# Design, Implementation, and Test of a Tri-Mode Ethernet MAC on an FPGA

by

Philipp Kerling

2015-09-18

TECHNISCHE UNIVERSITÄT
ILMENAU

Substantial parts of this document were previously submitted as thesis in partial fulfillment for the degree of Bachelor of Science at Technische Universität Ilmenau, Department of Electrical Engineering and Information Technology, Electronic Measurement Engineering Group. See https://find.bibliothek.tu-ilmenau.de/Record/834804387.

# *Abstract*

Technische Universität Ilmenau
Department of Electrical Engineering and Information Technology
Electronic Measurement Engineering Group

**Design, Implementation, and Test of a Tri-Mode Ethernet MAC on an FPGA**

by Philipp Kerling

Ethernet is a mature communication technology with a sizable number of advantages for sensor networks and many other use cases. Applications interact with a media access control (MAC) sublayer to get packets from and to the network. This thesis presents the design, implementation, and test of an Ethernet MAC targeted at FPGAs in VHDL. It supports full-duplex interfacing with an Ethernet physical layer integrated circuit via the standard media-independent interface (MII) variants MII and GMII at link speeds of 10, 100, and 1000 Mb/s. In contrast to prior solutions, it is devised primarily with simplicity in mind, both in external usage and in internal operation. Users benefit from the straightforward FIFO interface offered while nevertheless being able to send and receive at almost full Gigabit Ethernet speed. The solution is designed for and verified on a Trenz Electronic GmbH GigaBee micromodule with a Xilinx Spartan-6 family FPGA.

*Ethernet ist eine ausgereifte Kommunikationstechnologie mit vielen Vorteilen für Sensornetzwerke und einer Vielzahl anderer Einsatzbereiche. Anwendungen interagieren mit einer sogenannten Media-Access-Control-(MAC)-Unterschicht, um Pakete vom Netzwerk zu empfangen und zu versenden. Diese Arbeit stellt den Entwurf, die Implementierung und den Test einer in VHDL umgesetzten MAC-Schicht für Ethernet auf FPGAs vor. Die Kommunikation mit einem integrierten Schaltkreis, der die Bitübertragungsschicht von Ethernet umsetzt, erfolgt über das standardisierte Media-Independent Interface (MII) in den Varianten MII und GMII. Dabei werden Verbindungsgeschwindigkeiten von 10, 100 und 1000 Mb/s im Vollduplex-Modus unterstützt. Im Gegensatz zu bisherigen Lösungen liegt der Schwerpunkt des Entwurfs auf Einfachheit sowohl in der externen Benutzung als auch den internen Abläufen. Benutzern kommt die unkomplizierte FIFO-Schnittstelle zugute, mit der sie gleichzeitig bei nahezu voller Gigabit-Ethernet-Geschwindigkeit senden und empfangen können. Die MAC-Schicht wurde für ein GigaBee-Mikromodul der Trenz Electronic GmbH mit einem FPGA der Spartan-6-Familie von Xilinx entworfen und ebenfalls damit geprüft.*

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **CRC** | **c**yclic **r**edundancy **c**heck |
| **CSMA/CD** | **c**arrier **s**ense **m**ultiple **a**ccess with **c**ollission **d**etection |
| **DCM** | **d**igital **c**lock **m**anager |
| **DDR** | **d**ouble **d**ata **r**ate |
| **DRAM** | **d**ynamic **r**andom **a**ccess **m**emory |
| **FCS** | **f**rame **c**heck **s**equence |
| **FF** | **f**lip-**f**lop |
| **FIFO** | **f**irst-**i**n-**f**irst-**o**ut |
| **FPGA** | **f**ield **p**rogrammable **g**ate **a**rray |
| **FWFT** | **f**irst-**w**ord **f**all-**t**hrough |
| **GMII** | **g**igabit **m**edia-**i**ndependent **i**nterface |
| **HDL** | **h**ardware **d**escription **l**anguage |
| **HTTP** | **H**yper**t**ext **T**ransfer **P**rotocol |
| **IC** | **i**ntegrated **c**ircuit |
| **IEEE** | **I**nstitute of **E**lectrical and **E**lectronics **E**ngineers |
| **IO, I/O** | **i**nput/**o**utput |
| **IOB** | **i**nput/**o**utput **b**lock |
| **IP** | **i**ntellectual **p**roperty |
| **IP** | **I**nternet **P**rotocol |
| **IPG** | **i**nterpacket **g**ap |
| **ISO** | **I**nternational **O**rganization for **S**tandardization |
| **LAN** | **l**ocal **a**rea **n**etwork |
| **LLC** | **l**ogical **l**ink **c**ontrol |
| **LSB** | **l**east **s**ignificant **b**yte |
| **LUT** | **l**ook-**u**p **t**able |

| | |
|---|---|
| **MAC** | **m**edia **a**ccess **c**ontrol |
| **MAN** | **m**etropolitan **a**rea **n**etwork |
| **MII** | **m**edia-**i**ndependent **i**nterface |
| **MIIM** | **m**edia-**i**ndependent **i**nterface **m**anagement |
| **MSB** | **m**ost **s**ignificant **b**yte |
| **OSI** | **O**pen **S**ystems **I**nterconnection |
| **PHY** | **phy**sical (layer) |
| **PLD** | **p**rogrammable **l**ogic **d**evice |
| **PLL** | **p**hase-**l**ocked **l**oop |
| **RAM** | **r**andom **a**ccess **m**emory |
| **RGMII** | **r**educed **g**igabit **m**edia-**i**ndependent **i**nterface |
| **RS** | **r**econciliation **s**ublayer |
| **RTBI** | **r**educed pin count **t**en-**b**it **i**nterface |
| **RX** | **r**eceive, **r**eception |
| **SDR** | **s**ingle **d**ata **r**ate |
| **SGMII** | **s**erial **g**igabit **m**edia-**i**ndependent **i**nterface |
| **TBI** | **t**en-**b**it **i**nterface |
| **TCP** | **T**ransmission **C**ontrol **P**rotocol |
| **TX** | **t**ransmit, **t**ransmission |
| **UDP** | **U**ser **D**atagram **P**rotocol |
| **USB** | **U**niversal **S**erial **B**us |
| **VHDL** | *see VHSIC HDL* |
| **VHSIC HDL** | **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit **H**ardware **D**escription **L**anguage |

# Chapter 1

# Introduction

Modern electronic sensors are very sophisticated pieces of equipment that can, among other things, improve the safety and security of human beings. The Electronic Measurement Engineering Group at Technische Universität Ilmenau has been researching the use of a not yet widely used radio sensor technology called ultra-wideband for a wide range of applications such as breast cancer detection [1], buried weapon detection [2], or vitality monitoring of senior citizens [3], with promising results.

In the latter use case described by Sachs and Herrmann, a network of ultra-wideband radar sensors monitors apartments of elderly people and aims to automatically detect when they e.g. fall over or exhibit anomalous breathing behavior. In contrast to previous solutions, the system provides data continuously and can operate without interaction with the subject, greatly improving practical usability in many dangerous situations.

The measurements from multiple sensors are fed into a central assistance system running on a personal computer that combines and interprets the individual sensor values, necessitating a feasible method of data exchange. The Electronic Measurement Engineering Group currently uses universal serial bus (USB) version 2.0 technology for this purpose, but has encountered considerable limitations with this approach. Most importantly, the maximum cable length (without signal repeaters) of 5 m [4] is impractical for devices distributed throughout e.g. an apartment and even though the standard allows for a maximum of 127 devices on one bus, performance problems were encountered when connecting several sensors.

It is desirable to replace USB with another standard connection technology that does not exhibit these problems, scales better to a higher number of sensor devices, and, at the same time, does not lower other factors such as the data rate below what is achievable with USB 2.0. The well-known Ethernet technologies for local area networks [5] provide for all of that. To be specific, they allow for cable lengths of up to 100 m at a data rate of 1000 Mb/s, even higher than the 480 Mb/s signaling rate of USB 2.0. Ethernet is also a very mature technology, making related equipment available at low prices. As more than 95 percent of all wired local area networks already use it [6], measuring devices can be easily integrated there. The downside of this is that the implementation of standard protocols running on top of Ethernet such as the Internet Protocol is quite complex. Contrary to USB where integrated circuits providing the whole protocol stack such as the popular FX2 family by Cypress Semiconductor Corp. [7] exist, similar products for Ethernet are rare.

The preprocessing of sensor data is currently performed by field-programmable gate arrays (FPGAs), a special type of programmable circuits, on commercial off-the-shelf circuit boards. To not incur any additional cost, the FPGAs should take care of as much of the Ethernet communication process as possible. They can indeed handle at least part of it by addressing nodes on the network and encapsulating data into transmission units (packets). The other part consists of processing and generating the physical signals on the cable. Most FPGAs do not have the capability to perform this function, but special integrated circuits exist for this purpose. Communication with this circuit and the aforementioned packet encapsulation is the responsibility of a media access control (MAC) layer.

The goal of the present thesis is to design, implement, and test an Ethernet MAC on an FPGA that can then be used in ultra-wideband and other sensor devices to communicate with a central data capture and processing entity. The TE0600 FPGA board manufactured by Trenz Electronic GmbH [8] is used as primary target platform for implementation and verification purposes. It already includes an Ethernet physical layer chip. To maximize the suitability of the design for the UWB sensor network, it should:

- be easy to use, i.e. have a generic and familiar interface,
- be easy to understand and maintain (this also requires the source code to be readily available),
- be available free of charge,
- support the most common Ethernet link speeds of 10, 100, and 1000 Mb/s,
- be able to receive and transmit data at a rate very close to the maximum allowed by the Ethernet standard,
- communicate with a physical layer integrated circuit (IC) using the standard media-independent interface, and
- work on the TE0600 board, but also be designed with low-effort portability to other boards and FPGAs in mind.

Within the goals outlined above, portability to other FPGAs is considered secondary.

In contrast, objectives that are not the focus of the present thesis include:

- minimizing the FPGA resources used,
- supporting all possible Ethernet features such as flow control, multicast reception, and half-duplex connections, and
- implementing protocols on top of Ethernet.

For developing hardware structures for programmable logic devices, it needs to be possible to describe the functionality of digital electronic circuits using a formalized language. Every language that is primarily intended not for software development but this purpose is called a hardware description language (HDL). The two main competitors in this field are Verilog-HDL and Very High Speed Integrated Circuits Hardware Description Language (VHSIC HDL or VHDL) [9]. Only VHDL revision VHDL-93[1] as standardized in [10] will be considered for this MAC implementation. An explanation of the language is outside the scope of this thesis, so the reader is assumed to be familiar with at least the basic concepts of VHDL. A practically oriented introduction to the language offers Kafig, for instance, in [11], while a thorough discussion of all aspects can be found in "The Designer's Guide to VHDL" [12] by Ashenden.

In the present thesis, Chapter 2 will first of all introduce the primary technologies required, Ethernet and FPGAs, in more detail. We will then discuss prior implementations of Ethernet MACs on FPGAs and study the target hardware in Chapter 3, concluding the preliminary considerations. The next topic in Chapter 4 is the design of the MAC including an examination of the trade-offs that were made, followed by its implementation in Chapter 5. There, attention is drawn to a few points of special relevance to the core and the goals of this thesis. Chapter 6 shows how and with what results the functionality was tested before the presentation of the MAC sublayer ends in Chapter 7 with a resume of what was achieved and where to go from there.

[1]VHDL-93 will be used because tool support for the more modern VHDL-2008, which allows for simpler constructs in many cases and would in theory be preferable, is still very limited.

# Chapter 2

# Fundamentals

First, we take a look at the Ethernet technologies for use in local and metropolitan area networks (LANs and MANs). They form the very foundation of the work presented in this thesis. The other building block is programmable hardware, introduced in the section directly following.

## 2.1 Ethernet (IEEE Std 802.3)

The Ethernet family of technologies was standardized by the IEEE as IEEE Std 802.3™ for the first time in 1983 [13] describing half-duplex communication over copper wire at a maximum data rate of 10 Mb/s. Since then, a lot of additional features have been added, including full-duplex operation and higher data rates of first 100 Mb/s (referred to as Fast Ethernet), then 1000 Mb/s (referred to as Gigabit Ethernet) and most recently up to 100 Gb/s, an astounding increase by four orders of magnitude. This has allowed Ethernet to become the dominant technology used in computer LANs worldwide: more than 95 percent of all wired local area networks use it [6]. The latest revision as of the writing of this thesis is IEEE Std 802.3-2012 [5], published 2012-12-28. All further references to the standard pertain to this revision.

