F1
TX_D3	HPA02N	D5	E2
TX_D2	HPA02P	D4	F2
TX_D1	HPA01N	D8	D1
TX_D0	HPA01P	D7	E1
RX_CLK	HPA09P	D10	D4
RX_CTRL	HPA08N	C10	A4
RX_D3	HPA08P	C9	B4
RX_D2	HPA07N	B8	A3
RX_D1	HPA07P	B7	B3
RX_D0	HPA06N	A4	A1
RESET	HPA05N	B2	B1

Table 5.1: Pin assignment for GEM2

Device pin	Net name	Connector pin	Package pin
TX_CLK	HPB06P	A20	J1
TX_CTRL	HPB00N	D16	Y8
TX_D3	HPB02N	C18	V8
TX_D2	HPB02P	C17	U8
TX_D1	HPB01N	D13	V9
TX_D0	HPB01P	D12	U9
RX_CLK	HPB09P	C11	K4
RX_CTRL	HPB08N	A15	H3
RX_D3	HPB08P	A14	H4
RX_D2	HPB07N	B16	J2
RX_D1	HPB07P	B15	K2
RX_D0	HPB06N	A21	H1
RESET	HPB05N	B19	K1

Table 5.2: Pin assignment for GEM3

Once that information had been obtained, the last step was to craft the constraints. I decided to place the GE related physical constraints to a separate file named `ge_pinout.xdc` and the XGE related physical constraints to a separate file named `xge_pinout.xdc`. Apart from the pin assignment, the I/O standard which defines the low and high voltages should be also set for each bank.

- The LVCMOS18 standard which uses 1.8 V logic will be used for the data and management interfaces of the GE transceivers (PHY chips).
- The LVCMOS33 standard which uses 3.3 V logic will be used for the management interface of the XGE transceiver (SFP+ module).

To assign a package pin to a port of the top level entity, the following constraint shall be used:

```
set_property PACKAGE_PIN <package_pin> [get_ports <port>]
```

To set the I/O standard of a port of the top level entity, the following constraint shall be used:

```
set_property IOSTANDARD <standard> [get_ports <port>]
```

5.4 Adding timing contraints

For successful implementation, the timing constraints which specify the frequency of the clocks and also the skew between the clock signals and the data signals need to be defined. In our case, the most critical interface regarding timing is the RGMII interface used with the GE transceivers because the interface used with the XGE transceiver is constrained by the Xilinx IP.

I found out that when no timing constraints are specified, the implementation appears successful according to the log, but the design does not work at all on the real device. When timing constraints are specified, the implementation process tries to meet the constraints and it reports whether it was successful in the end.

Detailed description of the timing constraints can be found in the user guide [22]. Some useful information can be also found in chapter 4 of the UltraFast Design Methodology [15]. In our case, the following types of constraints were necessary:

- The `create_clock` and `create_generated_clock` constraints build the clock tree and set the frequency of the clocks. They are described in subsection 5.4.1. More information can be also found in chapter 3 of the user guide.
- The `set_input_delay` constraint sets the skew between clock and data for input ports. It is described in subsection 5.4.2. More information can be also found in chapter 4 of the user guide.
- The `set_output_delay` constraint sets the skew between clock and data for output ports. It is described in subsection 5.4.3. More information can be also found in chapter 4 of the user guide.
- The `set_clock_groups` constraint creates groups of clocks which are asynchronous (there is no known relationship between them) or exclusive (they never run at the same time). It is described in subsections 5.4.4 and 5.4.5. More information can be also found in chapter 5 of the user guide.

5.4.1 Creating clocks

The first step is to create all the clocks present in the design to be used by the timing analysis. Usually, it is necessary to create only the clocks which physically enter the device via one of the pins. Those are called **primary clocks**. Other clocks, such as those generated in the device by dividing another clock in the clock management tile, are usually created automatically during the implementation process. Those are called **generated clocks**. However, it is possible to give them a more meaningful name.

In order to create a primary clock, the following constraint shall be used:

```
create_clock -name <name> -period <period> [get_ports <port>]
```

The period is specified in nanoseconds. The port should be an existing port of the top level entity where the clock enters the device.

In order to rename a generated clock, the following constraint shall be used:

```
create_generated_clock -name <name> [get_pins <pin>]
```

The pin should be any pin of the design where the clock propagates. Unfortunately, it turned out that some pins do not have consistent names among different builds. The best choice was to use the output pin of the clock management tile. For example, I used the following constraint to rename the first clock generated by the clock management tile:

```
create_generated_clock -name clk_250 [get_pins clk/mmcm/CLKOUT0]
```

5.4.2 Setting input delays

Input delays specify the maximal and minimal skew between clock and data for input ports. The `set_input_delay` constraint should be added for every input port. Otherwise, no timing analysis is performed on that input port. The maximal and minimal values should be set according to the data sheet of the device to which the port is connected.

The data sheet of the Texas Instruments DP83867 Ethernet PHY [4] contains the diagram shown in figure 5.1 with the following values:

- The **setup time** at the receiver (marked as T_{setupR} in the diagram) is at least **1 ns** so the data is valid at least 1 ns **before** the respective edge of the clock.
- The **hold time** at the receiver (marked as T_{holdR} in the diagram) is at least **1 ns** so the data is valid at least 1 ns **after** the respective edge of the clock.

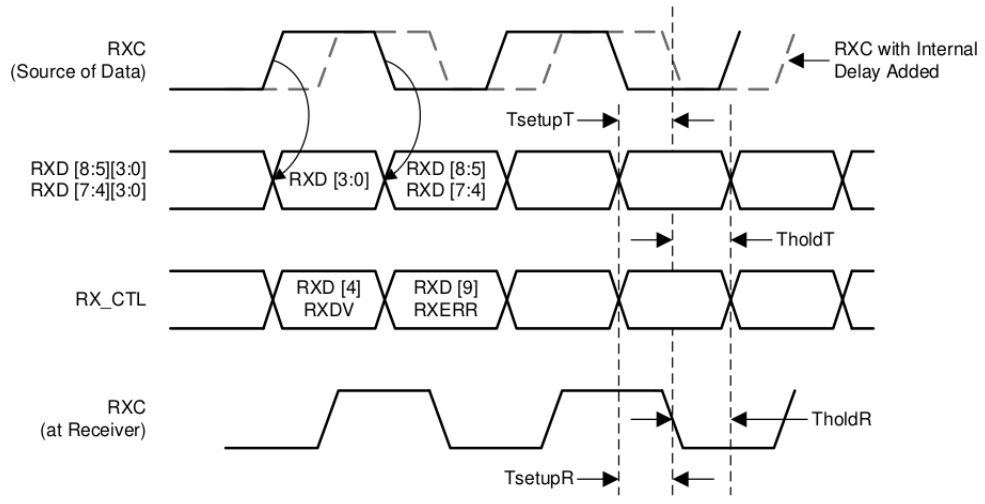

Figure 5.1: Timing diagram of RGMII reception [4]

The RX clock is shifted inside the PHY by **2 ns** as shown in the diagram and as configured by the strap resistors on the board. That means the interface is **center aligned** from the perspective of the FPGA (the clock edge is aligned to the center of the data window) while it is **edge aligned** from the perspective of the PHY (which we do not care about).

The most useful resource for creating the input delay constraints was the template located in Vivado in the following path: XDC / Timing Constraints / Input Delay Constraints / Source Synchronous / Center-Aligned / Double Data Rate (DDR). It says the maximal and minimal values should be set as follows:

- The maximal value should be equal to half of the **period** minus the **setup time**. In our case, that gives 4 ns - 1 ns = **3 ns**.
- The minimal value should be equal to the **hold time**. In our case, that is **1 ns**.

That gives the following timing constraints that have been added to the `ge_timing.xdc` file for every RGMII interface:

```
set_input_delay -clock rgmii_rx_clk -min 1
    [get_ports {rgmii_rx_data[*] rgmii_rx_control}]
set_input_delay -clock rgmii_rx_clk -max 3
    [get_ports {rgmii_rx_data[*] rgmii_rx_control}]
```

The clock specified in the constraints is the RX clock going from the PHY to the FPGA which has been created by the following constraint:

```
create_clock -name rgmii_rx_clk -period 8 [get_ports rgmii_rx_clk]
```

5.4.3 Setting output delays

Output delays specify the maximal and minimal skew between clock and data for output ports. The `set_output_delay` constraint should be added for every output port. Otherwise, no timing analysis is performed on that output port. The maximal and minimal skew should be set according to the data sheet of the device to which the port is connected.

The data sheet of the Texas Instruments DP83867 Ethernet PHY [4] contains the diagram shown in figure 5.2 with the following values:

- The **skew** at the transmitter (marked as T_{skewT} in the diagram) must be between **-0.5 ns** and **+0.5 ns** so the data can change only in a window of **1 ns** centered around the respective edge of the clock.

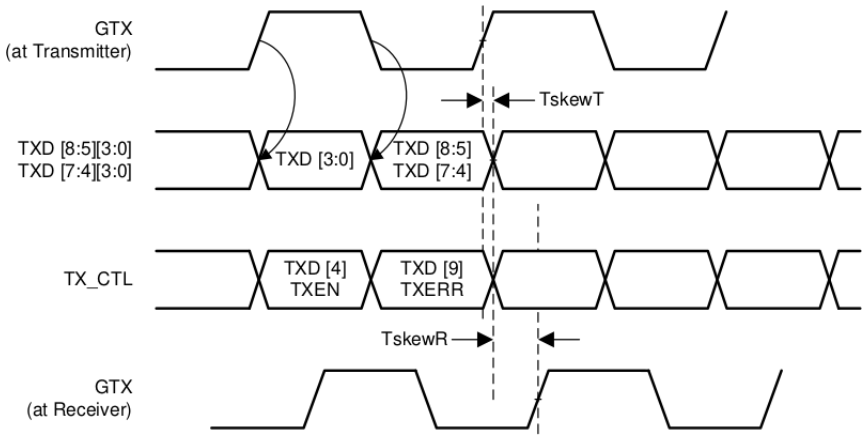

Figure 5.2: Timing diagram of RGMII transmission [4]

The TX clock is shifted inside the PHY by **2 ns** as shown in the diagram and as configured by the strap resistors on the board. That means the interface is **edge aligned** from the perspective of the FPGA (the clock edge is aligned to the data edge) while it is **center aligned** from the perspective of the PHY (which we do not care about).

The most useful resource for creating the output delay constraints was the template located in Vivado in the following path: XDC / Timing Constraints / Output Delay Constraints / Source Synchronous / Skew Based / Double Data Rate (DDR). It says the maximal and minimal values should be set as follows:

- The maximal value should be equal to half of the **period** minus the **skew**. In our case, that gives 4 ns - 0.5 ns = **3.5 ns**.
- The minimal value should be equal to the **skew**. In our case, that is **0.5 ns**.

That gives the following timing constraints that have been added to the `ge_timing.xdc` file for every RGMII interface:

```
set_output_delay -clock rgmii_tx_clk -min 0.5
    [get_ports {rgmii_tx_data[*] rgmii_tx_control}]
set_output_delay -clock rgmii_tx_clk -max 3.5
    [get_ports {rgmii_tx_data[*] rgmii_tx_control}]
```

The clock specified in the constraints is the TX clock generated by passing the RX clock through the ODDR primitive which has been created by the following constraint:

```
create_generated_clock -name rgmii_tx_clk -multiply_by 1
    -source [get_ports rgmii_rx_clk] [get_ports rgmii_tx_clk]
```

5.4.4 Defining asynchronous clocks

Whenever there are two or more **asynchronous** clocks, they must be explicitly defined by adding timing exceptions for meaningful timing analysis. There is no known frequency or phase relationship between asynchronous clocks. As explained in section 3.2, it is necessary to use an asynchronous FIFO or other synchronization primitives for crossing asynchronous clocks domains. Without the timing exceptions, the implementation process takes much longer due to tight timing constraints which are not met in the end anyway.

Asynchronous clocks are defined by dividing the clocks into several clock groups. It must hold that two clocks assigned to the same group are synchronous and two clocks assigned to different groups are asynchronous. The clocks in our design can be divided into the following clock groups with respect to that rule:

- The first clock group are the clocks which are generated by multiplying or dividing the clock coming from the 25 MHz oscillator.
- There is one clock group for every GE port which includes the RX clock recovered by the PHY and the TX clock generated by the MAC.
- There is one clock group for every XGE port which includes the RX and TX clock coming from the Xilinx IP implementing the PCS/PMA.

The following constraint has been added to the XDC file for every GE port in order to create the asynchronous clock group for that port:

```
set_clock_groups -asynchronous
    -group [get_clocks -include_generated_clocks clk_25]
    -group {rgmii_rx_clk rgmii_tx_clk}
```

The following constraint has been added to the XDC file for every XGE port in order to create the asynchronous clock group for that port:

```
set_clock_groups -asynchronous
    -group [get_clocks -include_generated_clocks clk_25]
    -group xgmii_rx_clk -group xgmii_tx_clk
```

5.4.5 Defining exclusive clocks

Whenever there are two or more **exclusive** clocks, they must be explicitly defined by adding timing exceptions for meaningful timing analysis. There is no interaction between exclusive clocks because they never run at the same time. As shown in section 3.1, there is a clock multiplexer for each pair of GE ports (currently, there is only one pair) which adjusts the frequency of the GE clock to the speed of the GE ports of the pair. Without the timing exceptions, the implementation process does not take into account that only one of the clocks actually propagates through the clock multiplexer.

The following lines have been added to the XDC file for every pair of GE ports in order to define the exclusive clocks for that pair:

```
create_generated_clock -name ge_clk_125
    -add -master_clock clk_125 -divide_by 1
    -source [get_pins core/clock/mux/I0] [get_pins core/clock/mux/O]
create_generated_clock -name ge_clk_12_5
    -add -master_clock clk_12_5 -divide_by 1
    -source [get_pins core/clock/mux/I1] [get_pins core/clock/mux/O]
set_clock_groups -physically_exclusive -group ge_clk_125 -group ge_clk_12_5
```

Chapter 6

Designing the Software

This chapter describes the process of designing all the software components needed to accomplish all the goals set by chapter 1.

6.1 Planning the structure

The only piece of software required to fulfill the assignment would be the PTP handler. However, I would like to build the software well so that it can be later extended with more functionality. Moreover, it should be later possible to switch from the bare metal environment where the programmer fully and directly controls the processor to the Linux environment where all interactions with hardware are done by the operating system. For those reason, I decided to split the software into several layers:

- The **drivers** are responsible for communication with my custom hardware peripherals described in chapter 3 and also with some standard hardware peripherals described in the technical reference manual [23]. More about the drivers can be found in section 6.2.
- The **protocols** include structures and functions for working with standard internet protocols (IPv4, IPv6, ARP, ICMP and so on). They do not interact with drivers so they could be used with any hardware. More about the protocols can be found in section 6.3.
- The **handlers** are responsible for receiving and transmitting frames using standard protocols (currently ICMP and PTP). They depend on the drivers to receive and transmit frames on one of the Ethernet interfaces. They also depend on the respective protocols. More about the handlers can be found in section 6.4 for ICMP and section 6.5 for PTP.
- The **examples** are full programs that can be built and run as shown in chapter 7. Their purpose is to provide reference for creating custom programs and also to test the collaboration of the underlying hardware and software which is done in chapter 8. They depend on the drivers to setup the hardware peripherals. They also depend on the handlers to deal with the upper layer protocols. More about the examples can be found in section 6.6.

6.2 Writing the drivers

This section describes the design of the drivers which are responsible with communication with my custom hardware peripherals.

6.2.1 Configuring the drivers

To support multiple hardware configurations, I created a configuration file called `config.h` where the drivers can be configured to match the hardware configuration by creating appropriate

structures and setting their fields according to the current requirements. Replacing that file will allow fast switching from the starter kit which provides only two GE ports and one XGE port to the custom motherboard which will hopefully provide eight GE ports and two XGE ports.

The following snippet shows the configuration file currently used for the starter kit:

```
ge_port_t ge1 = {
    .base = 0x80010000,
    .pair = &ge2,
    .dma = {
        .rx_buffer_start = (uint8_t *)0xfffd1000,
        .rx_buffer_size = 0x1000,
        .tx_buffer_start = (uint8_t *)0xfffe1000,
        .tx_buffer_size = 0x1000
    },
    .phy = {
        .smi_address = 0x02
    }
};

ge_port_t ge2 = {
    .base = 0x80020000,
    .pair = &ge1,
    .dma = {
        .rx_buffer_start = (uint8_t *)0xfffd2000,
        .rx_buffer_size = 0x1000,
        .tx_buffer_start = (uint8_t *)0xfffe2000,
        .tx_buffer_size = 0x1000
    },
    .phy = {
        .smi_address = 0x03
    }
};

xge_port_t xge = {
    .base = 0x80030000
};
```

The address map is given by combining the address map of the Zynq UltraScale+ system as specified in chapter 10 of the technical reference manual [23] and the address map of the Ethernet system as specified in section 3.15.5. The following Ethernet ports are configured:

- The first GE port (which is the upper physical port) is mapped to 0x80010000. The RX buffer of 4 kB starts at 0xFFFD1000. The TX buffer of 4 kB starts at 0xFFFE1000. The SMI address of the PHY is fixed to 0x02 by the schematic of the board.
- The second GE port (which is the lower physical port) is mapped to 0x80020000. The RX buffer of 4 kB starts at 0xFFFD2000. The TX buffer of 4 kB starts at 0xFFFE2000. The SMI address of the PHY is fixed to 0x03 by the schematic of the board.
- The XGE port (implemented by the SFP+ module) is mapped to 0x80030000.

All buffers are currently located in the OCM (on chip memory) because it does not require any initialization so it is simpler to use than other types of memory.

6.2.2 Setting up the components

For each component, there is a setup function which initializes that component with default settings. To simplify the initialization that must be done at the beginning of every program, there is also a setup function which initializes all the components belonging to one of the GE or XGE ports.

To set up one of the GE ports, the `ge_port_setup` function must be called. It needs a pointer to the `ge_port_t` structure which is usually created in the configuration file as described above. It calls the following setup functions:

- The `ge_splitter_setup` function for the RX splitter and the FB splitter designed in section 3.7.4 which enables the splitter and sets the default action to drop all frames.
- The `ge_cmp_setup` function for the CMP block designed in section 3.13.7 which enables both the RX CMP packager and the TX CMP packager.
- The `ge_dma_setup` function for the DMA block designed in section 3.8.6 which enables both the DMA receiver and the DMA transmitter and sets the buffer start and size registers to the values defined in the configuration file.
- The `ge_mac_setup` function for the MAC block designed in section 3.6.5 which enables both the MAC receiver and the MAC transmitter.
- The `ge_phy_setup` function for the PHY block which enables the SMI controller designed in section 3.14.3 and sets the appropriate MDC divisor.

For more information about the structure of the GE port, see section 3.15.3.

To set up one of the XGE ports, the `xge_port_setup` function must be called. It needs a pointer to the `xge_port_t` structure which is usually created in the configuration file as described above. It calls the following setup functions:

- The `xge_cmp_setup` function for the CMP block which enables the CMP aggregator designed in section 3.13.4 and sets the default values of the maximal length and delay registers.
- The `xge_sfp_setup` function for the SFP block which enables the I2C controller designed in section 3.14.4 amd sets the appropriate SCL divisor.

For more information about the structure of the XGE port, see section 3.15.4.

6.2.3 Receiving and transmitting frames

This subsection shows how the DMA block designed in section 3.8.6 can be used to receive and transmit frames. Before calling the functions described below, it is necessary to call the `ge_dma_setup` function which enables the DMA block. It is usually done as part of the initialization of the GE port.

Receiving frames

To receive a frame from one of the Ethernet ports, the `ge_port_rx_read` function must be called. It needs a pointer to the `ge_port_t` structure to specify the port from which the frame shall be received and also a pointer to the `ge_frame_t` structure where the metadata of the frame will be put by the called function. It returns 0 in case of success and -1 when there are no new frames available.

Transmitting frames

To transmit a frame to one of the Ethernet ports, the `ge_port_tx_write` function must be called. It needs a pointer to the `ge_port_t` structure to specify the port to which the frame shall be transmitted and also a pointer to the `ge_frame_t` structure where the metadata of the frame has been put by the calling function. It returns 0 in case of success and -1 when there is no space available.

Before creating the frame to be transmitted, the `ge_port_tx_create` function must be called to allocate space for the frame in the TX buffer. It needs pointers to the `ge_port_t` and `ge_frame_t` structures as above and also the length of the frame to allocate enough space. It fills in the fields of the `ge_frame_t` structure and also sets all bytes of the frame to zeros. It returns 0 in case of success and -1 when there is no space available.

Getting TX timestamps

After transmitting the frame, the `ge_port_fb_read` function may be called to get the TX timestamp. It needs a pointer to the `ge_port_t` structure to specify the port to which the frame has been transmitted and also a pointer to the `ge_frame_t` structure where the metadata of the frame will be put by the called function. It returns 0 in case of success and -1 when there are no new frames available.

6.2.4 Setting up the splitters

This subsection shows how the splitter designed in section 3.7.4 can be managed from software. Before calling the functions described below, it is necessary to call the `ge_splitter_setup` function which enables the splitter. It is usually done as part of the initialization of the GE port.

Setting the default action

To set the default action of the splitter, the `ge_splitter_set_default` function can be used. It needs a pointer to the `ge_splitter_t` structure which is member of the `ge_port_t` structure to specify the splitter to configure. The Nth bit of the action parameter deteremines whether the frame is passed to the Nth output of the splitter when all matchers report a mismatch. The output assignment is described in section 3.15.3. A convenient macro is defined for every output in the `ge/port.h` file.

For example, to set the default action of the RX splitter of the GE1 port to pass the frame to the CMP packager and also to the other GE port of the pair, the following line should be used:

```
ge_splitter_set_default(&ge1.rx, GE_RX_CMP | GE_RX_PASS);
```

Enabling the matchers

To enable one of the matchers which are part of the splitter, the `ge_splitter_enable_matcher` function can be used. It needs a pointer to the `ge_splitter_t` structure which is member of the `ge_port_t` structure to specify the splitter to configure. The index parameter selects the matcher to enable. The first matcher has index equal to zero and the last matcher has index equal to the count of the matchers minus one. The Nth bit of the action parameter determines whether the frame is passed to the Nth output of the splitter when that matcher reports a match.

For example, to enable the 1st matcher of the RX splitter of the GE1 port and set the associated action to pass the frame to the DMA receiver, the following line should be used:

```
ge_splitter_enable_matcher(&ge1.rx, 0, GE_RX_DMA);
```

Disabling the matchers

To disable one of the matchers which are part of the splitter, the `ge_splitter_disable_matcher` function can be used. It needs a pointer to the `ge_splitter_t` structure which is member of the `ge_port_t` structure to specify the splitter to configure. The index parameter selects the matcher to disable. The first matcher has index equal to zero and the last matcher has index equal to the count of the matchers minus one.

For example, to disable the 1st matcher of the RX splitter of the GE1 port, the following line should be used:

```
ge_splitter_disable_matcher(&ge1.rx, 0);
```

Configuring the matchers

To set the length of the matcher, the `ge_matcher_set_length` function shall be used. It needs a pointer to the `ge_matcher_t` structure which can be obtained from the `matchers` array of the `ge_splitter_t` structure. The length parameter sets the length of the matcher in bytes. The matcher then ignores the content of the value and mask arrays beyond that length.

To modify the value array and the mask array of the matcher, the `ge_matcher_value` function and the `ge_matcher_mask` function shall be used to obtain a pointer to the respective array. They need a pointer to the `ge_matcher_t` structure which can be found in the `matchers` array of the `ge_splitter_t` structure. They return a pointer to the first byte of the respective array which can be conveniently used with the structures and functions defined for the internet protocols as described in section 6.3.