### 2.1.1 Overview

A layered architecture is described in the document which, at a high level of abstraction, differentiates between the physical layer (PHY) and the media access control (MAC) sublayer. The main responsibility of the physical layer is to transmit arbitrary opaque data over a physical medium while the media access control has to take care of encapsulating the payload into valid Ethernet frames, addressing nodes on the network, and error detection. This is intentionally very similar to the lower layers of the ISO Open Systems Interconnection model (OSI model, ISO/IEC 7498-1 [14]) which describes communication as interaction of abstract layers with well-defined responsibilities as shown in Figure 2.1. Layers communicate only with the layers directly above and below them. As long as the interfaces are compatible, the implementation of a layer can be freely exchanged with another one. The Ethernet PHY layer directly corresponds to the physical layer of the OSI model while the MAC sublayer together with the logical link control (LLC) sublayer form the OSI data link layer. The LLC sublayer (shown with dashed border in the figure) is not mandatory and not specified in IEEE Std 802.3.

Within the context of the standard, the operation of the MAC sublayer is always identical no matter what physical layer is used. Conceptually, it can only send and receive single bits, which makes an additional reconciliation sublayer located between the MAC sublayer and the physical layer necessary. Its only purpose is to translate this very generic interface to the media-independent interface (MII) accepted by a given physical layer. It is called media-independent because although it depends on the PHY implementation, usually all specified PHY variants that support the same transmission speed use the same MII. The interface between the physical layer and the physical medium is called media-dependent interface (MDI) accordingly. Throughout this thesis, the acronym PHY will be used not only to refer to the layer but also a device that implements Ethernet physical layer functionality.

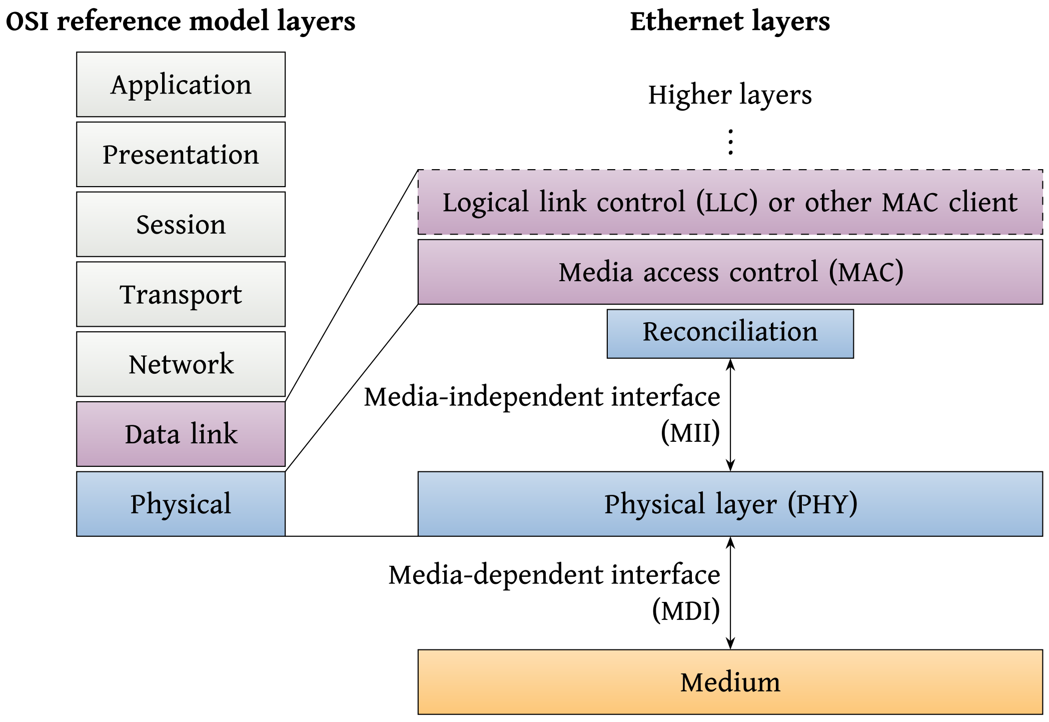

OSI reference model layers
Ethernet layers
Application
Presentation
Session
Transport
Network
Data link
Physical
Higher layers
Logical link control (LLC) or other MAC client
Media access control (MAC)
Reconciliation
Media-independent interface (MII)
Physical layer (PHY)
Media-dependent interface (MDI)
Medium

**Figure 2.1:** IEEE 802.3 simplified layer architecture in relation to the OSI reference model. Adopted from [5, fig. 1-1].

The higher layers are usually comprised of members of the Internet protocol suite: the Internet Protocol (IP) for the network layer, the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP) for the transport layer, and arbitrary session, presentation, and application layer protocols such as the Hypertext Transfer Protocol (HTTP). As this thesis focuses on the data link layer, we will not discuss them further.

Of all physical layer technologies stated in IEEE Std 802.3, the ones most commonly used today are 100BASE-TX with a data rate of 100 Mb/s and 1000BASE-T with a data rate of 1000 Mb/s on copper wires [6]. The 10 Mb/s transmission that has initially spawned the Ethernet standardization is preserved under the name 10BASE-T but is not in wide usage any more. Since 10BASE-T, 100BASE-TX, and 1000BASE-T all share a shielded twisted pair copper wire as defined in the standard TIA/EIA-568-A as their physical medium, it is common for network devices to support multiple standards. A backwards-compatible auto-negotiation procedure is in place that ensures that link partners establish a connection at the fastest possible mode.

Although half-duplex operation with multiple devices on a shared bus was the initially predominant form of Ethernet setups, full-duplex operation has replaced it in most installations. It is safe to assume that Gigabit Ethernet is not practically used in half-duplex mode at all [6]. Using the same line for multiple nodes means that collisions are bound to occur when more than one node tries to access the shared medium at the same point in time. A complicated procedure called carrier sense multiple access with collision detection (CSMA/CD) is used in Ethernet to resolve such situations by detecting that a collision has occurred and retrying the transmission after a random amount of time [6]. In contrast, full-duplex Ethernet mandates that all connections are point-to-point with exactly two devices on the link where collisions are physically impossible. Central switches with multiple ports connect the nodes to form a network and replace the outdated bus topology. Larger commercially available switches usually have around 50 ports but can of course also be connected again to other switches. The benefit of this architecture is vastly improved performance both between two nodes and in the network as a whole. In full-duplex operation, almost the full 1000 Mb/s 1000BASE-T transmission speed can be achieved in each direction between two nodes transmitting to each other simultaneously. Using a single switch port per device also allows a different physical layer and thus data rate to be used for each node [5]. This is crucial as not all network participants may support the same and newest Ethernet technology.

### 2.1.2 Media-independent interface

Since the goal of this thesis is to implement a media access control sublayer for Ethernet, the media-independent interface situated between the MAC and the physical layer is of special importance. If the MAC is separate from the PHY, the latter is usually implemented as an integrated circuit (IC) connected to the MAC unit via electrical wires on a printed circuit board. The electrical, signal timing, and signal functional characteristics of this connection are defined in the MII specifications.

A lot of different MII variants have been proposed both directly in IEEE Std 802.3 and by the telecommunication industry. They differ in their required number of signal lines, supported data rates, and interface clock speeds. An overview of the most important ones is provided in Table 2.1. Link speeds refer to the supported data rates on the physical layer while clock rate designates the maximum speed of the MII clock. Furthermore, double data rate (DDR) indicates transmission of data on both clock edges [15], thereby doubling the interface data rate but imposing stricter timing constraints on both sides. Using only one clock edge, in contrast, is called single data rate (SDR). All MIIs that do not use DDR transmit data on the rising clock edge only.

In order to support all common data rates of 10, 100, and 1000 Mb/s, either a single MII that supports all speeds such as RGMII or SGMII can be used, or switching between different MIIs that in sum can handle all cases such as MII plus GMII is necessary. The latter two interface variants are explained below in more detail.

**Media-independent interface (MII)** (without any prefix) is the oldest of all mentioned variants and was specified for the Fast Ethernet family (100 Mb/s), but also supports operation at 10 Mb/s link speed. It offers full-duplex operation through completely separate data paths for receiving and transmitting data. The data buses called `TXD` and `RXD` are 4 bit wide each and synchronous to the rising edge of their respective clock `TX_CLK` and `RX_CLK`. The detailed timing characteristics are given in clause 22.3 of [5]. Both the RX and TX path clock

**Table 2.1:** Comparison of MII specifications

| Name | # Pins | Link speeds [Mb/s] | Clock rate | DDR | Source |
|---:|---:|:---|---:|:---:|:---|
| MII | 16 | 10, 100 | 25 MHz | No | [5, clause 22] |
| RMII | 8 | 10, 100 | 50 MHz | No | [16] |
| GMII | 24 | 1000 | 125 MHz | No | [5, clause 35] |
| SGMII | 4 | 10, 100, 1000 | 625 MHz | Yes | [17] |
| RGMII | 12 | 10, 100, 1000 | 125 MHz | Yes | [18] |
| TBI | 24 | 1000 | 125 MHz | No | [5, clause 36.3] |

MII: Media-independent Interface, RMII: Reduced media-independent interface, GMII: Gigabit media-independent interface, SGMII: Serial gigabit media-independent interface, RGMII: Reduced gigabit media-independent interface, TBI: Ten-bit interface.

is sourced by the PHY. It has a frequency of 25 MHz at 100 Mb/s operation and 2.5 MHz at 10 Mb/s operation. Additionally, there are control signals for transmission error (TX_ER, RX_ER) and data validity indication (TX_EN, RX_DV) per direction as well as carrier sense and collision detection signals needed for CSMA/CD in half-duplex link modes. Lastly, the out-of-band MII management interface (MIIM) is included for exchanging configuration and status data with the PHY. It is a simple serial interface comprised of two wires with the clock signal MDC always provided by the MAC and the data signal MDIO by either entity as needed. A basic vendor-independent register set that every PHY that claims to support MII has to implement is given in Table 22-6 of the Ethernet standard [5]. It offers functionality for e.g. detecting whether a link has been established, perform a reset, or manually configure the link parameters speed and duplex mode. The current auto-negotiated link speed can be read indirectly by querying the capabilities of the link partner and choosing the fastest supported speed.

**Gigabit media-independent interface (GMII)** is the name of the MII variant for the Gigabit Ethernet family and can be seen mostly as an extension to the original Fast Ethernet MII. The data bus width is doubled to 8 bits and the clock rate quintupled to 125 MHz to account for the tenfold increase in data rate from 100 Mb/s to 1000 Mb/s. This means that timing margins are a lot smaller: the window during which data is guaranteed to be valid at the receiver is shortened from 20 ns with MII to 2 ns per clock cycle. For the TX path, the clock is now called GTX_CLK and provided by the MAC instead of the PHY because wire propagation and processing delays at the MAC would make it difficult to achieve correct timing otherwise. This is called source-synchronous clocking [19]. The interface is otherwise functionally identical for normal data transfer and management. As can be seen in Figure 2.2 which provides an overview of signal connections needed, many pins can be shared in devices supporting both MII and GMII. There are three groups, from top to bottom: transmit direction

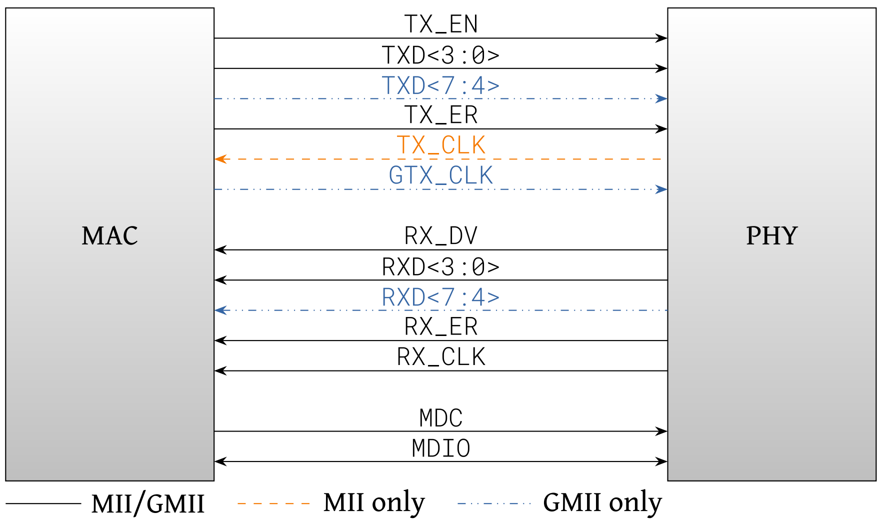


**Figure 2.2:** Signals used by MII and GMII in full-duplex operation. Adopted from [5, fig. 22-3] and [5, fig. 35-2].

signals synchronous to `TX_CLK` (in 10/100 Mb/s mode) or `GTX_CLK` (in 1000 Mb/s mode), receive direction signals synchronous to `RX_CLK`, and the management signal synchronous to `MDC`. Even when the TX data lines are shared between both variants, their differing clock structure complicates the clock distribution and data output architecture in MACs.

### 2.1.3 Packet/frame structure

As we have already discussed, the same conceptual media access control sublayer is used for all Ethernet variants. This also means that the Ethernet packet and MAC frame formats are identical no matter which transmission technology or speed is used, greatly simplifying devices that have to deal with more than one possible combination [6]. In the context of IEEE Std 802.3, a MAC frame contains all control and data information and an Ethernet packet "consists of a MAC frame [...] preceded by the Preamble and the Start Frame Delimiter" [5, clause 1.4.299]. These two added header fields have a combined length of 8 byte in total and their only purpose is synchronization at the physical layer [6]. When transmitting or receiving data, the full Ethernet packet is put on the media-independent interface sequentially. In the following chapters of this thesis, the general term packet (as opposed to Ethernet packet) will be used to denote MAC frames.

Besides the actual client data, the MAC frame contains the destination address, source address, a 2-byte length/type field that contains *either* the length of the frame *or* a MAC client protocol type indication exclusively, optional padding, and the 4-byte frame check sequence (FCS). Every Ethernet network interface controller has its own globally unique address with a length of 6 bytes. The FCS is a checksum calculated over the whole MAC frame minus the FCS itself according to a standard algorithm called cyclic redundancy check (CRC) defined in [5, clause 3.2.9]. Many common transmission errors at the physical layer like single flipped bits or sequences of corrupted bits can be detected with this checksum [20]. A general explanation of CRC calculation can be found e.g. from Halsall [21].

The achievable total performance of Ethernet is limited by the overhead introduced through the preamble and the interpacket gap (IPG) which requires a minimum duration of 96 bit times without activity after every packet transmitted. The maximum size of a normal Ethernet frame is 1518 bytes, but shorter frames are permitted down to a minimum of 64 bytes. When the payload is very short, the frame has to be padded to meet the requirement. As the overhead occurs per packet, data rate is at maximum when a lot of client data is transmitted in each one and at minimum when only short frames are sent. To illustrate the vast difference, we compare the achievable user data rate at 1 Gb/s link speed for both extremes.

The static amount of bytes added by headers and the FCS in every MAC frame $n_{Overhead,frame}$ is

$$\begin{aligned} n_{Overhead,frame} &= 2n_{Address} + n_{Length/type} + n_{FCS} \\ &= 2 \cdot 6 + 2 + 4 \\ &= 18, \end{aligned}$$

where $n_x$ is the number of bytes needed for header element $x$, respectively. The only dynamic length field is the padding which has a byte length of

$$\begin{aligned} n_{Padding} &= \max\{0, n_{Min.framesize} - n_{Overhead,frame} - n_{Data}\} \\ &= \max\{0, 64 - 18 - n_{Data}\} \\ &= \max\{0, 46 - n_{Data}\} \end{aligned}$$

depending on $n_{Data}$. Combined with the Ethernet packet preamble and the IPG they form the total amount of overhead bytes per packet $n_{Overhead}$ as follows:

$$\begin{aligned} n_{Overhead} &= n_{Preamble} + n_{SFD} + n_{Overhead,frame} + n_{Padding} + n_{IPG} \\ &= 7 + 1 + 18 + \max\{0, 46 - n_{Data}\} + 12 \\ &= 38 + \max\{0, 46 - n_{Data}\}. \end{aligned}$$

The achievable user data rate $r(n_{Data})$ in the case of a continuous Gigabit Ethernet transmission of packets with $n_{Data}$ bytes of payload is then

$$r(n_{Data}) = \frac{n_{Data}}{n_{Data} + n_{Overhead}} \cdot 1\,\text{Gb/s} = \frac{n_{Data}}{n_{Data} + 38 + \max\{0, 46 - n_{Data}\}} \cdot 1\,\text{Gb/s}.$$

Substituting one as the minimum number of bytes for $n_{Data}$, we get

$$r(1) = \frac{1}{1 + 38 + 45}\,\text{Gb/s} = \frac{1}{84}\,\text{Gb/s} \approx 11.9\,\text{Mb/s}.$$

In contrast, the maximum number of payload bytes is the maximum frame size of 1518 bytes minus $n_{Overhead,frame}$, resulting in 1500 bytes. The user data rate is then

$$r(1500) = \frac{1500}{1500 + 38}\,\text{Gb/s} = \frac{1500}{1538}\,\text{Gb/s} \approx 975.3\,\text{Mb/s}.$$

The difference is thus about two orders of magnitude, but the maximum rate achievable by using big packets is near the physical layer rate of 1000 Mb/s.

## 2.2 Hardware design with FPGAs

After introducing the networking technology that has to be supported, we can now continue with a look at the general type of hardware on which the implementation takes place. A field programmable gate array (FPGA) is the most complex variant of a programmable logic device (PLD) and "contain[s] digital logic cells and programmable interconnect" [9]. The logic elements can be connected together to form a circuit that realizes arbitrary functions when programmed accordingly. This is conceptually similar to a microprocessor which can also perform a wide range of functionality but is limited to sequential execution of a piece of software and a specific set of instructions that can be processed. In contrast, an FPGA operates at a much lower abstraction level with inherent parallelism, leading to better performance. This also means that developing designs is vastly different from software programming and in many cases more complicated. In fact, microprocessors themselves can be implemented on FPGAs and it is common to do so in order to combine the benefits of both concepts [22]. The general components of an FPGA are outlined below, but if more information about the general design process is desired, [23] will offer a practical introduction.

### 2.2.1 Components

The basic programmable logic functionality of FPGAs is realized not by simple boolean logic gates, but by a number of look-up tables (LUTs) and flip-flops contained in an array of slices. LUTs have multiple inputs and outputs and are dynamically programmed with the output values for every possible combination of input signal levels. Flip-flops are used as synchronous registers for intermediate storage of values. While the general idea is the same, the exact number and type of components in a slice varies between vendors [9]. Maxfield discusses logic slices and their internal operation in more detail in [22].

Interaction with the environment is required in any meaningful design, and an area that FPGAs excel in when compared to simpler PLDs. Larger devices provide a great number of package pins backed up by input/output (I/O) elements. They support different I/O standards and accommodate for high-speed buses with tight timing requirements by e.g. including flip-flops located very closely to the pins or providing special I/O clocking resources. A closer look at the I/O elements of the target device family of this thesis follows in Section 3.2.

As in most digital circuits, processing is done synchronously to a clock in many components of an FPGA. There is usually at least one system clock fed to the device that is used for general operation of the circuit and as a reference for synthesizing other frequencies if needed [9]. Additionally, I/O interfaces may require separate clocks if they do not use the same system clock as the FPGA does. Every clock signal spans a clock domain that encompasses all logic elements that use it as their clock [19]. Those signals can usually only be routed on special clocking interconnect that is separate from the connections used for standard logic. It offers fast signal propagation with very low skew throughout the whole device or parts of it, but only reaches dedicated clock pins such as the clock inputs of D-type flip-flops.

To filter clocks that may contain jitter and derive new clocks with selectable phase shifts and frequency ratios, FPGAs include special digital and analog clocking components. A very common structure for analog frequency synthesis is the phase-locked loop (PLL) that can generate a wide range of output frequencies with defined phase relationships from a single clock input. The digital clocking resources, which fulfill a function similar to the PLLs, vary in their implementation, the FPGA vendor Xilinx for instance calls their equivalent digital clock managers (DCMs) in the Spartan device family [24].

FPGAs usually also include blocks of RAM to store larger amounts of data efficiently without using flip-flops. Additionally, digital signal processing blocks, memory controllers, and other circuitry realizing complex functions may be available [9].

A single aspect that typically does not warrant much attention when programming software but is of special importance in FPGA designs that perform high-speed interaction with external entities is the correct use of clock signals, which we will discuss next.

### 2.2.2 Clocking and metastability

When using multiple clocks, passing signals between logic of different clock domains is a very complicated matter because if it is done wrong, non-repeatable failure conditions can occur that are very difficult to debug.

The root of this kind of problems lies in metastability, a physical phenomenon exhibited by a great number of digital devices. The D-type flip-flop for example samples its input pin `D` every time a rising edge is detected on the clock input `CLK` and then puts the value on its output pin `Q`. For correct behavior, it is essential that the `D` input stays stable for a minimum amount of time before the clock edge arrives (the setup time $T_{setup}$) and after the clock edge has arrived (the hold time $T_{hold}$) [25]. If these timing constraints are violated as shown in

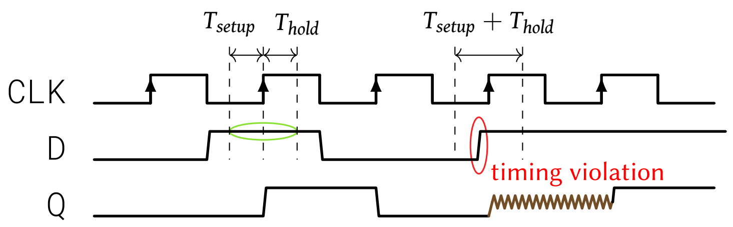


**Figure 2.3:** Metastability in a D-type flip-flop. General idea adopted from [25, fig. 16.8].

Figure 2.3, the flip-flop may enter a metastable state, which means that the output lingers somewhere in between the voltage levels of a valid digital 0 and a valid digital 1 or even oscillates as discussed by Kleeman and Cantoni [26].

Chu claims in [25] that other gates and components that use the flip-flop output in some way can then interpret the output as either a logic 0 or 1 independently from each other, bringing the system to an invalid state from which it may not be able to recover. The metastable state can continue theoretically indefinitely, but it has been shown that the probability of the flip-flop resolving to a stable state increases exponentially over time. If there is no defined phase relation between clocks of different clock domains, it is impossible to correctly sample a signal from another domain without possibly violating the setup or hold time requirements of the D-FF in the target domain. Or as Kleeman et al. put it: “it is generally accepted that a perfect synchronizer cannot be physically realized.” [26]

As the phenomenon cannot be prevented from occurring and will resolve by itself over time, the only direct solution is to give the FF enough time so that the probability of failure becomes small enough. The common way to do this is to add two (or more) chained flip-flops clocked in the target domain with the input of the first one connected to the signal to synchronize and the output of the last one used as synchronized signal. This structure protects the following combinatorial logic from intermittent invalid output states and gives the first flip-flop sufficient time to resolve to a valid state should timing requirements be violated. This simple approach introduces a delay of at least two clock cycles and can only be used for single independent signals as it can not be guaranteed that all individually synchronized signals become valid in the same clock cycle.

### 2.2.3 First-in-first-out buffers

To effectively transfer data across clock boundaries, more sophisticated schemes have to be employed. A very straightforward and efficient option is to use an asynchronous first-in-first-out (FIFO) buffer. Data can be written into a FIFO and then read out only in the exact order it was written [25]. Being asynchronous means that there is a read and and a write side and that they can be used independently, concurrently, and have individual clocks. This makes the FIFO buffer an ideal candidate for cross-domain data transfer. Buffer empty and full indications can be provided for flow control.

Implementing this behavior correctly on hardware is rather complicated as can be seen in an exemplary design by Chu [25, pp. 652-660], but it is rarely necessary to do so. FPGA vendors provide automated generators tailored to their products with their design suites, usually free of charge. Xilinx for instance provides the “LogiCORE IP FIFO Generator” [27] that can generate FIFOs of almost arbitrary size and data bus width with or without separate clock domains for reading and writing. Optionally, counters that indicate the amount of data available for reading or writing can also be included. When both sides of the FIFO are clocked with the same clock speed and a continuous stream of data is written into it, the buffer can also be read continuously and at full speed after an initial delay of a few clock cycles depending on the implementation. A common variant called first-word fall-through (FWFT) or show-ahead [28] mode allows looking ahead to the next available data unit without explicitly reading it: the data, so to speak, "falls through" from the write to the read side.

# Chapter 3

# State of the Art

We have seen the fundamental technologies used in this thesis and what we must consider when using them. Next, it is essential to introduce existing solutions to our problem and how they relate to our goals. After that, studying the target platform and its specific FPGA will help in understanding the following chapters.

## 3.1 Existing Ethernet MAC intellectual property cores

As the Ethernet technology is more than 20 years old, it is no surprise that a lot of MAC sublayers are already implemented in a wide range of technologies including FPGAs. When such a design is generic enough to be useful to a number of people, it is called an intellectual property core (IP core) that can be shared or sold. As many companies feel uncomfortable distributing the HDL code which shows off all their raw engineering ingenuity to potential competitors, a functionally equivalent but opaque (to humans, at the very least) list of gates and their connections can be offered instead.