For example, to configure the 1st matcher of the RX splitter of the GE1 port to match when the destination MAC address equals to AA:BB:CC:DD:EE:FF, the following code shall be used:

```
ge_matcher_t *matcher = &ge1.rx.matchers[0];
ge_matcher_set_length(matcher, sizeof(mac_header_t));

byte_t dst_addr[6] = {0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF};
mac_header_t *mac_value = mac_get_header(ge_matcher_value(matcher));
byte_copy(mac_value->dst_addr, dst_addr, 6);

mac_header_t *mac_mask = mac_get_header(ge_matcher_mask(matcher));
byte_set(mac_mask->dst_addr, 0xFF, 6);
```

As another example, to configure the 1st matcher of the FB splitter of the GE1 port to match when the EtherType equals to 0x88F7 (PTP), the following code shall be used:

```
ge_matcher_t *matcher = &ge1.fb.matchers[0];
ge_matcher_set_length(matcher, sizeof(mac_header_t));

mac_header_t *mac_value = mac_get_header(ge_matcher_value(matcher));
mac_value->eth_type = byte_rev(0x88F7);

mac_header_t *mac_mask = mac_get_header(ge_matcher_mask(matcher));
mac_mask->eth_type = 0xFFFF;
```

6.2.5 Setting up the packagers

This subsection shows how the CMP packagers designed in section 3.13.3 and the CMP aggregator designed in section 3.13.4 can be managed from software. Before calling the functions

described below, it is necessary to call the `ge_cmp_setup` function which enables the CMP packagers and the `xge_cmp_setup` function which enables the CMP aggregator. It is usually done as part of the initialization of the GE port and the XGE port respectively.

To set the interface ID of the CMP packagers, the `ge_cmp_set_interface_id` function can be used. It needs a pointer to the `ge_cmp_t` structure which is member of the `ge_port_t` structure to choose the GE port. The interface ID must be a 32 bit integer according to the specification of the CMP protocol.

To set the device ID of the CMP aggregator, the `xge_cmp_set_device_id` function can be used. It needs a pointer to the `xge_cmp_t` structure which is member of the `xge_port_t` structure to choose the XGE port. The device ID must be a 16 bit integer according to the specification of the CMP protocol.

To set the stream ID of the CMP aggregator, the `xge_cmp_set_stream_id` function can be used. It needs a pointer to the `xge_cmp_t` structure which is member of the `xge_port_t` structure to choose the XGE port. The stream ID must be an 8 bit integer according to the specification of the CMP protocol.

6.2.6 Managing the transceivers

This subsection shows how the SMI controller designed in section 3.14.3 can be managed from software. Before calling the functions described below, it is necessary to call the `ge_phy_setup` function which enables the SMI controller. It is usually done as part of the initialization of the GE port.

To write a value to one of the PHY management registers, the `ge_phy_reg_write` function can be used. It needs a pointer to the `ge_phy_t` structure which is member of the `ge_port_t` structure to specify the port to which the PHY belongs. The address parameter sets the address of the management register which is specified either by the IEEE standard or the manufacturer. The value parameter sets the value to be written to that register. The function is capable of writing to the extended register set by using the REGCR and ADDAR registers.

To read a value from one of the PHY management registers, the `ge_phy_reg_read` function can be used. It needs a pointer to the `ge_phy_t` structure which is member of the `ge_port_t` structure to specify the port to which the PHY belongs. The address parameter sets the address of the management register which is specified either by the IEEE standard or the manufacturer. The function returns the value that was read from that register. The function is capable of reading from the extended register set by using the REGCR and ADDAR registers.

6.3 Defining the protocols

Working with internet protocols in C can be simplified by creating structures which could be mapped onto the headers in order to get or set the value of the individual fields of the headers. So far, I have created the following structures and functions:

- The `mac_header` structure can be used to work with Ethernet frames in general.
  - The `mac_get_header` and `mac_get_payload` functions return pointer to the MAC header or the MAC payload respectively given a pointer to an Ethernet frame.
  - The `mac_header_print` and `mac_address_print` functions print a MAC header or a MAC address respectively in human readable form.
- The `ipv4_header` structure can be used to work with IPv4 (Internet Protocol version 4) frames.

  - The `ipv4_get_header` and `ipv4_get_payload` functions return pointer to the IPv4 header or the IPv4 payload respectively given a pointer to an Ethernet frame. However, they do not check whether the frame actually uses the IPv4 protocol.
  - The `ipv4_header_print` and `ipv4_address_print` functions print an IPv4 header or an IPv4 address respectively in human readable form.

- The `ipv6_header` structure can be used to work with IPv6 (Internet Protocol version 6) frames.
  - The `ipv6_get_header` and `ipv6_get_payload` functions return pointer to the IPv6 header or the IPv6 payload respectively given a pointer to an Ethernet frame. However, they do not check whether the frame actually uses the IPv6 protocol.
  - The `ipv6_header_print` and `ipv6_address_print` functions print an IPv6 header or an IPv6 address respectively in human readable form.

- The `arp_payload` structure can be used to work with ARP (Address Resolution Protocol) frames.

- The `icmp_header` structure can be used to work with ICMP (Internet Control Message Protocol) frames.
  - The `icmp_echo_message` structure can be mapped onto Echo Reply and Echo Request messages (used in both ICMPv4 and ICMPv6).
  - The `icmp_neighbor_message` structure can be mapped onto Neighbor Solicitation and Neighbor Advertisement messages (used only in ICMPv6).

- The `ptp_header` structure can be used to work with PTP (Precision Time Protocol) frames.
  - The `ptp_delay_message` structure can be mapped onto Delay Request, Delay Response and Delay Response Follow Up messages.
  - The `ptp_sync_message` structure can be mapped onto Sync and Follow Up messages.

6.4 Writing the ICMP handler

This section describes the design of the ICMP handler which allows responding to echo messages used by the `ping` utility both on IPv4 and IPv6. It implements the minimal subset of the ICMP specification in order to achieve that goal.

The ICMP handler in our design consists of the following components:

- The handler of ARP requests. These allow other devices on the same network to translate the IPv4 address of our device to the MAC address. The handler is described in subsection 6.4.1.

- The handler of ICMPv6 neighbor solicitations. These allow other devices on the same network to translate the IPv6 address of our device to the MAC address. The handler is described in subsection 6.4.2.

- The handler of ICMPv4 echo requests. These allow other devices on the same network to check the presence of our device. The handler is described in subsection 6.4.3.

- The handler of ICMPv6 echo requests. These allow other devices on the same network to check the presence of our device. The handler is described in subsection 6.4.4.

6.4.1 ARP request handler

It is implemented in the `arp_request_handler` function according to RFC 826. It receives ARP Request messages and transmits ARP Reply messages with the following fields:

- The source MAC address is given by the global constant `my_mac_addr`.
- The destination MAC address is taken from the source MAC address of the request.
- The source IP address is given by the global constant `my_ipv4_addr`.
- The destination IP address is taken from the source IP address of the request.

6.4.2 ICMPv6 neighbor solicitation handler

It is implemented in the `icmpv6_neighbor_solicitation_handler` function according to RFC 4861. It receives ICMPv6 Neighbor Solicitation messages and transmits ICMPv6 Neighbor Advertisement messages with the following fields:

- The source MAC address is given by the global constant `my_mac_addr`.
- The destination MAC address is taken from the source MAC address of the solicitation.
- The source IP address is given by the global constant `my_ipv6_addr`.
- The destination IP address is taken from the source IP address of the solicitation.
- The ICMPv6 payload includes the same source MAC and IP addresses.
- The ICMPv6 checksum is computed by the `ipv6_checksum` function.

6.4.3 ICMPv4 echo request handler

It is implemented in the `icmpv4_echo_request_handler` function according to RFC 792. It receives ICMPv4 Echo Request messages and transmits ICMPv4 Echo Reply messages with the following fields:

- The source addresses are taken from the destination addresses of the request.
- The destination addresses are taken from the source addresses of the request.
- The identifier, the sequence number and the payload are copied from the request.
- The IPv4 and ICMPv4 checksums are computed by the `word16_sum` function.

6.4.4 ICMPv6 echo request handler

It is implemented in the `icmpv6_echo_request_handler` function according to RFC 4443. It receives ICMPv6 Echo Request messages and transmits ICMPv6 Echo Reply messages. It works similar to the previous handler.

6.5 Writing the PTP handler

This section describes the design of the PTP handler which allows propagating time synchronization from the **slave port** to the **master port**. It implements the minimal subset of the PTP specification in order to achieve that goal.

The PTP handler is designed in a way that it may be instantiated multiple times on different pairs of ports. It is represented by the `ptp_handler` structure which internally contains more structures, one for each implemented entity.

The PTP handler in our design consists of the following components:

- The **Delay Request** state machine which periodically sends Delay Request messages on the **slave port** and then receives Delay Response messages and their Follow Up messages. Finally, it computes the mean link delay and the neighbor rate ratio. It is described in subsection 6.5.1.
- The **Delay Response** state machine which receives Delay Request messages on the **master port** and then responds with Delay Response messages and their Follow Up messages. It is described in subsection 6.5.2.
- The **Sync Receive** state machine which receives Sync messages and their Follow Up messages on the **slave port** and then passes the sync information to the Sync Send state machine. It is described in subsection 6.5.3.
- The **Sync Send** state machine which sends Sync messages and their Follow Up messages on the **master port** once it receives the sync information from the Sync Receive state machine. It is described in subsection 6.5.4.

6.5.1 Delay Request state machine

The Delay Request state machine runs the following loop:

1. It waits until the Delay Request interval timer expires.
2. It increments the current sequence ID.
3. It transmits the Delay Request message on the slave port.
   - The sequence ID is set to the current sequence ID.
4. It waits until the Delay Request message is transmitted.
   - The `req_egress` variable is set to the local timestamp of when the message was transmitted.
5. It waits until the Delay Response message with the current sequence ID is received on the slave port.
   - The `req_ingress` variable is taken from the message.
   - The `resp_ingress` variable is set to the local timestamp of when the message was received.
6. It waits until the Delay Response Follow Up message with the current sequence ID is received on the slave port.
   - The `resp_egress` variable is taken from the message.
7. It computes the mean link delay and the neighbor rate ratio using the equations below.

$$\text{mean\_link\_delay} = \frac{(\text{resp\_ingress} - \text{req\_egress}) - (\text{resp\_egress} - \text{req\_ingress})}{2} \tag{6.1}$$

$$\text{neighbor\_rate\_ratio} = \frac{\text{resp\_egress} - \text{last\_resp\_egress}}{\text{resp\_ingress} - \text{last\_resp\_ingress}} \tag{6.2}$$

6.5.2 Delay Response state machine

The Delay Response state machine runs the following loop:

1. It waits until the Delay Request message is received on the master port.
   - The current sequence ID is taken from the message.
   - The `req_ingress` variable is set to the local timestamp of when the message was received.
2. It transmits the Delay Response message on the master port.
   - The sequence ID is set to the current sequence ID.
   - The timestamp is taken from the `req_ingress` variable.
3. It waits until the Delay Response message is transmitted.
   - The `resp_egress` variable is set to the local timestamp of when the message was transmitted.
4. It transmits the Delay Response Follow Up message on the master port.
   - The sequence ID is set to the current sequence ID.
   - The timestamp is taken from the `resp_egress` variable.

6.5.3 Sync Receive state machine

The Sync Receive state machine runs the following loop:

1. It waits until the Sync message is received on the slave port.
   - The current sequence ID is taken from the message.
   - The `ingress_timestamp` variable is set to the local timestamp of when the message was received.
2. It waits until the Follow Up message with the current sequence ID is received on the slave port.
   - The `origin_timestamp` variable is taken from the message.
   - The `upstream_tx_time` variable is calculated using the equation below.
3. It sends the sync information to the Sync Send state machine.

$$\text{upstream\_tx\_time} = \text{ingress\_timestamp} - \frac{\text{mean\_link\_delay}}{\text{neighbor\_rate\_ratio}} \tag{6.3}$$

6.5.4 Sync Send state machine

The Sync Send state machine runs the following loop:

1. It waits until the sync information is received from the Sync Receive state machine.
2. It increments the current sequence ID.
3. It transmits the Sync message on the master port.
   - The sequence ID is set to the current sequence ID.
4. It waits until the Sync message is transmitted.
   - The `egress_timestamp` variable is set to the local timestamp of when the message was transmitted.
5. It transmits the Follow Up message on the master port.
   - The sequence ID is set to the current sequence ID.
   - The origin timestamp is taken from the sync information.
   - The correction field is calculated using the equation below.

$$\text{correction\_field} = \text{rate\_ratio} \times (\text{egress\_timestamp} - \text{upstream\_tx\_time}) \tag{6.4}$$

6.6 Writing the examples

This section describes the design of the examples which are full programs that can be built and run as shown in chapter 7.