Commercial Ethernet cores are available either directly from FPGA vendors such as Xilinx [29] and Altera [30] or third-party IP vendors such as Synopsys [31], mobiveil [32], and morethanIP [33]. They are available at least in a variant supporting 10/100/1000 speed operation, very feature-rich, and verified in tests and practice. However, they each come with one or more of the following downsides:

- The HDL source code is not available for inspection and modification, only an opaque netlist.
- Although the actual prices are usually not made public, high costs pose a problem especially for small or non-commercial projects including usage in research institutions.

- The interface offered to the application is tailored for integration into larger on-chip buses or microprocessors which makes it unnecessarily complex for simple applications.
- More features than needed are provided, increasing the complexity and resource usage of the FPGA design.
- Code that is specific to an FPGA family such as MII input/output meeting the timing required by the Ethernet standard and clocking is not included and would need to be developed by the user.
- A function to determine the current link speed automatically by reading MII management registers is missing.

At least the cost and intransparency arguments do not apply to cores from the Internet community platform OpenCores which aims to provide IP cores with full source code under open-source licenses. There are a few MAC implementations available, but only one that supports flexible operation at 10, 100, and 1000 Mb/s speed called ethernet_tri_mode [34]. It is written in Verilog and as a consequence may not be ideal for integrating into VHDL-based designs. Although it is possible to interface Verilog with VHDL code and vice-versa, keeping all components in the same language makes the source more consistent and thus easier to understand.

As can be seen in the official documentation [35], the user interface is documented only by two vague timing diagrams, one for transmission and one for reception, without any explanation of the signals involved. The fact that the user must resort to guesswork makes the core difficult to use. The RX part is included here without modification in Figure 3.1,

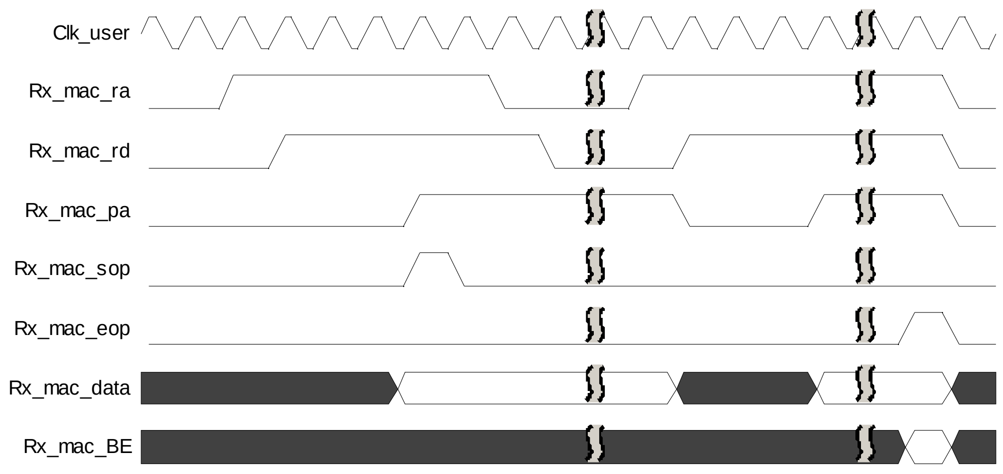


**Figure 3.1:** Timing diagram of the OpenCores ethernet_tri_mode IP core RX user interface. Source: [35]

the TX case is similar. The data bus `Rx_mac_data` is 32 bits wide. It is difficult to ascertain the meaning of the signals, but `Rx_mac_sop` is assumed to be the start of packet indicator while `Rx_mac_eop` looks like the end of packet indicator. They provide the actual length of the data only indirectly; the user needs to count how many bytes were transferred between assertion of the indicators if the size information is needed for further processing, which is very often the case. A look into the source code reveals that `Rx_mac_BE` serves as indicator for byte validity in the last 4-byte unit of packet data. As packets can have lengths that are not multiples of 4 bytes, the user has to put some effort into special handling of the last data unit. These considerations already show that the interface is more complex than it needs to be for general-purpose applications.

Another MAC implementation in the recent journal article “Design and Implementation of an Ethernet MAC IP Core for Embedded Applications” [36] has a similar aim of providing a flexible core for various applications. Because the full text could not be procured, only the abstract is considered here. It shows that the primary goal was to reduce FPGA resource usage and power consumption, which is deliberately not the case in the present thesis. The focus on ease of use and understanding here is a clear distinction from the work demonstrated in the article.

In summary, a sizable number of prior implementations exist but we have seen none that meet the goals of this thesis, especially being easy to use and understand and having the source code available free of charge.

## 3.2 Hardware platform

Section 2.2 provided a general introduction to field-programmable gate arrays that is now expanded upon by introducing the specific device and environment used during development of the MAC sublayer implementation presented in this thesis. The primary target platform is a GigaBee XC6SLX series industrial-grade FPGA micromodule manufactured by Trenz Electronic GmbH [8]. It includes a Xilinx Spartan-6 LX family FPGA at a system clock rate of 125 MHz and a 10/100/1000 tri-speed ethernet PHY chip Marvell Alaska 88E1111 [38] which supports a very wide range of interfaces: MII/GMII, RGMII, TBI, reduced pin count TBI (RTBI), and SGMII [39]. Which of those is used can be configured through the MII management interface. The smallest variant which is also the one used during development has a Xilinx XC6SLX45-2 FPGA and two 128 MiB dynamic random access memory (DRAM) ICs for data storage. As the micromodule cannot be used stand-alone, the optional baseboard TE0603 provides power supply circuitry and a standard Ethernet connector for testing purposes [40]. It can be replaced by a custom printed circuit board in actual applications. Figure 3.2 shows the two boards mated together.

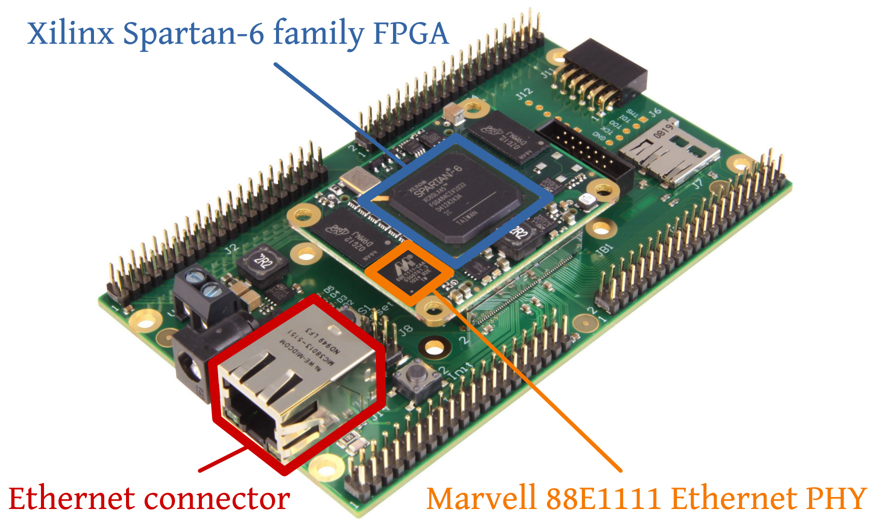


**Figure 3.2:** A Trenz Electronic TE0603 baseboard with a GigaBee XC6SLX series module mounted on top. Source: [37] (component markings added).

The XC6SLX45 offers a total amount of 43,661 logic cells, 2,088 kB of block RAM storage, 8 digital clock managers (DCMs), and 4 phase locked loops (PLLs) [41]. All Spartan-6 family FPGAs have 16 global clock buffers called BUFGMUX that must be used to get clock signals onto the clocking interconnect, each of which can switch between two input clocks by way of a third select input. The switch can be performed free of glitches provided both clocks are continuously running [24].

Furthermore, there are flip-flop elements located directly besides every device pin in their designated input/output blocks (IOBs) to provide outputs with low skew called OLOGIC2 and capture inputs directly at their source called ILOGIC2 [42]. When combined with the possibility to also capture a clock right at the pin in a BUFIO2 element and use it to clock the FFs of neighboring data pins, input from high-speed data buses that supply their own clock signal can be easily implemented, eliminating delay and skew issues when the clock is first routed to a global buffer and then to a FF located arbitrarily in the device. The configurable delay element IODELAY2 again built into every IOB can be used to delay the clock, the data signal, or both in order to place the rising edge of the clock right in the middle of the valid data window. All I/O registers additionally provide double data rate ability. Buffers supporting a wide range of digital I/O standards are available. They drive either the general (input buffer IBUF, output buffer OBUF) or the clocking interconnect (input buffer IBUFG). There is no dedicated clock output buffer.

Up to this point, only preexisting technologies and products were introduced. Subsequently, this information is put to good use by moving on to the design and implementation of the FPGA-based Ethernet MAC sublayer.

# Chapter 4

# Design

The first step in developing the Ethernet MAC was to partition the task into smaller components (VHDL entities) with distinct responsibilities and define their interconnection. Special attention is drawn to a few key points that we will discuss first before we take a look at the resulting design in Section 4.2. A more thorough look at each component follows in Section 4.3.

## 4.1 Design trade-offs

The interface offered to the user is of primary concern: to reach the goals of this thesis, it has to be fast enough to allow for symmetrical 1 Gb/s data transfer but also be easy to use, i.e. use simple signals and a common pattern that hardware developers are inherently familiar with. Furthermore, the users should be able to freely choose the clock they want to use for communicating with the MAC to avoid implementing clock domain crossing - the MAC should do that for them. A standard on-chip system bus like the Advanced eXtensible Interface (AXI) by ARM Limited [43] or Wishbone by OpenCores [44] is unfeasible because even though the concept is certainly well-known, it is over-sized for simple applications both in terms of signal count and complexity. Already less complex but similar in the overall idea is a block of RAM which acts as a buffer. To transmit a packet, the user could write it into the memory, but then the following questions have to be considered: how does the transmission logic know where the next packet lies in the buffer? How does it know that a packet is ready to send? How does the user know which packets in the buffer were sent and where he or she can write the next bulk of data? These are questions that can be answered, but in the course of doing so the naively suggested RAM interface will inevitably become more complex. If we think back to Section 2.2.3 though, we remember the FIFO interface and its characteristic of being very simple. It is also well-known, allows for high-speed data transfer and, when backed by a standard implementation, offers easy clock domain crossing, making it ideal for the MAC sublayer presented in this thesis.

When describing an interface, not only the connections, but also the width and format of the data that is passed over them must be clear. To be able to construct a valid MAC frame, at least the destination address, source address, length/type, and client data is needed as we have seen in Section 2.1.3. The preamble and start frame delimiter can be skipped as they do not carry any information anyway. Theoretically speaking, these fields can be encoded in any fashion and order. But for both the user and the implementation of the media access control sublayer, keeping it plain and sticking to the format already defined in the Ethernet standard is the easiest option. There is also nothing to be gained by being fancy about the frame data representation. By sticking to the standard, the MAC can pass through most of the frame without any modification.

There is one complication concerning the frame trailer though: on the one hand, the user should not have to include the padding and frame check sequence on transmission. This problem can be solved when the data is defined to end right before padding and the MAC takes care of adding it. On reception on the other hand, the original length of the frame is not necessarily part of the header and thus unknown, so the padding cannot be removed in the general case. The FCS is a fixed-length field at the end of the packet and thus can always be stripped. Doing so makes the interface as symmetrical as possible. The user then has no possibility of verifying the integrity of a received packet, but that is also not necessary when the MAC takes proper care that erroneous packets are internally consumed and never reach the FIFO interface.

We have yet to examine the question of how many bits to transfer in one clock cycle, but as Ethernet counts data in multiples of bytes, making the interface 8 bit wide is the natural choice. A wider data bus allows for lower interface clock rates at identical data rates, but necessitates adaption logic to guarantee that byte units can be received and transmitted. In a general-purpose MAC, it is impossible to know the exact requirements of the user and what width would benefit his or her application, so the design has to cover all possibilities with the least common denominator of 1 byte. Width conversion logic can be easily added on top of the MAC FIFO interface by the user if there is a need for it.

A problem that still lingers in the FIFO architecture is that if only the data of packets is passed on, there is no way to know where they start and end. This is fixed by placing two bytes that indicate the size of the next packet in the buffer before each data part. Ethernet frames can be up to 1514 bytes long (excluding the frame check sequence), so one byte (maximum value 255) would not be enough for storing their size; the next possibility is using two bytes (maximum value 65535). As $\lceil \log_2 1514 \rceil = 11$ bits are strictly needed for storing the maximally possible frame size, 5 of the 16 bits will inevitably remain unused but the wasted space is so small that it is of no concern. It accounts for between 0.04 and 1.02 percent of the total stored data, depending on the packet size.