6.6.1 The passthrough example

The passthrough example was created to check the Ethernet data path of the Ethernet system. It consists of several different tests which are run one by one. Switching between tests is done by pressing Enter in the console which sends CR via the UART.

Before every test, the RX splitters are configured in a specific way depending on the test. After every test, the RX and TX counters are printed to the console and reset to zero. The passthrough example runs the following tests in an infinite loop:

- Log from GE1 and GE2
- Pass between GE1 and GE2
- Pass only from GE1 to GE2
- Pass only from GE2 to GE1
- Divert from GE1 to XGE
- Divert from GE2 to XGE
- Log ARP frames and pass all frames
- Divert ICMP frames and pass other frames
- Divert HTTP frames and pass other frames

6.6.2 The loopback example

The loopback example was created to test the reliabilility and also the performance of the Ethernet data path of the Ethernet system. It uses the DMA block (and implicitly the MAC block and the PHY block) to transmit frames on one Ethernet port and to receive frames on the other Ethernet port. It compares the frames transmitted and received and counts valid (matching) and invalid (mismatching) frames.

The loopback example consists of the following processes:

- The **TX** process activates every 100 microseconds. It builds a frame of 100 bytes and transmits the frame on the TX port. It has an 8-bit sequence counter which is incremented for every transmitted frame in order to detect lost frames. It also has a 32-bit total TX counter which counts the transmitted frames.
- The **RX** process activates whenever a frame is received on the RX port. It checks the frame and prints the frame in case it is invalid. It has an 8-bit sequence counter which is incremented for every received frame in order to detect lost frames. It also has a 32-bit valid and invalid RX counter which counts the received frames.
- The **STAT** (statistics) process activates every 1 second. It prints all the counters and so provides the results of the test.

The frames consist of a valid MAC header and a sequence of bytes starting with the current value of the sequence counter. All bytes of the destination address are set to 0xFF which corresponds to broadcast. All bytes of the source address are set to 0xAE.

All the processes also measure the time it took for the process to complete using an independent timer/counter unit.

6.6.3 The ping example

The ping example was created to test interaction between our device and other devices connected to the same network. It uses the ICMP handler designed in section 6.4 to handle ARP and ICMPv4 messages on IPv4 networks and ICMPv6 messages on IPv6 networks.

6.6.4 The sync example

The sync example was created to test time synchronization between two PTP capable devices connected through our device. It uses the PTP handler designed in section 6.5 to handle PTP messages for delay measurement and time synchronization.

It uses two GE ports and one XGE port. The first GE port is called **slave port** and it should be connected to the master device. The second GE port is called **master port** and it should be connected to the slave device. Both GE ports are set up to catch received and transmitted PTP frames and to pass other frames. At the same time, the XGE port is set up to log received and transmitted PTP frames using the CMP protocol.

Chapter 7

Running the Software

After creating all software components, the next step was to load them into the processing system. This chapter is devoted to building and running the software. It is divided into the following sections:

- In section 7.1, I show how to set up the software environment for the build and run process.
- In section 7.2, I describe the first programs for tuning the build and run process.
- In section 7.3, I show how to build software for the ARM Cortex-R5 processor.
- In section 7.4, I show how to run software on the ARM Cortex-R5 processor.

7.1 Preparing the environment

The simplest option to develop software for the Zynq UltraScale+ platform is to use the IDE (integrated development environment) which is provided by Xilinx (now AMD) for this purpose. However, I found using that IDE very inconvenient. First, it provided almost no programming aids like automatic completion or instant error checking. Second, it took forever to build new version of the software and run it on the hardware so iterative development was very slow. So I spent a lot of time figuring out how to build and run software for the given platform outside of the toolchain provided by Xilinx (now AMD).

7.1.1 Vitis

The software kit provided by Xilinx for software development is called Vitis. Just like Vivado, it provides both GUI (graphical user interface) built on Eclipse and CLI (command line interface) for all related tasks. For reasons mentioned above, I decided not to use Vitis and I replaced it with open source tools described below.

Installing

The current version of Vitis can be downloaded from the following link: `https://www.xilinx.com/support/download.html`. It is available for Windows and Linux operating systems. There is actually one installer for Vivado and Vitis so both tools can be installed together.

7.1.2 VS Code

The first part of my custom toolchain is the editor. I have decided to use Visual Studio Code for this project. It is an open source editor with many useful programming and debugging features developed by Microsoft.

Installing the editor

On Linux, it can be either installed from the standard package repository or downloaded from the official web site. I downloaded the newest version from the web site and installed the package on my Debian based system using the following command:

```
dpkg -i code_xxx_amd64.deb
```

On Windows, it can be downloaded from the official web site.

Installing the extensions

The power of this editor is the huge amount of extensions available from the community. For this project, I installed the following extensions:

- The official **C/C++** extension (`ms-vscode.cpptools`) which provides syntax highlighting, automatic completion, instant error checking and other useful features for C and C++ programming languages.
- The official **Makefile Tools** extension (`ms-vscode.makefile-tools`) which provides syntax highlighting and other features for Makefiles.
- The unofficial **LinkerScript** extension (`ZixuanWang.linkerscript`) which provides syntax highlighting for GNU linker scripts.
- The unofficial **Arm Assembly** extension (`dan-c-underwood.arm`) which provides syntax highlighting for Arm assembly instructions.

Configuring the extensions

The official C/C++ extension needs a configuration file which specifies the compiler path and arguments. It is used by the extension to load the header files and other information from the compiler in order to provide more accurate completion and error checking. The file must be called `c_cpp_properties.json` and stored in the `.vscode` directory. In my case, I have put the following configuration there:

```
{
    "name": "ARM (cortex-r5)",
    "compilerPath": "/usr/bin/arm-none-eabi-gcc",
    "compilerArgs": [
        "-mcpu=cortex-r5",
        "-nostdlib"
    ],
    "includePath": [
        "sw/src"
    ]
}
```

7.1.3 Arm compiler

The second part of my custom toolchain is the compiler (or generally build tools which also include the linker and the debugger). I decided to use GCC (GNU Compiler Collection) which I had already been familiar with from other projects. It is a set of open source compilers for many modern computer architectures including **Arm** which is our point of interest. Specifically, we are interested in the `arm-none-eabi` version as we are going to build bare metal code for 32-bit Arm architecture.

Installing

On Linux, it can be either installed from the standard package repository or downloaded from the official web site. I used the following command which can be used on Debian or Ubuntu based systems:

```
apt install gcc-arm-none-eabi
```

On Windows, it can be downloaded from the official web site. The ZIP archive includes compiled binaries which can be used immediately after extracting the archive to any directory. I recommend adding the directory to PATH so that it can be always used from the command line.

7.1.4 Xilinx debugger

The third part of my custom toolchain is the debugger. I used the official debugger provided by Xilinx because I found no other options for this platform. It must be used to load both hardware bitstreams and software binaries into the chip. It may be also used to debug the program either directly or indirectly via GDB (GNU Debugger) when used as GDB server.

Installing

The Xilinx debugger is part of the software kit provided by Xilinx and it cannot be installed separately. It is part of both Vivado and Vitis installations. The most lightweight option is to install Vivado Lab Edition. It does not include the tools for synthesis and implementation but it does include the tools for programming and debugging.

7.2 The first programs

Before trying to build and run the software designed in chapter 6, I created a few simple programs to tune the build and run process. Since I decided not to use the tools provided by Xilinx including the BSP (board support package), I had to write my own drivers and functions even for the most basic things like printing using the UART and sleeping using the TTC.

7.2.1 Using the UART

For communication with outer world, one of the UART (Universal Asynchronous Receiver Transmitter) peripherals available on the chip will be used. Our board provides access to this UART via the same FTDI chip which is used for programming the FPGA and can be also used for debugging the CPU. Using this UART is therefore very conventient as it is available automatically and immediately after connecting the board to the computer via USB cable.

After proper initialization of the hardware, using the UART peripheral should be as simple as writing a byte to a register for transmission and reading a byte from a register for reception. For this purpose, I created a simple driver which provides the following functions:

- The `uart_setup` function enables the UART controller.
- The `uart_read` function blocks until there is data available for reading in the RX FIFO and then reads one byte from the RX FIFO.
- The `uart_write` function blocks until there is space available for writing in the TX FIFO and then writes one byte to the TX FIFO.

The addresses of the appropriate registers were taken from chapter 21 of the technical reference manual [23] where more information on the UART peripheral can be also found.

7.2.2 Testing the UART

For testing the UART driver and tuning the build and run process, I created a classic program which uses the UART transmitter to print "Hello world":

```
void main() {
    uart_setup();
    print("Hello world!\n");
}
```

Since there were no standard libraries, there was no function for printing strings either. So I used my own function which writes the given string character by character to the UART.

```
void print(char *s) {
    while (*s) {
        uart_write(*s);
        s++;
    }
}
```

7.2.3 Using the TTC

For accurate time measurement, one of the TTC (Triple Timer Counter) peripherals available on the chip will be used.

After proper initialization of the hardware, using the TTC peripheral should be as simple as writing to a register to start or stop the timer and reading from a register to get the current value. For that purpose, I created a simple driver which provides the following functions:

- The `ttc_setup` function enables the first TTC counter and disables the TTC prescaler.
- The `ttc_get_value` function reads the value of the first TTC counter.
- The `ttc_get_period` function returns the period of the first TTC counter. It is fixed to 10 nanoseconds at the moment.

The addresses of the appropriate registers were taken from chapter 14 of the technical reference manual [23] where more information on the TTC peripheral can be also found.

7.2.4 Testing the TTC

For testing the TTC driver and tuning the build and run process, I created a simple program which prints the value of the TTC counter every second:

```
void main() {
    ttc_setup();
    uart_setup();
    while (1) {
        uint64_t time = (uint64_t)ttc_get_value(0) * ttc_get_period();
        print_uint64(time, 10);
        sleep_ms(1000);
    }
}
```

Since there were no standard libraries, there was no function for printing integers either. So I used my own function which divides the given integer by ten repeatedly in order to get the decimal representation which is then written character by character to the UART.

7.3 Building the programs

This section describes the steps needed to build a program for the ARM Cortex-R5 processor on the Zynq UltraScale+ platform with the ARM GNU toolchain.

7.3.1 Compiling the program

For compiling the program, I used the following command:

```
arm-none-eabi-gcc
    -mcpu=cortex-r5 -nostdlib
    -g -I src -T src/zynq.ld
    <input_files>
    -l gcc -o <output_file>
```

It runs the GCC compiler and linker which is part of the ARM GNU toolchain. It contains the following flags:

- The `-mcpu=cortex-r5` flag sets the target processor to ARM Cortex-R5 so that the compiler uses only instructions available on that processor.
- The `-nostdlib` flag removes the standard library from the build. I decided not to use it in the beginning because it was easier to write my own basic functions than to implement all system calls required by the standard library.
- The `-g` flag embeds debugging information into the final ELF file. Thanks to that, the debugger is able to provide many useful features like setting breakpoints using their line numbers or printing variables using their names.
- The `-I` flag sets the include path that should be searched for header files in addition to the standard include path. I wanted to reference my header files relative to the `src` directory so I added it to the include path.
- The `-T` flag sets the linker script that should be used for linking object files together. I created my own linker script as described in the following section.

The command was too long to type every time so I created a simple Makefile to manage the build process:

```
CC=arm-none-eabi-gcc
FLAGS=-mcpu=cortex-r5 -nostdlib -g -I src -T src/zynq.ld

DRIVERS=$(wildcard src/drivers/*.c)
FUNCTIONS=$(wildcard src/functions/*.c)
INTERNALS=$(wildcard src/internals/*.s)

SOURCES=$(DRIVERS) $(FUNCTIONS) $(INTERNALS)

bin/tests/%.elf: src/tests/%.c $(SOURCES)
    $(CC) $(FLAGS) $^ -l gcc -o $@
```

Thanks to that, I could simply write `make bin/tests/uart.elf` to build the UART test and `make bin/tests/ttc.elf` to build the TTC test.

7.3.2 Writing the linker script

The purpose of the linker script is to specify the location in memory where the program and the data should be located. The linker script is usually part of the BSP and in our case, it can be created by Vitis.

Choosing the memory

In our system, there are basically three types of memory accessible from the processing system:

- The **TCM** (tightly coupled memory) is a 64 kB RAM placed right next to each of the two ARM Cortex-R5 cores. It is the closest memory from the processor perspective and thus provides the lowest latency but the smallest size. It is described in chapter 4 of the technical reference manual [23].
- The **OCM** (on chip memory) is a 256 kB internal RAM added to the system by Xilinx. It is shared among all processors in the system. It does not require any initialization because it is initialized by the boot ROM. It is described in chapter 18 of the technical reference manual [23].
- The **DDR** (double data rate) memory is a 4 GB external RAM available on the Kria module. It is shared among all processors in the system. It is the farthest memory from the processor perspective and thus provides the highest latency but the largest size. It is driven by a dedicated DDR controller which requires a complicated initialization procedure. The controller is described in chapter 17 of the technical reference manual [23].

Our programs will be very small in size so they should fit even into the smallest TCM memory. The advantage of using the TCM memory is that the latency is very low and the cache is completely bypassed. However, it turned out that the TCM memory is disabled by default. I was not able to upload my program there without setting some registers first.

For that reason, I decided to use the OCM memory for first experiments. The disadvantage of using the OCM memory is that the latency is higher and the cache should be enabled to overcome that problem. Unfortunately, enabling the cache brings other problems when the data in memory is modified externally (by the DMA block in our case).

Using the OCM

A basic linker script consists of three parts: specifying the entry point, describing the memories and mapping program sections to the memories.

The entry point is specified by the following statement:

```
ENTRY(_start)
```

The OCM memory is described by the following statement:

```
MEMORY {
    OCM (rwx) : ORIGIN = 0xFFFC0000, LENGTH = 256K
}
```

The program sections are mapped by the following statement:

```
SECTIONS {
    .text : {
        *(.text)
    } > OCM
```

```
    .data : {
        *(.data)
    } > OCM
    .rodata : {
        *(.rodata)
    } > OCM
    .bss : {
        *(.bss)
    } > OCM
}
```

7.3.3 Writing the startup file

The purpose of the startup file is to initialize the processor before jumping to the main function. It also provides the interrupt and exception vectors. The startup file is usually part of the BSP and in our case, its name is `boot.s`.

Initializing the stack

The only thing that must be done before simple programs writen in C can be executed is initializing the stack pointer. Before I added this to the startup file, the program sometimes worked and sometimes not.

The following snippet shows the minimal startup file which initializes the stack pointer and jumps to the main function:

```
_start:
    ldr sp, =0xffffff00
    bl main
_exit:
    b _exit
```

7.4 Running the programs

This section describes the steps needed to run a program on the ARM Cortex-R5 processor on the Zynq UltraScale+ platform with the Xilinx debugger.

7.4.1 Connecting the hardware

The Zynq UltraScale+ chip available on the AMD Kria board we are using provides the JTAG interface for programming and debugging purposes. The JTAG interface is connected to the FTDI chip on the board which makes it available via the microUSB port on the board. It is therefore very easy to connect the board to any computer using a standard USB to microUSB cable.

Apart from Vivado and Vitis which are also able to communicate with the chip, Xilinx provides a command line tool called **XSDB** (Xilinx System Debugger). If the installation directory has been added to the PATH environment variable, it can be started by simply running the `xsdb` command.

To connect to the local hardware server which allows communication with chips that are connected to the local computer, the following command must be run first:

```
connect
```

To connect to a remote hardware server which allows communication with chips that are connected to a remote computer, the following command must be run first:

```
connect -host <host> -port <port>
```

To verify that the communication with the chip is working correctly, the following command can be executed to list the targets available for programming and debugging:

```
targets
```

With our board, it should produce the following output:

```
1  PS TAP
   2  PMU
   3  PL
5  PSU
   6  RPU
      7  Cortex-R5 #0
      8  Cortex-R5 #1
   9  APU
     10  Cortex-A53 #0
     11  Cortex-A53 #1
     12  Cortex-A53 #2
     13  Cortex-A53 #3
```

7.4.2 Initializing the hardware

Once the build of the first program had completed successfully, it was time to run it on the hardware. Before that, it was necessary to initialize the hardware.

According to chapter 11 of the reference manual [23], the chip uses four input pins in order to determine the boot mode. Unfortunately, the starter kit we are using does not allow changing the values of the pins. They are fixed to set the QSPI mode where the first stage and second stage bootloaders are loaded. Those expect a Linux image to be present on the SD card which is not our case. The first goal is therefore to change the boot mode to the JTAG mode. After that, there will be no bootloader and all initialization will be done via JTAG.

In order to switch the boot mode, it is necessary to change the value of a dedicated register and then perform a system reset. The following Tcl commands may be used:

```
targets -set -filter {name == "PSU"}
mwr 0xff5e0200 0x100
rst -system
```

The process of initializing the hardware is rather complicated and involves writing specific values to hundreds of registers. The simplest way to obtain the configuration is to use Vivado and configure the processing system IP as described in section 5.2.2. During the synthesis of that IP, the initialization script called `psu_init` is created in both C and Tcl languages. For simple access, I extracted this file from the `ip` directory and placed it to the `tcl` directory.

The script has thousands of lines and consists of several functions. I was able to isolate three functions that must be executed before running the software:

- The `psu_init` function initializes the processing system. Among other things, it configures the mapping of the multiplexed inputs and outputs and the frequency of the clocks.

- The `psu_ps_pl_reset_config` function sets up and toggles the reset signal going from the processing system to the programmable logic. Without that, the design was held in reset permanently.
- The `psu_ps_pl_isolation_removal` function enables the programmable logic. Without that, the AXI interface between the processing system and the programmable logic was not working.

In order to speed up the whole process, I combined everything into a procedure in Tcl and put that into the `functions.tcl` file. After sourcing that file using the `source functions.tcl` command, initializing the hardware is as simple as running the following command:

```
init
```

7.4.3 Loading the bitstream

The final product of the hardware designed in chapter 3 and implemented in chapter 5 is the **bitstream** in BIT format. It must be loaded to the PL (programmable logic) part of the chip during the boot process. It is necessary to load the bitstream via JTAG in our case because we are bypassing the bootloader as shown above. Using this method is more convenient than having to rebuild the bootloader every time the bitstream changes during the development process.

In order to load the BIT file, I created the following procedure in Tcl and put it into the `functions.tcl` file:

```
proc bit {bit} {
    fpga $bit
}
```

After sourcing that file using the `source functions.tcl` command, loading the bitstream is as simple as running the following command:

```
bit hw/bin/ethernet_system.bit
```

7.4.4 Loading the binary

The final product of the software designed in chapter 6 and compiled in chapter 7 is the **binary** in ELF format. It must be loaded to the PS (processing system) part of the chip during the boot process. It is necessary to load the binary via JTAG in our case because we are bypassing the bootloader as shown above. Using this method is more convenient than having to rebuild the bootloader every time the binary changes during the development process.

In order to load the ELF file, I created the following procedure in Tcl and put it into the `functions.tcl` file:

```
proc elf {elf} {
    targets -set -filter {name == "Cortex-R5 #0"}
    rst -processor
    dow $elf
    con
}
```

After sourcing that file using the `source functions.tcl` command, loading the binary is as simple as running the following command:

```
elf sw/bin/sync.elf
```

Chapter 8

Testing the Device

This chapter shows the final and the most complex tests which require cooperation of the custom hardware designed in chapter 3 and the custom software designed in chapter 6. It is divided into the following sections:

- In section 8.1, the test of passing Ethernet frames through our device is described.
- In section 8.2, the test of transmitting and receiving Ethernet frames by our device through loopback is described.
- In section 8.3, the test of interaction of our device with other devices by handling ICMP messages is described.
- In section 8.4, the test of time synchronization through our device by handling PTP messages is described.

8.1 Passthrough test

The purpose of the passthrough test is to make sure that Ethernet frames can pass through all paths of our device. It is actually a series of tests where each of them enables one or two paths and disables the rest of them.

Our device is inserted in the middle of an Ethernek link connecting a generic computer and a generic router. Once our device is set up as required by the test, the computer tries connecting to the internet via the router. For some tests, the internet connection speed is also measured because dropping frames often causes slowing down the connection.

The computer also has another network card which can be connected to the data acquisition port of the device to test logging or diverting of the frames.

8.1.1 Setup

1. The Ethernet ports of our device were connected to other devices depending on the variant of the test.
2. The Ethernet system described in section 3.15.5 was implemented and loaded into the programmable logic of our device.
3. The Passthrough example described in section 6.6.1 was built and loaded into the processing system of our device.

GE variant

The first variant of connection was used to test passing frames between the GE ports and logging frames to the XGE port. It is shown in figure 8.1.

- The GE1 port of our device is connected to the 1 Gb/s USB Ethernet of the computer.
- The GE2 port of our device is connected to the 1 Gb/s LAN Ethernet of the router.
- The XGE port of our device is connected to the 2.5 Gb/s PCI Ethernet of the computer.

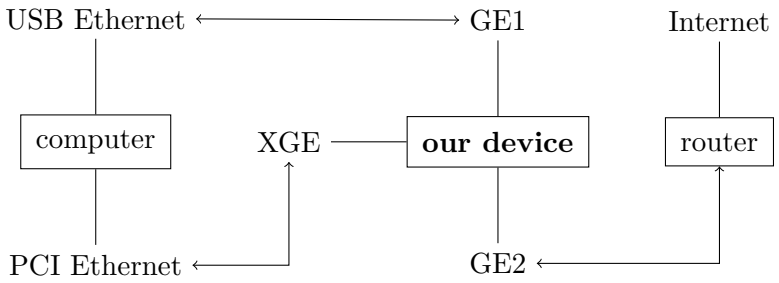

Figure 8.1: Passthrough test setup for GE

XGE variant

The second variant of connection was used to test diverting frames between the GE port and the XGE port. It is shown in figure 8.2.

- The GE1 port of our device is connected to the 1 Gb/s USB Ethernet of the computer.
- The GE2 port of our device is connected to the 2.5 Gb/s PCI Ethernet of the computer.
- The XGE port of our device is connected to the 1 Gb/s LAN Ethernet of the router.

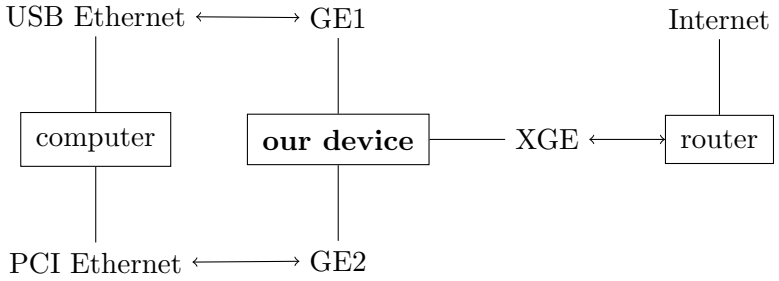

Figure 8.2: Passthrough test setup for XGE

8.1.2 Result

The passthrough test consists of several tests which were run one by one.

Log from GE1 and GE2

This test checks logging frames from both GE ports.

Before starting the test, the RX splitter of both GE ports was configured to log all frames by default.

During the test, the computer was not able to connect to the internet because no frames could pass through our device. However, it was possible to see the communication from both GE ports on the XGE port thanks to the logging via the CMP protocol. The computer was transmitting ARP requests to the router. The router was transmitting broadcast traffic to the computer.

After stopping the test, the program printed the following values of the counters:

```
GE1 rx=258 tx=0
GE2 rx=484 tx=0
```

In total, 742 frames were received and 0 frames were transmitted. Wireshark listening on the XGE port captured only 659 frames because one CMP package may contain more CMP payloads. The value of the sequence counter in the last package was equal to the count of the packages.

Pass between GE1 and GE2

This test checks passing frames between GE1 and GE2.

Before starting the test, the RX splitter of both GE ports was configured to pass all frames by default.

During the test, the computer was able to connect to the internet. When measuring the internet connection speed, the results were the same as if our device was not inserted in the middle of the link.

After stopping the test, the program printed the following values of the counters:

```
GE1 rx=117768 tx=316805
GE2 rx=316805 tx=117768
```

In total, 117 768 frames were passed from GE1 to GE2 and 316 805 frames were passed from GE2 to GE1. It holds that the RX count of GE1 equals the TX count of GE2 and the RX count of GE2 equals the TX count of GE1.

Pass only from GE1 to GE2

This test checks passing frames only from GE1 to GE2 and not the other way round.