Nevertheless, we can see that there are in fact two limitations with this simple approach:

1. Packets must be delayed in the FIFO until they have been completely received so their size is known and can be put in front of the data.
2. The user needs to know the sizes of packets prior to starting transmission.

Both are rarely relevant in practice: a packet can usually not be acted upon as it is still being received since it could contain errors that are only detected when the frame check sequence that follows the data payload is verified. It is very important to understand that only the latency is different but that the achievable RX data rate is identical both when delaying the packet and when handing it to the user immediately. As for the second restriction, most users will want to use protocols that include further header checksums like UDP/IP instead of raw Ethernet frames. The IP header for example includes a field for the length of the packet, so to calculate its checksum the packet size again has to be known beforehand no matter how the MAC sublayer is designed.

One of the other choices to be made is whether to have completely separate modules in the transmit and receive paths or to combine some or all of them. Either option is feasible as there is zero interaction between both directions. However, the functionality is very similar, symmetrical even, so the code will be easier to understand if it is kept in one place. Splitting it per direction serves little purpose besides artificially increasing the component count, complicating the design. The single exception where it is worthwhile to make a separation are the FIFOs. The user might want to replace only one direction with a custom implementation.

These were the key points that we need to take into account when studying the MAC design in the following section.

## 4.2 Overview

Figure 4.1 shows an overview of the MAC design. Every block corresponds to a VHDL entity with the indicated name. The arrows indicate conceptual flow of data and do *not* imply VHDL port directions. We shortly discuss the diagram beginning at the interface to the user application on the left-hand side and continuing rightwards from there until reaching the connection to the physical layer on the right-hand side.

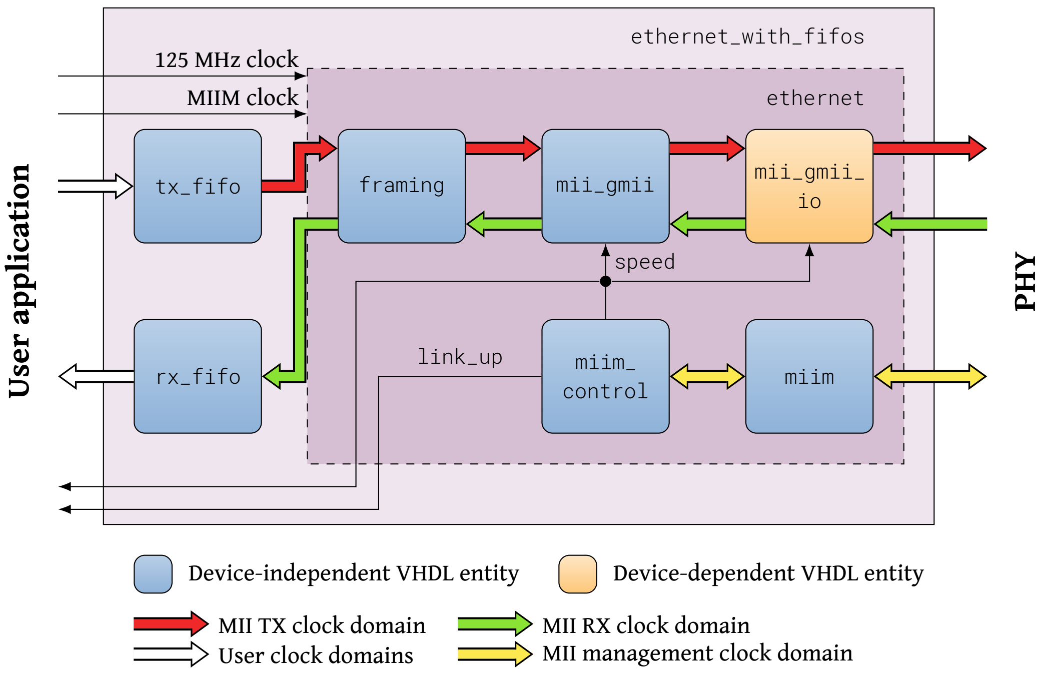


**Figure 4.1:** Block diagram of the Ethernet MAC presented in this thesis

To not require users to learn the details of the MAC implementation and buffer packets themselves, FIFOs with standard interfaces are provided for both the transmit and receive data paths as previously discussed. As can be seen in the diagram, the two directions are completely independent from each other throughout the whole design. The first entity that implements actual Media Access Control sublayer functionality is the `framing` component. Functionality-wise, it performs the core operations that are expected from an Ethernet MAC like generating and verifying checksums and packet encapsulation. The following `mii_gmii` module acts as transparent converter between the generic streams of packet data and the actual signals expected by the media-independent interfaces as well as switching between MII and GMII as needed based on the current link speed. The speed is periodically read out of the MII management interface by `miim_control` which also takes care of the initial configuration of the PHY after reset. Only the high-level functionality is allotted to this module while the actual MIIM interface transactions are performed by `miim`. Lastly, due to the narrow timing requirements of the MII/GMII connections, it is not possible to directly connect them to the FPGA package pins. The device-specific I/O configuration needed in between is provided by `mii_gmii_io`.

Besides the clock pins used by the MII, the MAC requires the user to supply a 125 MHz reference clock to use as transmission clock in GMII mode, a clock for the MII management interface, and two FIFO clocks, one for each direction. The latter three of these clocks can be identical to simplify the user design.

The `reset_generator` entity (not shown in the figure) monitors the current link speed and issues a complete reset for the MII RX and TX clock domains when it changes. This guarantees a consistent system state after a potential transmission clock switch between MII and GMII. The reset indication is also provided to the user to ensure that no mismatched data is written to or read from the FIFOs. We will discuss the exact reasons why this procedure is necessary later when describing the IO implementation in Section 5.1.1.

As a convenience for using the MAC core, all mentioned components beginning at `framing` are preconnected to each other in a structural entity called `ethernet` which is then combined again with the FIFOs to form `ethernet_with_fifos`. The latter is a ready-to-use entity offering the FIFOs' interfaces on the user side and direct MII connections on the PHY side. It can be instantiated in custom designs that want to use Ethernet without much effort. It generally does not provide any benefit to bypass the FIFOs and directly connect to the framing component except in sufficiently sophisticated applications.

To clarify how the components relate to the layered architecture defined in the Ethernet standard as introduced in Section 2.1.1, Figure 4.2 shows the Ethernet layers and selected entities of the MAC design presented in this thesis side-by-side. The higher layers on the right-hand side are only exemplary and can be exchanged for arbitrary user application layers. Although the purposes of the reconciliation sublayer (RS) and the original Ethernet MAC sublayer correspond to the `mii_gmii` and `framing` entities respectively, the interfaces between them differ. The standard e.g. suggests that the RS and the MAC sublayer communicate data one bit at a time, which is impractical for actual implementation on an FPGA. Similar considerations apply to the interface the MAC offers to higher layers. The logical link control (LLC) sublayer is designated optional by the standard and must be implemented by the user if needed.

## 4.3 Components

Let us now take a closer look at each component and the input and output connections it exposes to its neighbors.

### 4.3.1 FIFOs

The only modules that the user will normally interact with directly are the transmit and the receive FIFO buffer. Consistent with our prior considerations, both have a standard interface as seen for instance with Chu [25, p. 280] with a data width of 8 bits and only one side facing the user. The TX FIFO can be written, the RX FIFO can be read by the user. They both have

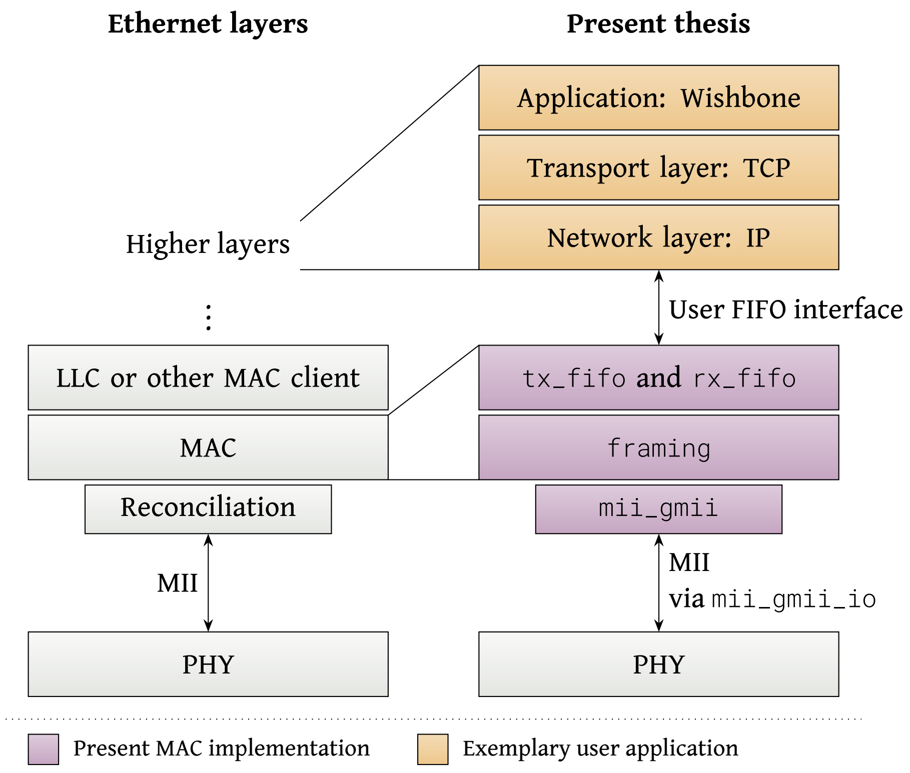


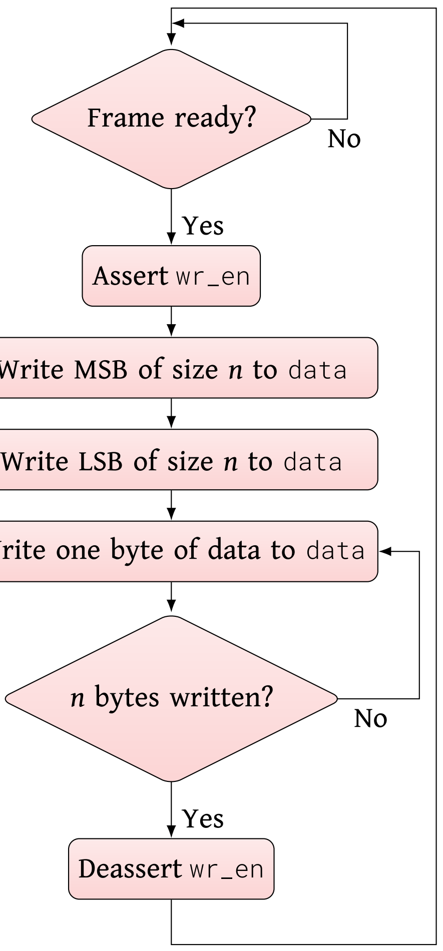
**Figure 4.2:** Ethernet simplified layer architecture in relation to the MAC implementation presented in this thesis. Ethernet side adopted from [5, fig. 1-1].

their own clock domain with the clock supplied by the user. The timing characteristics conform to the interface of first-word fall-through FIFOs generated by the Xilinx LogiCore IP FIFO generator [27].

To send a packet, first of all the FIFO must not be full. The application must put the 2-byte size of the packet into the buffer, most significant byte (MSB) first, then the least significant byte (LSB), after which precisely as many bytes of data as were indicated must follow. Keeping the structure identical to the Ethernet standard avoids unnecessary and costly rearrangements within the Ethernet packet when sending. As soon as the FIFO detects that the data for the next packet has been written completely, it starts passing the data on to the framing module.

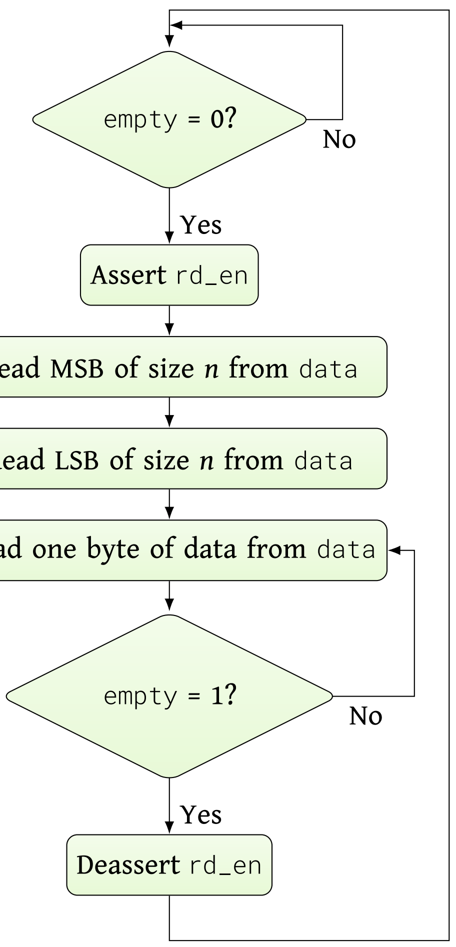
Receiving a packet works analogously: `framing` writes an incoming packet to the buffer. After it has been received completely and without errors, the RX FIFO will indicate that it is not empty and the size and data can be read. As an added convenience, the FIFO will pretend to be empty for a few clock cycles after every packet even when technically more packets are available, so the user can sense the end of a transmission unit and does not need to count the data bytes while reading. During ongoing read-out of a packet, the FIFO never becomes empty. Although the interface is identical to that of a normal FIFO and the entity is called `rx_fifo`, it cannot work like one internally. The packet size is known and written to the memory only after the last data byte has been received, but needs to be read out first, before the data, in clear violation of the first-in-first-out principle. Also, erroneous frames must be skipped without requiring the user to read them out.