Before starting the test, the RX splitter of GE1 was configured to pass all frames by default and the RX splitter of GE2 was configured to drop all frames by default.

During the test, the computer was not able to connect to the internet. It was not able to receive any frames from the router which was checked by running Wireshark on the USB Ethernet NIC of the computer.

After stopping the test, the program printed the following values of the counters:

```
GE1 rx=275 tx=0
GE2 rx=274 tx=275
```

In total, 275 frames were passed from GE1 to GE2 and 274 frames were received on GE2 and dropped. It holds that the RX count of GE1 equals the TX count of GE2.

Pass only from GE2 to GE1

This test checks passing frames only from GE2 to GE1 and not the other way round.

Before starting the test, the RX splitter of GE2 was configured to pass all frames by default and the RX splitter of GE1 was configured to drop all frames by default.

During the test, the computer was not able to connect to the internet. It was able to receive frames from the router which was checked by running Wireshark on the USB Ethernet NIC of the computer. However, no frames transmitted by the computer could reach the router.

After stopping the test, the program printed the following values of the counters:

```
GE1 rx=532 tx=194
GE2 rx=194 tx=0
```

In total, 194 frames were passed from GE2 to GE1 and 532 frames were received on GE1 and dropped. It holds that the RX count of GE2 equals the TX count of GE1.

Divert from GE1 to XGE

This test checks diverting frames between GE1 and XGE.

Before starting the test, the RX splitter of GE1 was configured to divert all frames by default and the RX splitter of GE2 was configured to drop all frames by default. The splitter of XGE always passes all received frames to all ports.

During the test, the computer was able to connect to the internet via the USB Ethernet NIC. When measuring the internet connection speed, the results were the same as if our device was not inserted in the middle of the link.

Unfortunately, there are no counters in the XGE MAC so it was not possible to compare the RX count of GE1 and the TX count of XGE.

Divert from GE2 to XGE

This test checks diverting frames between GE2 and XGE.

Before starting the test, the RX splitter of GE2 was configured to divert all frames by default and the RX splitter of GE1 was configured to drop all frames by default. The splitter of XGE always passes all received frames to all ports.

During the test, the computer was able to connect to the internet via the PCI Ethernet NIC. When measuring the internet connection speed, the results were the same as if our device was not inserted in the middle of the link.

Unfortunately, there are no counters in the XGE MAC so it was not possible to compare the RX count of GE2 and the TX count of XGE.

Log ARP frames and pass all frames

This test checks matching frames by the EtherType field of the Ethernet header.

Before starting the test, the RX splitter of both GE ports was configured as follows:

- The 1st matcher compared the EtherType field of the Ethernet header to **0x0806** (ARP). The splitter logged and passed the frame in case of a match.
- The splitter passed the frame in all other cases.

During the test, the computer was able to connect to the internet because all frames could pass through our device. Wireshark listening on the XGE port captured ARP frames going in both directions and no other frames as requested by the setup of the RX splitter.

Divert ICMP frames and pass other frames

This test checks matching frames by the Protocol field of the IPv4 header.

Before starting the test, the RX splitter of GE2 was configured to pass all frames by default and the RX splitter of GE1 was configured as follows:

- The 1st matcher compared the Protocol field of the IPv4 header to **0x01** (ICMP) in addition to checking the EtherType field. The splitter diverted the frame in case of a match.
- The splitter passed the frame in all other cases.

During the test, the computer was able to connect to the internet because most frames could pass through our device. However, it was not able to ping the router because the ICMP requests were diverted to the XGE port as confirmed by capturing the CMP frames on that port in Wireshark.

Divert HTTP frames and pass other frames

This test check matching frames by the Destination Port field of the TCP header.

Before starting the test, the RX splitter of GE2 was configured to pass all frames by default and the RX splitter of GE1 was configured as follows:

- The 1st matcher compared the Destination Port field of the TCP header to **80** (HTTP) in addition to checking the EtherType and the Protocol fields. The splitter diverted the frame in case of a match.
- The 2nd matcher compared the Destination Port field of the TCP header to **443** (HTTPS) in addition to checking the EtherType and the Protocol fields. The splitter diverted the frame in case of a match.
- The splitter passed the frame in all other cases.

During the test, the computer was able to connect to the internet because most frames could pass through our device. However, it was not able to browse the web because the HTTP and HTTPS requests were diverted to the XGE port as confirmed by capturing the CMP frames on that port in Wireshark.

Summary

The passthrough test was successful.

8.2 Loopback test

The purpose of the loopback test is to make sure that the Ethernet data path for transmission and reception does not introduce any errors into the Ethernet frames generated and checked by our device. The two Ethernet ports of our device are directly or indirectly connected together. The frame generator periodically transmits Ethernet frames. The frame checker receives Ethernet frames and compares the received data to the expected data.

From the hardware perspective, every frame must successfully pass the DMA and MAC transmitters, the Ethernet PHY, the MAC and DMA receivers and many other components.

From the software perspective, every frame must be correctly handled by the drivers.

8.2.1 Setup

1. The Ethernet ports of our device were connected to each other depending on the variant of the test.
2. The Ethernet system described in section 3.15.5 was implemented and loaded into the programmable logic of our device.
3. The Loopback example described in section 6.6.2 was built and loaded into the processing system of our device.
    - The GE1 port was used as the TX port.
    - The GE2 port was used as the RX port.

Cable variant

In the first variant, the GE1 and GE2 ports were directly connected via an Ethernet cable as shown in figure 8.3.

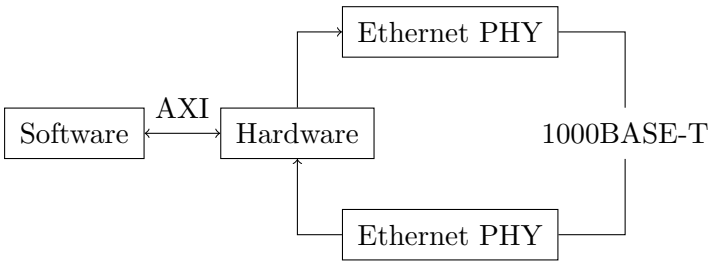

Figure 8.3: Loopback test with direct connection

Switch variant

In the second variant, the GE1 and GE2 ports were connnected through a generic Ethernet switch as shown in figure 8.4. The other ports of the switch were left unconnected.

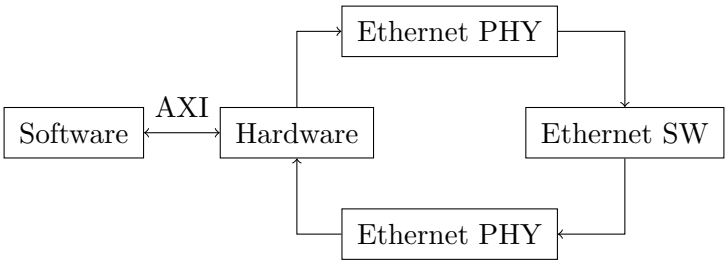

Figure 8.4: Loopback test with generic switch

Converter variant

In the third variant, the GE1 and GE2 ports were first converted to automotive Ethernet and then connected via an Ethernet cable as shown in figure 8.5. To be specific, the conversion was done from 1000BASE-T to 1000BASE-T1 by external media converters.

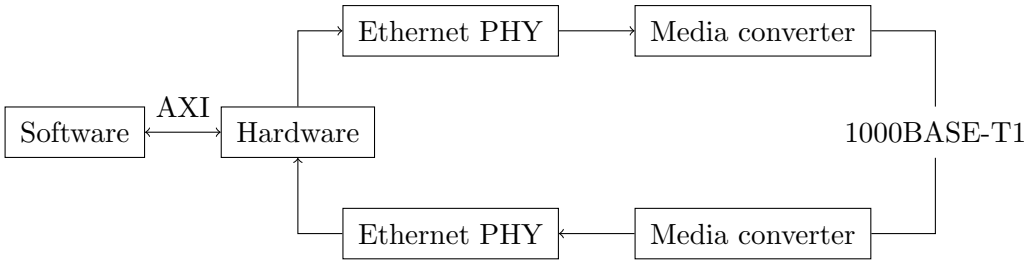

Figure 8.5: Loopback test with media converters

8.2.2 Result

The loopback test was successful. It was kept running for more than 30 minutes without receiving any invalid frames. During that time, the TX process transmitted 18 821 250 frames and the RX process received 18 821 250 valid frames. That means no frames were lost on the way.

In addition to that, the time spent on generating and checking the frames was measured to evaluate the performance of the processing system. In average, the TX process spent 3.25 microseconds generating one frame and the RX process spent 5.94 microseconds checking one frame. The length of every frame was 100 bytes. The frequency of the processor was about 500 MHz so the period was about 2 nanoseconds. Having said that, the TX process spent 16 cycles generating one byte of data and 30 cycles checking one byte of data in average.

Our device produced the following output at the end of the test:

```
tx_total = 18821250
rx_total = 18821250
rx_valid = 18821250
rx_invalid = 0
tx_time = 325
rx_time = 594
```

8.3 Ping test

The purpose of the ping test is to test the interaction between our device and other devices on the same network. One of the Ethernet ports is connected to a generic computer running a general purpose operating system. Both our device and the computer are assigned a private IPv4 address and a link local IPv6 address. Our device is then supposed to respond to ICMP echo requests and also other messages required for translating the IP address to the MAC address.

8.3.1 Setup

1. The Ethernet ports of our device were connected to other devices as follows:
   - The upper port of our device (referred to as GE1 in software) was connected to the computer running the ping utility.
2. The Ethernet system described in section 3.15.5 was implemented and loaded into the programmable logic of our device.
3. The Ping example described in section 6.6.3 was built and loaded into the processing system of our device.
   - The GE1 port was used in the ARP and ICMP handler.
   - The XGE port was set up to log everything using the CMP protocol.

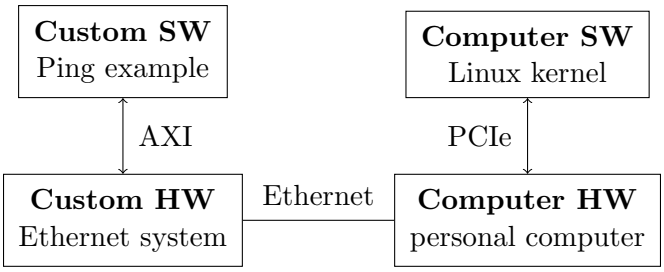

Figure 8.6: Ping test setup

8.3.2 Result

IPv4 variant

In the first variant, the ICMPv4 protocol was used.

- Our device was assigned IPv4 address **192.168.0.10**.
- The computer was assigned IPv4 address **192.168.0.20**.

The command `ping 192.168.0.10` was kept running on the computer for 70 seconds and produced the following output:

```
PING 192.168.0.10 (192.168.0.10) 56(84) bytes of data.
64 bytes from 192.168.0.10: icmp_seq=1 ttl=255 time=0.437 ms
64 bytes from 192.168.0.10: icmp_seq=2 ttl=255 time=0.415 ms
64 bytes from 192.168.0.10: icmp_seq=3 ttl=255 time=0.407 ms
...
--- 192.168.0.10 ping statistics ---
70 packets transmitted, 70 received, 0% packet loss, time 70650ms
rtt min/avg/max/mdev = 0.298/0.400/0.480/0.037 ms
```

The average round trip time was **0.400 ms** which is typical for devices connected to the same network.

IPv6 variant

In the second variant, the ICMPv6 protocol was used.

- Our device was assigned IPv6 address **fe80::1**.
- The computer was assigned IPv6 address automatically.

When using a link local address, it is necessary to specify the interface to use by appending its name after the percent sign.

The command `ping fe80::1%eth0` was kept running on the computer for 70 seconds and produced the following output:

```
PING fe80::1%eth0(fe80::1%eth0) 56 data bytes
64 bytes from fe80::1%eth0: icmp_seq=1 ttl=255 time=0.315 ms
64 bytes from fe80::1%eth0: icmp_seq=2 ttl=255 time=0.440 ms
64 bytes from fe80::1%eth0: icmp_seq=3 ttl=255 time=0.356 ms
...
--- fe80::1%eth0 ping statistics ---
70 packets transmitted, 70 received, 0% packet loss, time 70634ms
rtt min/avg/max/mdev = 0.311/0.424/0.468/0.036 ms
```

The average round trip time was **0.424 ms** which is typical for devices connected to the same network but slightly higher than in the IPv4 variant.