Figure 4.3 shows basic flowcharts for both transmission and reception of packets using the FIFO. The signal name `wr_en` means "write enable" while `rd_en` stands for "read enable". In the actual entities, the port names are additionally prefixed by the direction (`tx_` or `rx_`). Although it is not shown for the TX case (Figure 4.3a) because it would considerably clutter the diagram, each write step can only occur when `tx_full_o` is not asserted. If it is, the user has to deassert `tx_wr_en_i` and delay the byte until the FIFO has space available again, or the write will be lost.



Frame ready?
No
Yes
Assert wr_en
Write MSB of size $n$ to data
Write LSB of size $n$ to data
Write one byte of data to data
$n$ bytes written?
No
Yes
Deassert wr_en

**(a)** Transmission



empty = 0?
No
Yes
Assert rd_en
Read MSB of size $n$ from data
Read LSB of size $n$ from data
Read one byte of data from data
empty = 1?
No
Yes
Deassert rd_en

**(b)** Reception

**Figure 4.3:** Flowcharts for packet transmission and reception on the MAC's FIFO user interface

### 4.3.2 Framing

As the core part implementing actual MAC sublayer functionality, the `framing` component has the following responsibilities:

- **Encapsulation** of packets on transmission:
  - Insertion of the preamble and start frame delimiter before transmitting the data
  - Insertion of padding as required
  - Calculation and insertion of the frame check sequence
  - Enforcement of the interpacket gap
- **Verification and decapsulation** of packets on reception:
  - Verification of the frame start which should be any number of preamble bytes followed by the start frame delimiter
  - Verification of the frame check sequence
  - Verification of the packet length which should be within the limits required by the Ethernet standard

All tasks are fulfilled by forwarding the data stream from the FIFO to `mii_gmii` and vice-versa, carefully inserting and removing data bytes as necessary. The interface offered to the FIFOs is almost identical to the one towards the `mii_gmii` module. Table 4.1 lists all ports of the module. Transfer of a single byte of data can take a different amount of clock cycles depending on the underlying MII variant in use: only 4 bits can be transferred at a time with the Fast Ethernet MII for example, so it will take two cycles to completely send 8 bits. Since `mii_gmii` does not buffer data, a new byte can only be placed on the line after the current one was processed. The next data unit should be placed on `tx_data` in each clock cycle that has `tx_byte_sent` high when transmitting data, and read from `rx_data` in each clock cycle that has `rx_byte_received` high when receiving data. The `tx_byte_sent` and `rx_byte_received` signals effectively act as a data strobe signal. Exemplary timing diagrams for packet transactions are found in Appendix A.

All differences between the two sides of `framing` are related to the interpacket gap (IPG) in frame transmission: `tx_busy` exists only towards the FIFO and `tx_gap` only towards `mii_gmii`. The busy indication is needed for signaling the FIFO to delay sending the next packet during the IPG while `tx_gap` actively requests continued operation of `tx_byte_sent` without actually transmitting data on the media-independent interface.

Verification of incoming packets has two distinct ways in which errors are handled: should the frame beginning be invalid already, the remainder of the frame is silently skipped as no

**Table 4.1:** Ports of the `framing` entity

| Port name | Function |
|---|---|
| `tx_reset_i` | Active-high asynchronous reset of all TX functions |
| `tx_clock_i` | Clock for all transmit signals |
| `tx_enable_i` | Active-high transmit enable |
| `tx_data_i<7:0>` | Data to transmit |
| `tx_byte_sent_o` | Put next byte on `tx_data` when asserted |
| `tx_busy_o` | Start new packet only when deasserted |
| `rx_reset_i` | Active-high asynchronous reset of all RX functions |
| `rx_clock_i` | Clock for all receive signals |
| `rx_frame_o` | Asserted as long as one continuous frame is being received |
| `rx_data_o<7:0>` | Data received |
| `rx_byte_received_o` | Asserted when `rx_data` is valid |
| `rx_error_o` | Active-high receive error indication |
| `mii_tx_enable_o` | Function equivalent to `tx_enable_i` |
| `mii_tx_data_o<7:0>` | Function equivalent to `tx_data_i` |
| `mii_tx_byte_sent_i` | Function equivalent to `tx_byte_sent_o` |
| `mii_tx_gap_o` | Transmit IPG when asserted |
| `mii_rx_frame_i` | Function equivalent to `rx_frame_o` |
| `mii_rx_data_i<7:0>` | Function equivalent to `rx_data_o` |
| `mii_rx_byte_received_i` | Function equivalent to `rx_byte_received_o` |
| `mii_rx_error_i` | Function equivalent to `rx_error_o` |

data has left `framing` yet. If the FCS or length checks fail, a reception error is indicated by `rx_error_o`. In any case, the FCS is not stripped from the packet on reception since doing so would require delaying the data stream by the checksum length of 4 bytes. This would substantially increase the complexity of the receive state machine. Instead, the `rx_fifo` entity is responsible for removing the FCS before it reaches the user.

### 4.3.3 Media-independent interface

As the name `mii_gmii` indicates, we have reached the first entity where actual communication with the physical layer device takes place. Because it primarily serves as a thin signal adaption layer, there is not much to do here, especially when GMII is used. Then the data received from `framing` can be passed on to the MII IO entity by directly mapping the ports. For MII though each data byte must be split into two 4-bit units on transmission and combined on reception. The MII signals `CRS`, `COL`, and `TX_ER` are omitted from the MII/GMII ports because they are only useful in half-duplex modes.

### 4.3.4 Media-independent interface input/output

The outputs and inputs of the `mii` entity on the MII side already conform to the functional characteristics of the Ethernet specification, but their timing has to be adapted. Getting this right is dependent on the concrete FPGA architecture and as a consequence put into a separate exchangeable entity named `mii_gmii_io`. The ports must then finally be connected to the pins of the device by the user where they will reach the PHY.

### 4.3.5 Media-independent interface management

`miim` implements the standard management interface of MII. In addition to the actual MIIM `MDC` and `MDIO` signals for connection to the PHY, an interface to read and write single registers one at a time is provided. It runs in a user-supplied separate MIIM clock domain together with `miim_control`.

### 4.3.6 Media-independent interface management control

For both the IO block and the MII/GMII interface adapter it is critical to know which speed is used on the physical medium because the PHY will only meaningfully process MII signals in 10 or 100 Mb/s operation and GMII signals in 1000 Mb/s operation. `miim_control` leverages the `miim` entity to poll the status and auto-negotiation management registers which show if and at what speed a link is currently established. The speed information is propagated both MAC-internally and to the user. Whether a link has been established at all is irrelevant to the operation of the MAC; the indication is only provided for the user application. MII operation is completely stateless, so it is impossible to bring the interface into an invalid state. As long as the physical layer device has not successfully connected to a link partner, it will ignore transmission requests and not put any received packets on the RX interface.

The MAC sublayer will not support half-duplex connections, yet the PHY still needs to know that it should never try to establish a link at any such mode, which can be done via MIIM. The MIIM control entity thus is also responsible for configuring the physical layer device for full-duplex-only operation after reset.

In this chapter, we discussed the abstract design of the MAC sublayer implementation, what decisions were made for design trade-offs and why. We can now go one step further and consider the actual VHDL implementation on a Xilinx Spartan-6 family FPGA.

# Chapter 5

# Implementation

The task of implementing the system design presented in Chapter 4 means devising working, synthesizable VHDL code for each entity of the MAC core as well as their interconnections. For the verification purposes of this thesis, it is sufficient if it runs on a GigaBee module, but as was stated in the initial goals, the code should be flexible enough to allow for easy porting to other devices and vendors. In fact, the majority of it should be generic VHDL that does not use components specific to certain FPGA families at all. Some design entities such as the MII/GMII input/output architecture and the user FIFOs warrant special attention as their implementation needs more thought beyond just adhering to the specification. We will see how they are realized next.

The previous design choices were independent of the hardware description language used, as will many of the considerations relevant for implementation be. Generally, the focus is drawn to the ideas and concepts inherent to the source code rather than the concrete lines of code here. For some details that concern the goal of easy maintainability and understandability of the actual code, though, it is necessary to briefly discuss a few aspects of VHDL. As a general measure to make the code easier to grasp, a consistent coding style was applied throughout the source code; specific rules are found in Appendix B.

## 5.1 (Gigabit) media-independent interface input/output

The only entity introduced as device-dependent was `mii_gmii_io` which implements the input/output part of MII and GMII. We will now see why this is not possible with generic HDL code and how it was concretely realized, starting with the transmission of Ethernet frame data.

### 5.1.1 Transmission

As explained in Section 2.1.2, MII/GMII transmission clocking is complicated by the fact that two completely different clock sources must be used depending on the link speed. This and the need to meet the gigabit media-independent interface timing led to the structure seen in Figure 5.1. Note that the `mii_gmii` TX logic block is only included for clarity and not part of `mii_gmii_io`.

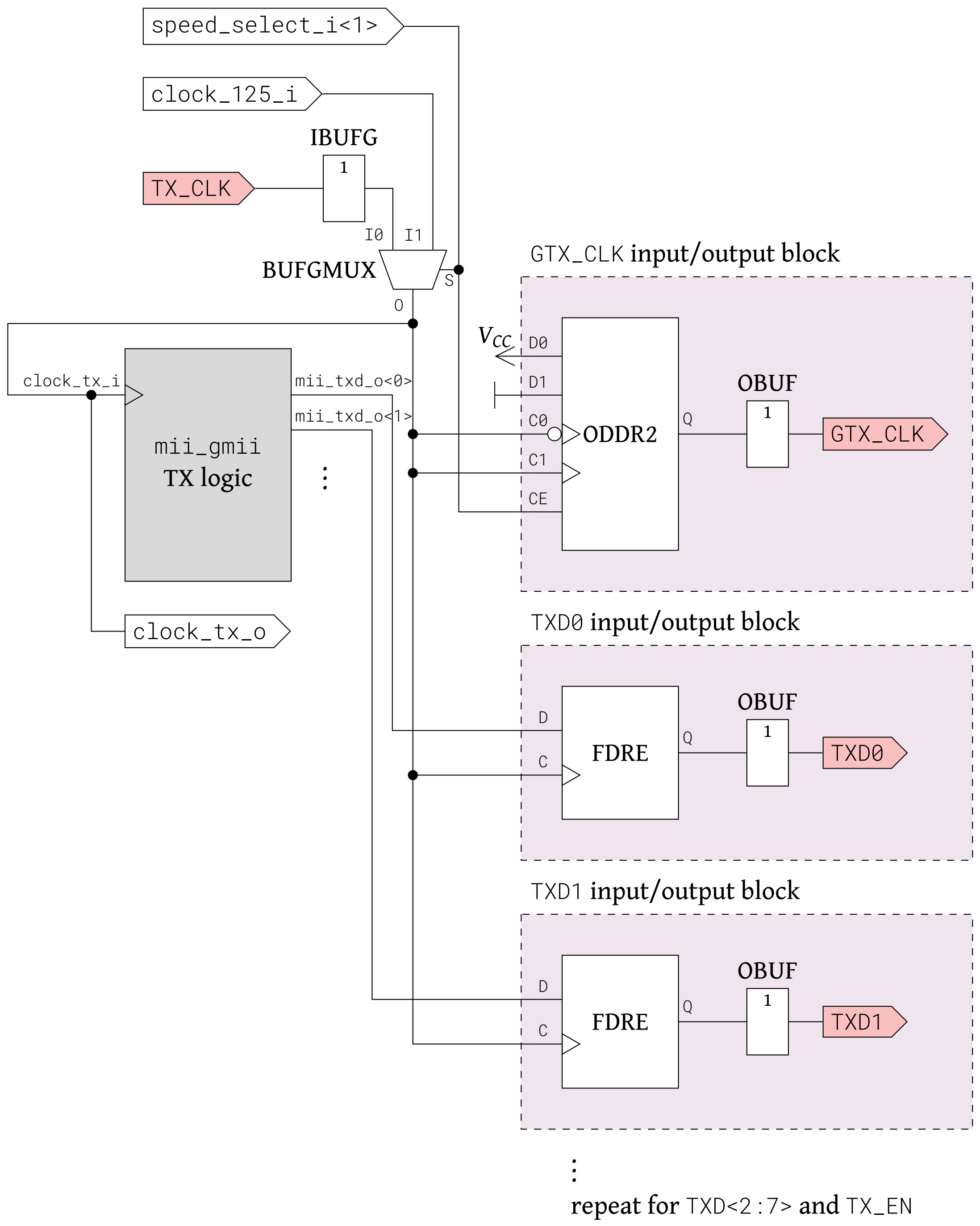


**Figure 5.1:** Structure of the transmission part of `mii_gmii_io` on Spartan-6 family FPGAs

The flip-flops in the Spartan-6 FPGAs do not have the ability to switch between different clock signals, but the global clock buffers (BUFGMUX) do. One of them is used for generating the `clock_tx` signal necessary for the TX logic and output. The first input `I0` is the MII `TX_CLK` pin buffered by an IBUFG clock input buffer that provides a relatively direct connection to the multiplexer. The second input `I1` needs to be a user-provided 125 MHz reference clock which is called `clock_125`. Where this signal comes from is not important, it might for example also come directly from another input pin (although there are restrictions on which pins can be used in this case), or be generated by a digital clock manager from a system clock with differing frequency. What does matter is that it should not already be the output of another global clock buffer as cascading them results in the usage of general-purpose routing for the connection [24]. Which of those two clocks is put on the output `O` is decided by the select input `S`. The most significant bit of the `speed_select` signal generated by the MII management interface control component indicates whether the PHY currently operates at gigabit speed (logic high) or 10 or 100 megabit speed (logic low). This is exactly what is needed for the `S` input, no further conversion is necessary. The correct TX clock is then provided by the `mii_gmii_io` entity on the `clock_tx_o` port for the transmission logic throughout the MAC.

The Ethernet standard does not specify whether the MII `TX_CLK` continues running in GMII mode when both interfaces are supported, thus we have to assume that it stops on transition to gigabit speed. This means that the BUFGMUX can not be operated in synchronous glitch-free mode. Switching asynchronously between clocks implicates that glitches can occur on the resulting clock signal e.g. when changing from a clock that has just fallen to one that is just about to rise, possibly violating timing requirements in every circuit that uses the signal. The consequences are metastability and an inconsistent system state. To get the MAC back into a defined state, the `reset_generator` component initiates a complete reset of the core whenever it detects a speed change.

When the 125 MHz clock is used in GMII mode, it is necessary to output the TX clock for the PHY on the `GTX_CLK` pin. Connecting the signal directly to an output buffer makes for a sub-optimal solution due to routing restrictions in the Spartan-6 family FPGAs. If the design is routable at all, the output may exhibit high delay and skew issues. The technique recommended by Xilinx in its ISE design suite is to use a double data rate output register ODDR2, connect the desired clock to the first clock input `C0`, and its inverse to the second clock input `C1`. We will discuss later why the exact opposite was done here. Each time a rising edge is detected on `C0`, the value of the input `D0` is put on the output `Q`. Similarly, a rising edge on `C1` (which is equal to a falling edge on `C0` since both are the inverse of each other here) puts `D1` through. Fixing `D0` to a logic one and `D1` to a logic zero then leads to a clock signal on `Q` that is identical to the input `C0` save for the delay induced by the register and the output buffer. To avoid outputting an unused high-frequency signal in MII mode when GTX_CLK is not used, the clock enable input CE of the ODDR2 instance is connected to the speed indicator. When it is at logic low for 10/100 Mb/s link speed, the output will stop toggling.

For the TX signal lines TXD<0:7> and TX_EN, we need to take special care to meet the GMII timing requirements of 2.5 ns setup time and 0.5 ns hold time with respect to the rising edge of GTX_CLK. Connecting the flip-flops that generate the final output values inside the FPGA logic slices directly to the output buffers results in high routing skew between the bus signals and an unclear timing relation to the clock output pin. Utilizing the OLOGIC2 output flip-flops integrated into the input/output blocks, however, eliminates any larger skew as the routing delay between the flip-flop and the buffer located in the same IOB is close to identical for all signals. The relation to the GTX_CLK output is also clear: although the HDL primitive for a D-type flip-flop with synchronous reset FDRE is different from the ODDR2 primitive used for the clock signal, both end up in OLOGIC2 blocks here, using the same hardware. The delay from a rising edge on the TX clock to the register input value appearing on the device pin for any TX signal including the output clock is thus almost equal. With this structure, the data pins will change when GTX_CLK rises and are almost guaranteed to violate the hold time requirement. Actually, the opposite should be done: if the outputs change on the falling edge of the clock, both the setup and hold time is met and there is still a 1.5 ns setup margin for bus skew. This is easily achieved by swapping the C0 and C1 input pins of the ODDR2 instance used for GTX_CLK, thereby inverting the clock output.

The aforementioned considerations were only valid for gigabit speed, but the resulting solution can be used without modifications for plain MII, too. It mandates a setup time of 15 ns and a hold time of 0 ns. Put differently, with a 25 MHz interface clock at 100 Mb/s link speed, 25 ns are available for the propagation of a rising edge on TX_CLK through the input and global clock multiplexer to the clock input pin of the output flip-flops and from there for the propagation of the FF input signal to the device pin. All component and routing delays are small enough to meet this requirement without special intervention.

As different aspects need to be discussed for the reception case, it can not be implemented analogously and warrants a closer inspection.

### 5.1.2 Reception

To begin with a simplification compared to the transmission structure, the same interface clock is used for both MII and GMII. In contrast, other points get more complicated: for GMII output, 5 ns of the 8 ns clock period are available for transitioning the data signals. The opposite is true for the input case: the pins need to be sampled within a short window of 2 ns, not even 3 ns as one would assume because the Ethernet standards mandates differing timing requirements for the sender and the receiver to account for propagation delay mismatch between the clock and signal lines [5]. A prerequisite for matching the timing at all is that the Spartan-6 family IOB input flip-flops allow for combined setup and hold times of shorter than 2 ns, which they do. Depending on the speed grade of the device, the duration for which the signal has to stay stable is between 0.9 ns and 0.3 ns [45]. The key problem is making sure that the rising edge of the `RX_CLK` clock reliably arrives at the capture flip-flops when the centers of the data validity window of the signal lines and the flip-flop input window requirement line up.

Skew is usually the engineer's enemy, but here we can use it to our advantage: controlled insertion or removal of skew between the clock and signal paths is required. Possibilities to achieve this include (in order of complexity):

- Trying to set timing constraints on the FPGA input pins in the design and letting the hardware synthesizer figure out how to make ends meet. As design tools are generally not advanced enough to devise complicated IO clocking schemes and using only routing delays comes with a number of limitations, this will most likely fail.
- Using a digital clock manager or PLL to induce a defined phase shift on the clock. 10 Mb/s operation is not supported in this case as the minimum input frequencies for both the DCM (5 MHz) and the PLL (19 MHz) [45] are well above the 2.5 MHz required.

  A switch between 25 MHz and 125 MHz interface clock rate when the PHY changes speeds will cause the clock output to become incoherent with the interface clock until the DCM/PLL regains its locking state, but only for a short time of 5 ms at maximum. A loss of link also allows the device to increase the time between two rising edges from 8 ns up to 16 ns according to the IEEE standard [5], leading to the same situation. Both conditions should only occur on initial connection of the link or when errors emerge at the physical layer. Under those circumstances however, frames must be expected to get lost until the link is stable, so the small recovery time of the clock manager is of no concern.

  The clock may also be stretched or shortened by the PHY when it transitions between its reference clock and the clock recovered from the medium. Although the standard does not explicitly state it for the MII and GMII interfaces, with the most commonly used copper Ethernet variants 100BASE-TX and 1000BASE-T this should only happen in half-duplex operation. The stream and therefore clocking on the physical medium in full-duplex mode is continuous as long as power-saving capabilities are not specifically enabled.

- Using the dedicated IO clocking buffer BUFIO2 that provides minimum delay routing at low skew for the clock and inserting an IODELAY2 element into each data signal path to compensate for its propagation delay. The delay time is configurable, so in theory, the value could be exactly matched to the signal setup time and the time needed for the clock to propagate through the BUFIO2 to the clock pin. In practice though, "the Spartan-6 FPGA delay line is not compensated for either temperature or voltage" [42], which means that the exact delay time varies greatly depending on the operating conditions.
- Combining a DCM or PLL for 1000 Mb/s GMII operation with a simple buffer of `RX_CLK` for 10/100 Mb/s MII operation and switching the clock source for the capture flip-flop as needed with a BUFGMUX clock multiplexer. The DCM/PLL with its configurable phase shift guarantees that timing is reliably met for Gigabit Ethernet, but is bypassed for interface clock speeds that it does not support and, in fact, is not needed for. MII has much more lenient timing requirements of 10 ns setup and 10 ns hold time that can be met without any special arrangements. The issue with the clock manager losing its lock on the clock signal when the PHY decides to stretch or short a pulse stays.

None of the proposed solutions is completely satisfactory. The last one is the most reliable option, but at the same time, it is also the most complex and requires a DCM or PLL, of which only a small number are available in the XC6SLX45, and they might be needed by the application using the MAC core. Using BUFIO2 and delay lines is straightforward and only requires one of 32 IO clocking buffers. The IODELAY2 elements are always associated with exactly one device pin and can not be used for another purpose by the user anyway. We see the resulting RX structure of `mii_gmii_io` in Figure 5.2. The BUFIO2 element also provides a second output clock that can be put on the global clocking interconnect by a global clock buffer BUFG. All reception logic in the design uses this clock to capture the MII interface data from the IO flip-flops. Again, the `mii_gmii` RX logic block is not part of `mii_gmii_io` and only included for clarity.

Due to the way the IO clocking is routed inside the device, some constraints apply to the MII/GMII RX pin placement beyond the limitation that `RX_CLK` must be put on a global clock input pin which is true for all possible solutions. All RX interface signal pin locations must additionally belong to the same BUFIO2 clocking region. The exact extent of those regions is described in detail in the clocking user guide [24], but it should generally not be a problem for the user to meet this requirement. The primary target platform of this thesis - the Trenz Electronic GigaBee module - allows for this option.

We still have to address the major downside of this implementation variant: setting a delay value that guarantees reliable operation under all conditions. The IODELAY2 element can

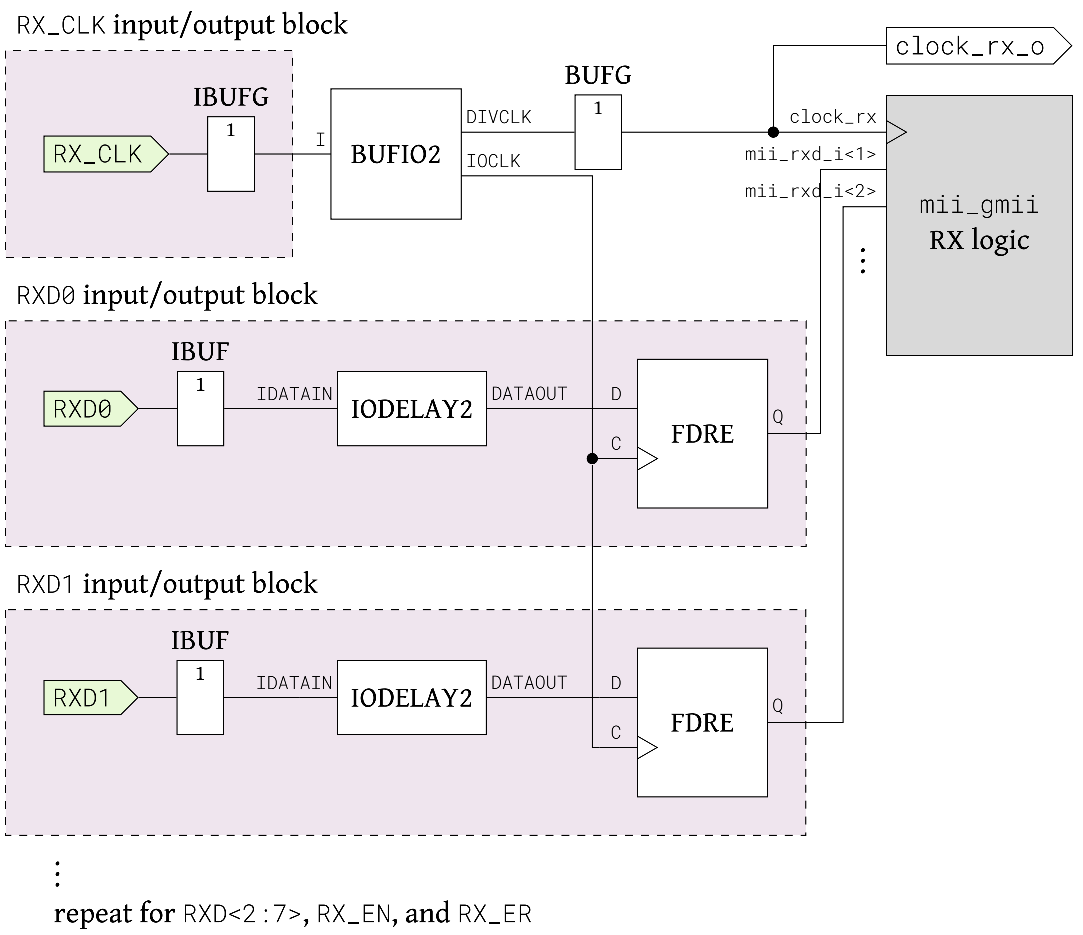


**Figure 5.2:** Structure of the reception part of `mii_gmii_io` on Spartan-6 family FPGAs

be configured with the IDELAY_VALUE parameter, an integer between 0 and 255, that determines how much delay is applied to the signal. How this number maps to the time domain is not stated in the SelectIO user guide [42], so it must be found by trial and error. After implementing a design, the static timing report generated by the Xilinx ISE tool suite shows how long the delay applied actually is.