8.4 Sync test

The purpose of the sync test is to verify the software implementation of the PTP protocol for propagating time synchronization across Ethernet ports of our device. It requires two devices with support of PTP protocol and hardware timestamping. The first device is set to master mode to become the source of the time synchronization. The second device is set to slave mode to become the target of the time synchronization. Our device is then inserted between those two devices so that the time is synchronized through our device.

8.4.1 Setup

1. The Ethernet ports of our device were connected to other devices as follows:
   - The upper port of our device (referred to as GE1 in software) was connected to the PTP master device (Raspberry Pi 5 in our case).
   - The lower port of our device (referred to as GE2 in software) was connected to the PTP slave device (another AMD Kria in our case).
   - The XGE port of our device was connected to a computer for logging the traffic.
2. The Linux PTP utility was run on the PTP master device (Raspberry Pi 5 in our case) as described below.
3. The Linux PTP utility was run on the PTP slave device (another AMD Kria in our case) as described below.
4. The Ethernet system described in section 3.15.5 was implemented and loaded into the programmable logic of our device.
5. The Sync example described in section 6.6.4 was built and loaded into the processing system of our device.
   - The GE1 port was used as the **slave port** of the PTP handler.
   - The GE2 port was used as the **master port** of the PTP handler.
   - The XGE port was set up to log everything using the CMP protocol.

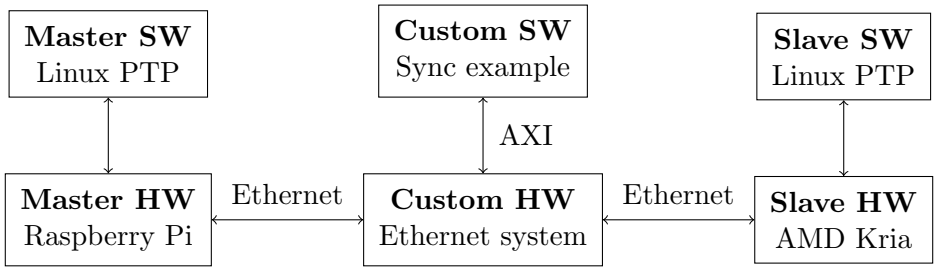

Figure 8.7: Sync test setup

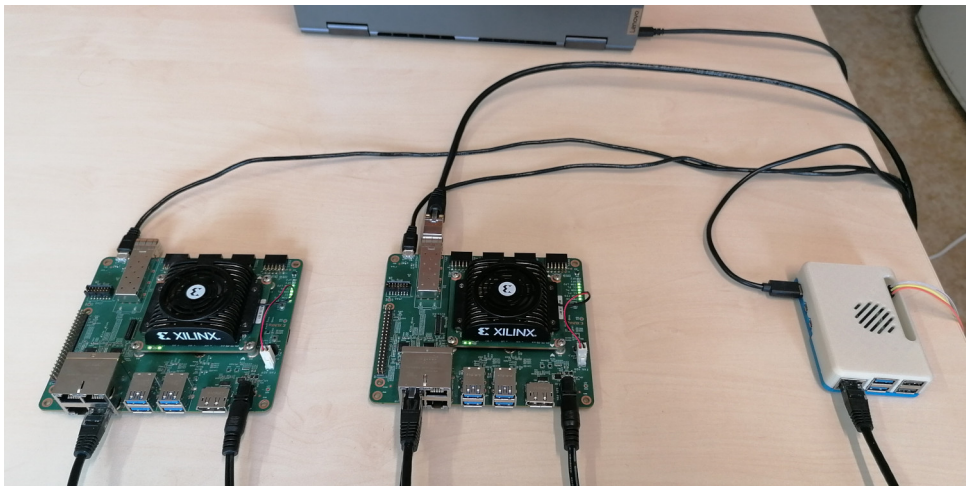

Figure 8.8: Image of sync test setup

Master setup

As the PTP master device, we used a Raspberry Pi 5. It is the first version of Raspberry Pi to support hardware timestamping which is required for precise time synchronization. It actually uses the same Cadence IP to implement the Ethernet MAC as the Zynq UltraScale+ chip in AMD Kria boards.

After I flashed the SD card with a pre-built Raspbian image, I installed the `linuxptp` package from the standard repository by running the following command:

```
apt install linuxptp
```

To simplify configuration, the package should contain a configuration file for the automotive master profile. I could not find it anywhere so I downloaded the file from the source repository of the project. To run the PTP daemon in master mode, I used the following command:

```
sudo ptp4l -i eth0 -f automotive-master.cfg -m
```

Slave setup

As the PTP slave device, we used another AMD Kria KR260 starter kit. It supports hardware timestamping (at least when using the latest version of kernel) which is required for precise time synchronization.

After I flashed the SD card with a pre-built Petalinux image, I installed the `linuxptp` package from the standard repository by running the following command:

```
dnf install linuxptp
```

To simplify configuration, the package should contain a configuration file for the automotive slave profile. I could not find it anywhere so I downloaded the file from the source repository of the project. To run the PTP daemon in slave mode, I used the following command:

```
sudo ptp4l -i eth0 -f automotive-slave.cfg -m
```

8.4.2 Result

To check whether our implementation of the PTP protocol is correct, the time synchronization between master and slave was attempted several times with the same software configuration but different hardware configuration.

Direct connection

In the first variant, the master and slave devices were directly connected with an Ethernet cable so our device was not involved at all.

The time synchronization was successful. It took about 10 seconds to stabilize. After running stable for 48 seconds, the offset jumped between **-27 ns** and **+12 ns** with an average of -8 ns which was far better than expected. The measured frequency offset was between +33925 ppb and +34064 ppb with an average of +33989 ppb which is within the 100 ppm tolerance. The measured path delay was between 397 ns and 406 ns with an average of **402 ns**. That includes the RX and TX latency of the PHY in addition to the propagation delay of the Ethernet cable because the timestamping is done in the MAC on our board.

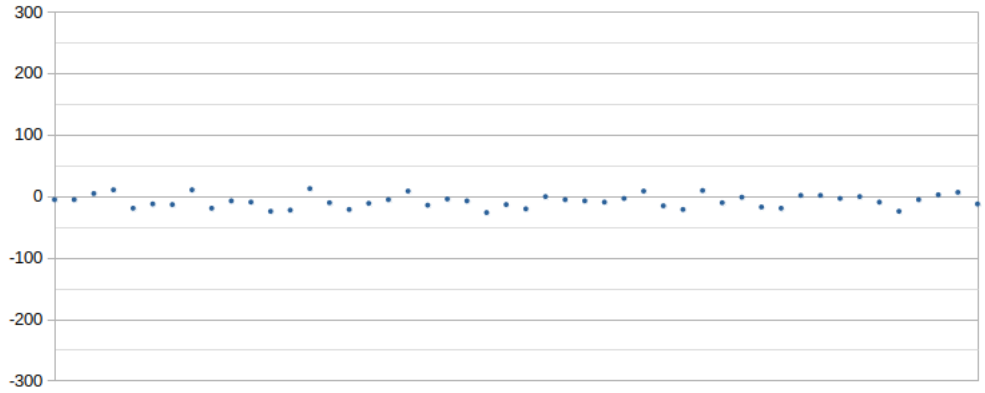

Figure 8.9: Offset of time synchronization with direct connection

Passive mode

In the second variant, the master and slave devices were indirectly connected through our device which was set up to pass all frames in both directions without any modifications.

The time synchronization was successful. It took about 15 seconds to stabilize. After running stable for 72 seconds, the offset jumped between **-1 ns** and **+50 ns** with an average of +23 ns which was slightly worse than in the first variant. The measured frequency offset was between +41572 ppb and +42077 ppb with an average of +41848 ppb. The measured path delay was between 1063 ns and 1072 ns with an average of **1068 ns**. That reflected the fact that every frame had to pass through four transceivers in total where each of them added between 200 ns and 300 ns of latency.

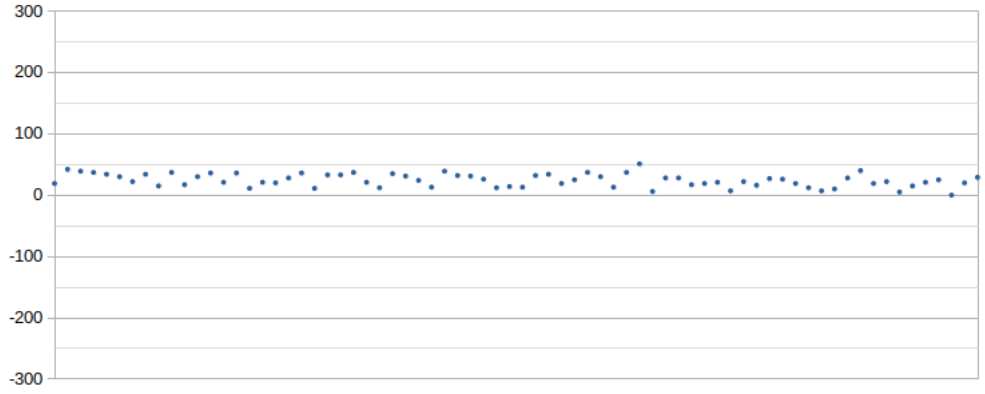

Figure 8.10: Offset of time synchronization in passive mode

Active mode

In the third variant, the master and slave devices were indirectly connected through our device which was set up to catch PTP frames and participate in both delay measurement and time synchronization process.

The time synchronization was successful. It took about 20 seconds to stabilize. After running stable for 90 seconds, the offset jumped between **-621 ns** and **+254 ns** with an average of -5 ns which was worse than in the second variant. The measured frequency offset was between +34064 ppb and +35300 ppb with an average of +34769 ppb. The measured path delay was between 434 ns and 438 ns with an average of **436 ns**. That confirmed the fact that our device participated in the delay measurement process.

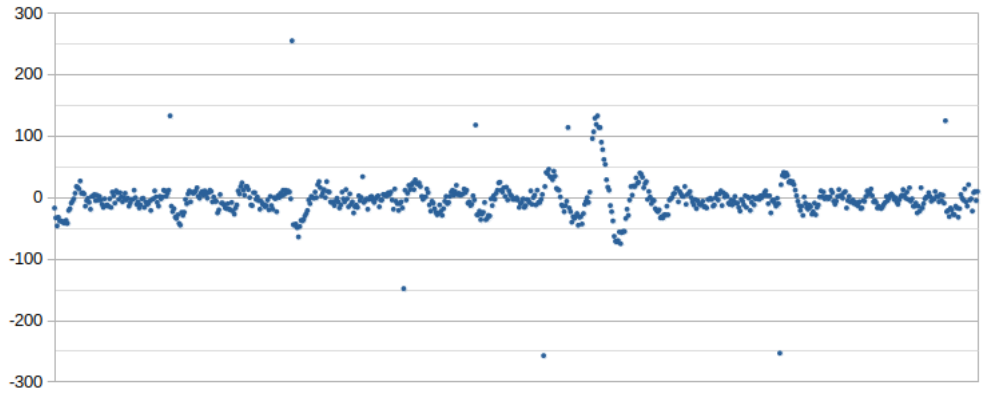

Figure 8.11: Offset of time synchronization in active mode

Chapter 9

Conclusion

Finally, the last chapter comes where I will summarize my bachelor thesis, point out the results that have been achieved and list the features that could be improved or added. This chapter is divided into the following sections:

- In section 9.1, I sum up the features of the final device and compare them to the original assignment.
- In section 9.2, I discuss which existing features could be improved for better reliability and performance.
- In section 9.3, I discuss which new features could be added in future development.

9.1 What has been done

The final device consists of the hardware designed in chapter 3 and the software designed in chapter 6. The hardware was verified in simulation as described in chapter 4 and then implemented in the programmable logic as described in chapter 5. The software was built and run in the processing system as described in chapter 7. The device was finally tested as described in chapter 8.

The device has two Ethernet ports which support 100 Mb/s and 1 Gb/s Ethernet. Those can be used to tap into one segment of the automotive Ethernet network when combined with two media converters.

The device also has one SFP+ module slot which supports 10 Gb/s Ethernet. That can be used to watch and log the traffic on the automotive Ethernet network when populated with a SFP+ module and connected to a computer.

Handling received frames

Every frame received on one of the Ethernet ports can have one or more of the following actions applied depending on the configuration:

- It can be **passed** to the other Ethernet port of the pair without any change.
- It can be **logged** to the 10 Gb/s Ethernet port packaged in the CMP protocol.
- It can be **diverted** to the 10 Gb/s Ethernet port without any change.
- It can be **processed** by software running in the processing system.

Handling transmitted frames

Every frame transmitted on one of the Ethernet ports can have one or more of the following actions applied depending on the configuration:

- It can be **logged** to the 10 Gb/s Ethernet port packaged in the CMP protocol.
- It can be **processed** by software running in the processing system.

Summary

All requirements given by the assignment have been fulfilled:

1. Bridge function on the automotive Ethernet ports with support for filtering and mirroring communication on the data acquisition port has been implemented in hardware.
2. PTP protocol support on the automotive Ethernet ports has been implemented in software.
3. CMP protocol support on the data acquisition port has been implemented in hardware.

9.2 What could be improved

During the design process, I had a lot of ideas how the design could be improved. However, I was not able to work on all of them before I had to hand off the thesis due to the overall complexity of the project. The purpose of this section is to save those ideas for future development.

Adding counters

Every component working with Ethernet frames should include counters for frames coming in to the component and frames going out from the component.

The counters could be used by various tests in order to check whether any frames were lost on their way through the system which would indicate an error in the design. That would increase the *reliability* of the system as it could be tested more thoroughly.

The counters could be also used by user applications in order to measure current and total utilization of the Ethernet link. That would enhance the *functionality* of the system.

Detecting overflow

Every component buffering Ethernet frames should be able to detect overflow of the internal FIFO used to store the frames. The overflow would be recorded in hardware by incrementing a counter and also indicated to software by triggering an interrupt in order to report the loss of the frame. That would increase the *reliability* of the system.

The DMA controller transporting Ethernet frames from and to system memory should also detect overflow of the RX and TX buffer in the memory. Cooperation of hardware and software would be required to keep track of the current utilization of the buffer. In case the frame was too long to fit into the buffer, the transport would be delayed until enough space was available in the buffer. That would increase the *reliability* of the system.

With the current version of hardware and software, it is not possible to handle that situation. The buffer may overflow when the frames are either received by hardware faster than processed by software or generated by software faster than transmitted by hardware. The first case is usually solved by setting up the splitter to filter the frames passed to the DMA controller. The second case is currently not possible because the generation is much slower than the transmission.

Using interrupts

Events originating in hardware should be communicated to software by triggering an interrupt. That includes receiving a frame on one of the Ethernet ports, transmitting a frame on one of the Ethernet ports, finishing a management transaction with the Ethernet transceiver, receiving input from the UART controller and so on. That would increase the *performance* of the system.

At the moment, the detection of all those events is done by polling the respective register in hardware from software. The polling interval is usually set to tens or hundreds of microseconds so that the system bus is not used too much. However, that singnificantly increases the latency of handling the event. Moreover, that is possible only in the bare metal environment because there are no other programs running and will not be possible in the Linux environment.

More extensive testing

The testing has been limited only to the minimum to make sure that our device does exactly what we want it to do under normal conditions. Much more testing will have to be done before we are able to claim that our device is reliable. It will probably no longer be sufficient to test something manually. It will be necessary to test everything automatically.

In simulation, the framework for automated testing has already been developed. However, the testbenches created for testing both the individual components and the whole system are mostly very simple. In our case, they usually transmit and receive only one or two types of frames in a loop in order to ensure that the component works correctly in general. In proper testing, they should always transmit and receive frames of every possible length starting with the minimal frame length and ending with the maximal frame length.

On real device, no framework for automated testing has been developed. My idea was that two boards could be connected together. The first one would be the unit under test with the current version of hardware and software loaded in it. The second one would make the testbench. It could be either running Linux or loaded with the stable version of my hardware and software. The testbench would transmit and receive Ethernet frames just like in simulation. It could even run the same set of tests as in simulation.

9.3 What could be done next

Since the beginning of this project, it has been clear that there is much more work than I will be able to cover in my thesis. The purpose of this section is to discuss the possible next steps of this project.

Designing custom motherboard

The starter kit used for the initial implementation and testing is not ideal for this project. Although it has four Ethernet ports in total, only two of them are available to the programmable logic and the remaining two of them are directly connected to the processing system. Another thing is that we need automotive Ethernet transceivers instead of industrial Ethernet transceivers so that the device could be plugged into the vehicle directly without the media converters. Finally, there are many other minor modifications that we would like to make on the board.

Since the beginning, it has been clear that this project will require designing a custom motherboard. The AMD Kria ecosystem is well prepared for it. The starter kit actually consists of the AMD Kria K26 module with the system on chip and the DDR memory and the AMD Kria KR260 board with other peripherals which we do not need. It is therefore possible to replace the board and keep the same module. They also provide the schematic of the board which can be used as a source of inspiration for our custom design.

The custom motherboard used for the final implementation should include the following peripherals:

- 8 automotive Ethernet transceivers supporting both 100 Mb/s and 1 Gb/s Ethernet will be connected to the high performance I/O of the programmable logic via RGMII.
- 2 industrial Ethernet transceivers supporting up to 1 Gb/s Ethernet for management will be connected to the gigabit transceivers of the processing system via SGMII.
- 2 slots for SFP+ modules supporting up to 10 Gb/s Ethernet for data acquisition will be connected to the gigabit transceivers of the programmable logic.
- 6 CAN transceivers will be connected to the high density I/O of the programmable logic.
- 6 LIN transceivers will be connected to the high density I/O of the programmable logic.
- 2 USB transceivers will be connected to the multiplexed I/O of the processing system.
- 1 microSD card slot will be connected to the multiplexed I/O of the processing system.
- 1 FTDI chip will be connected to the JTAG and UART interfaces of the processing system for programming and debugging.

Adding support for CAN and LIN

With the custom motherboard, it would be desirable to extend the hardware implemented in the programmable logic with more components to provide support for CAN (Controller Area Network) and LIN (Local Interconnect Network). These interfaces with bus topology are slower but cheaper than automotive Ethernet so they are still used a lot in the automotive industry.

The CAN bus is usually used to connect the electronic control units together.

The LIN bus is usually used to connect sensors and actuators to the electronic control units.

The features and hence the structure could be actually very similar to the Ethernet system. Every frame would be examined and one or more of the following actions would be applied:

- It could be **logged** to the 10 Gb/s Ethernet port packaged in the CMP protocol.
- It could be **diverted** to another CAN/LIN interface without any change.
- It could be **processed** by software running in the processing system.

Moving to Linux

The bare metal environment was the simplest way to get our custom hardware and software working together. However, it does not allow running multiple programs together because there is no operating system to switch between them, neither it allows running programs developed by others because there is no standard API to be called by them. For future development of this project, a proper operating system will be required. Fortunately, it is possible to run full fledged Linux on the APU (application processing unit) part of the processing system. It should be even possible to run a bare metal program on the RPU (real-time processing unit) part at the same time.

Running Linux will provide many advantages, starting with the ones mentioned above. For example, it will be possible to build and run the industry standard penetration testing tools available for Linux. As another example, it will be much easier to manage the device remotely by using protocols like SSH for remote access to the terminal or HTTP for access to a web server providing a user friendly way to configure the hardware.

Developing drivers for Linux

The first step of moving to Linux would be developing drivers similar to the ones created for the bare metal environment in section 6.2. On Linux, it is usually not possible to directly access the memory from the user space. Although it is possible to map the memory when running the program with root privileges, it is definitely not the correct way to do it. The driver should rather run in the kernel space where it has direct access to the memory and provide a way to control the hardware from the user space. There would be two different kinds of drivers actually:

First, there would be drivers for components that allow some kind of configuration like splitters and joiners. Those would have to use one of the standard ways of communication between the user space and the kernal space in Linux. For example, they could create a device file for every component and use custom `ioctl` commands in order to get or set the configuration.

Second, there would be drivers for the DMA receiver and transmitter that allow receiving or transmitting Ethernet frames. Those would have to communicate with the network subsystem of Linux so that every Ethernet port implemented in the programmable logic appeared as a network interface in Linux. Once that was done, it would be possible to use any existing programs to receive and transmit Ethernet frames on any Ethernet port of the device.

Adding user interface

Once Linux is running on the board, it would be nice to develop a CLI (command line interface) utility and possibly a GUI (graphical user interface) utility for controlling the system. At the moment, the custom hardware can be controlled either by directly writing to the memory (which is not possible on Linux anyway) or by building a program which configures the hardware for a specific situation. It would be more user friendly if there was a utility capable of configuring the hardware for any situation by running commands from the command line. Access to the utility would be possible either via the serial port or even via the Ethernet port thanks to remote access protocols like SSH.

List of Figures

List of Tables

Bibliography

[1] *10G/25G High Speed Ethernet Subsystem v4.1 Product Guide*. PG210. v4.1. Advanced Micro Devices, Inc. May 2023.

[2] *AMBA AXI and ACE Protocol Specification*. IHI 0022E. ARM. Feb. 2013.

[3] *Capture Module Protocol. Protocol Layer Specification*. Version 1.0.0. ASAM e.V. Mar. 2022.

[4] *DP83867E/IS/CS Robust, High Immunity, Small Form Factor 10/100/1000 Ethernet Physical Layer Transceiver*. SNLS504D. Rev. D. Texas Instruments, Inc. Nov. 2022.

[5] *GMII to RGMII v4.1 LogiCORE IP Product Guide*. PG160. v4.1. Xilinx, Inc. June 2022.

[6] *IEEE Standard for Ethernet*. IEEE Std 802.3-2022. LAN/MAN Standards Committee of the IEEE Computer Society. 2022.

[7] *IEEE Standard for Local and Metropolitan Area Networks — Timing and Synchronization for Time-Sensitive Applications*. IEEE Std 802.1AS-2020. LAN/MAN Standards Committee of the IEEE Computer Society. 2020.

[8] *Kria K24 SOM Data Sheet*. DS985. v1.1. Advanced Micro Devices, Inc. Mar. 2024.

[9] *Kria K26 SOM Data Sheet*. DS987. v1.4. Advanced Micro Devices, Inc. July 2023.

[10] *Kria KR260 Robotics Starter Kit Data Sheet*. DS988. v1.0. Advanced Micro Devices, Inc. May 2022.

[11] *Kria KV260 Vision AI Starter Kit Data Sheet*. DS986. v1.1. Advanced Micro Devices, Inc. Mar. 2022.

[12] *Management Interface for SFP+*. SFF-8472. Rev 12.4. SFF Committee. Mar. 2021.

[13] *SFP+ 10 Gb/s Electrical Interface*. SFF-8418. Rev 1.4. SFF Committee. July 2015.

[14] *SFP+ Power and Low Speed Interface*. SFF-8419. Rev 1.3. SFF Committee. June 2015.

[15] *UltraFast Design Methodology Guide for FPGAs and SoCs*. UG949. v2023.1. Advanced Micro Devices, Inc. June 2023.

[16] *UltraScale Architecture Configurable Logic Block User Guide*. UG574. v1.5. Xilinx, Inc. Feb. 2017.

[17] *UltraScale Architecture GTH Transceivers User Guide*. UG576. v1.7.1. Xilinx, Inc. Aug. 2021.

[18] *UltraScale Architecture Memory Resources User Guide*. UG573. v1.13. Xilinx, Inc. Sept. 2021.

[19] *UltraScale Architecture SelectIO Resources User Guide*. UG571. v1.14. Xilinx, Inc. Sept. 2022.

[20] *Vivado Design Suite Tcl Command Reference Guide*. UG835. v2023.1. Advanced Micro Devices, Inc. May 2023.

[21] *Vivado Design Suite User Guide. Design Flows Overview*. UG892. v2023.1. Advanced Micro Devices, Inc. May 2023.

[22] *Vivado Design Suite User Guide. Using Constraints*. UG903. v2023.1. Advanced Micro Devices, Inc. May 2023.

[23] *Zynq UltraScale+ Device Technical Reference Manual*. UG1085. v2.3.1. Advanced Micro Devices, Inc. Jan. 2023.

[24] *Zynq UltraScale+ MPSoC Processing System v3.5 LogiCORE IP Product Guide*. PG201. v3.5. Advanced Micro Devices, Inc. June 2023.