The total delay of the RX clock from the device pin until the final flip-flop input port is at least 1.7 ns under optimum conditions and at most 3 ns under worst-case conditions on the reference board according to the timing report. The input FF in the Spartan-6 FPGA used needs 0.54 ns of setup time $T_{setup,FF}$ and zero hold time when used in combination with IODELAY2 [45]. To meet both that and the GMII validity window $T_{valid,GMII}$ of 2 ns combined setup and hold time, the delay on the data signals must thus be between 1.7 ns and 3.2 ns for the best case and between 3 ns and 4.5 ns for the worst case. Figure 5.3 illustrates the difference between the signals at the device pin and inside the FPGA for the best-case scenario. The data signals are captured correctly when they are valid during the time span indicated by $T_{setup,FF}$. Although there is some overlap between the best and worst case intervals, we must keep in mind that no matter what value is chosen for IDELAY_VALUE, the data path delay

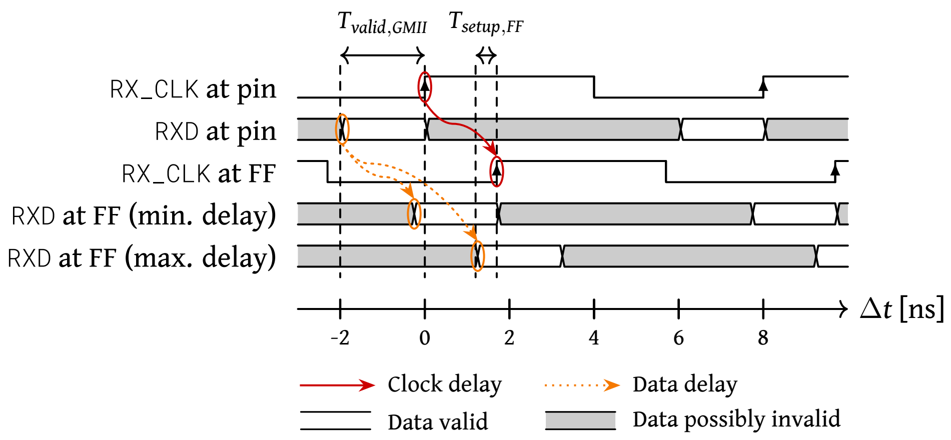


**Figure 5.3:** Delay of clock and data signals on the GMII receive side at minimum clock delay

will also exhibit strong variation with device temperature and voltage. This is not only a result of the nonexistent compensation in IODELAY2, but also of varying propagation and routing delays of the input buffer.

Experimenting with IDELAY_VALUE shows that an optimal solution is not possible in this configuration. A value of at least 27 is needed to reach the required 1.7 ns under the best operating conditions, but it will cause the worst case data path delay to climb up to 6.2 ns already, exceeding the maximum allowed 4.5 ns by 1.7 ns. However, using a value of 10 provides a delay range of 1.4 ns (best case) to 4.8 ns (worst case). Correlating this to the GMII timing, we see that 2.3 ns setup and 0.3 ns hold time can be met under all circumstances. This is well within the range of 2.5 ns setup and 0.5 ns hold time the PHY has to guarantee at its output according to the standard. Usually, there will be some added margin on the PHY side. The only limitation is therefore that the printed circuit board must be designed with very low skew between the data and clock MII receive direction signals.

Looking back to Section 3.1, we know that Xilinx has also released an Ethernet MAC compatible with Spartan-6 FPGAs themselves. The approach to realizing MII and GMII I/O in a simple and reliable manner described above has led to a structure very similar to what Xilinx is using [46], although they do not describe the delay line timing for an exemplary case. This should be of no surprise as the limitations of the device do not allow for radically different solutions.

## 5.2 Transmit FIFO

As Xilinx provides a very flexible generator for FIFOs, it is not necessary to implement the actual FIFO memory. However, a component that reads the data and passes it on to the `framing` entity is still needed. The previously introduced `tx_fifo` entity is therefore subdivided into two: the actual FIFO memory and `tx_fifo_adapter`. This adaption module is device-independent. It reads the size of the next packet out of the FIFO when it is not empty, waits until at least as many bytes as were indicated are available for reading, passes the data on to `framing`, and then repeats the process.

The actual FIFO memory in fact does not need any device-specific features, too, but as the output of the Xilinx core generator is proprietary and can only be used with Xilinx devices, it should be considered at least vendor-dependent. Other FPGA manufacturers have similar generators that can be used instead when adopting the MAC core for their devices. It is also possible, albeit a lot more complicated, to implement a custom asynchronous FIFO in VHDL.

## 5.3 Receive FIFO

While it was viable to use a preexisting FIFO implementation for the transmit buffer, we have already discussed in Section 4.3.1 that the `rx_fifo` module can not be realized with a single actual first-in-first-out memory. It can be done with two of them, one for the packet sizes and one for their data, but this increases vendor dependencies when using generated FIFOs again like for the transmit path. For a custom implementation, just one memory is sufficient and simpler than two.

FIFOs on FPGAs are usually implemented with block RAM used as ring buffer for data storage and separate counters in the read and write domain that designate the next memory location to read or write. The most difficult problem is the synchronization of the counters across clock domains so that on the one hand values that have yet to be read out are not overwritten by the write side when the memory overflows and on the other hand the data transfer to the user stops when he or she has read all available bytes. A general implementation would use grey-encoded values when crossing clock domains to ensure that only one bit changes between clock cycles [25]. This results in a design that is quite complicated to get completely right. In this Ethernet MAC design, a different approach is used that results in a structure whose implications are more easily understood.

What makes this possible is that we know that the memory stores packets and their sizes and as such it is sufficient to synchronize the read and write sides per packet transaction as opposed to per data unit. Inside the memory, a 3-byte header is attached to each received packet. It contains a 1-byte validity indicator and the 2-byte size of the packet. The validity indicator is either all-zeros when there is no complete packet at the address yet or all-ones otherwise. The write side uses two pointers when receiving packets: the current packet pointer and the write pointer. The current packet pointer designates the address of the currently receiving packet's first header byte while the write pointer is the address the next byte of packet data should be written to. It advances after each received byte, while the current packet pointer only gets a new value after reception of one packet has completed. When a packet is received from `framing`, the following happens, in order:

1. If there is not enough room in the buffer to store the size and validity indication, skip the whole packet.
2. Set write pointer to 3 bytes after the current packet pointer. Write the first byte of data there.
3. Continue writing bytes and advancing the write pointer until the frame ends. If the buffer or the packet size would overflow on the next byte, skip the rest of the packet.
4. Subtract 4 bytes from the received packet size to remove the FCS when read out.
5. Write an all-zeros value at 3 bytes before the write pointer to mark the following packet invalid.
6. Write the least significant byte of the packet size at two bytes after the current packet pointer.
7. Write the most significant byte of the packet size at one byte after the current packet pointer.
8. Write an all-ones value at the current packet pointer to mark the packet valid.
9. Advance current packet pointer to the first byte after the packet data that has just been written.

The read side is simpler, it only uses one pointer, the read pointer. Initially, the read and write pointers point to the same address, the validity indicator of the first packet in the buffer. The read state machine waits for the memory value at the pointer to become all-ones, then it knows that the following packet can be read safely. It deasserts `rx_empty_o` to show the user that data is available and begins reading the packet size. The size is both given to the user and internally stored to detect when the packet ends. The following packet data is passed on to `rx_data_o` each time a byte is requested, until the end of the packet is reached. The state machine asserts the empty indicator again and tries to read the next packet.

So far, this process makes sure that the read side does not underflow. It is equally important to guarantee that the buffer contents are not corrupted when the FIFO is full and packets arrive. The memory itself was used to cross clock domains from the write to the read side. Now we need the opposite direction which can not be achieved by the same means as block

RAMs usually do not have two sides that can both be used for reading and writing. Instead, a normal register is used to push the last data address of a packet to the write side after it was completely read. This address is called the write safe pointer because it is guaranteed that writing bytes to the memory will not corrupt unread data until it is reached. An update enable signal synchronized by two flip-flops indicates that the address is valid to counter metastability issues.

No assumptions should be made about the clock speed of either side; it is for example plausible that the write side is at 2.5 MHz (10 Mb/s link speed MII), but the read side is at 125 MHz to support gigabit data transfer. Consequently, it can happen that the user has read a complete packet while the write state machine has not seen a single clock edge in the meantime. If the write safe address is updated again, it might get sampled in the write clock domain while it is transitioning, leading to metastability issues. A two-phase cross-domain handshake as proposed by Chu [25] solves the problem. The read state machine delays reading the next packet until the write side acknowledges the write safe address update.

On first glance, this approach might seem to negatively impact the maximum data throughput by a considerable amount, but it in fact does not. The Ethernet standard guarantees 12 byte times of interpacket gap, which is sufficient to complete the handshake when both sides run at the same clock speed. If the read side is faster, throughput is not adversely affected as data cannot arrive faster than the PHY allows. If the read side is slower, the user cannot expect to achieve the full data rate allowed by the Ethernet connection in either case, and the two or three read clock cycles needed to get the handshake acknowledgement into the read clock domain will not make a big difference.

However, a portion of the memory will always be unused at any time since buffer space belonging to a single packet is only made available after the user has read it completely. If the read clock is at least 125 MHz, a Gigabit Ethernet connection is used, and packets are read out of the FIFO immediately after they become available, the buffer must be big enough to hold two complete Ethernet frames of maximum size plus three bytes header and one frame check sequence (3038 bytes in sum) at the very least to avoid packet loss. To accommodate some leeway, a default RX memory size of 4 kiB is chosen. The user is free to change this size provided it stays a power of two.

What has not been clarified yet is whether accessing the block RAM of the FPGA device can be implemented in a device-agnostic manner. This is indeed the case, as special coding techniques exist that allow HDL synthesizers to automatically infer memory primitives of the targeted device. The documentation of the Xilinx Synthesis Technology (XST) for instance shows the exact conditions needed [47]. Other tools provide similar support with the same coding style. The general idea is to use a signal with an array type of the desired content type for conceptual storage, resulting in concise code. The following lines from the `rx_fifo` implementation serve as an example:

```
type t_memory is array (0 to (RX_MEMORY_SIZE - 1)) of t_ethernet_data;
signal memory : t_memory;
```

It is obvious that the `memory` signal can hold as many data bytes as specified in the `RX_MEMORY_SIZE` constant. Writing and reading the RAM need similarly few lines of code. This process writes `write_data` to the integer address `write_address`, synchronously to `clock_write`:

```
process(clock_write)
begin
        if rising_edge(clock_write) then
                memory(write_address) <= write_data;
        end if;
end process;
```

Reading data from the memory works analogously.

The code is much clearer than using a block RAM primitive with data, address, and enable signals, but has the same effect. The instantiation alone needs more than 100 lines of mostly boilerplate code with a Spartan-6 FPGA. Additionally, the memory size can be easily changed and the implementation can be used throughout FPGAs of many families and vendors without any modification. We should, however, keep in mind that the memory signal may only be manipulated in ways that can be mapped back to the hardware. It is, for example, not possible to read the same memory array from four different processes with distinct clocks if the FPGA does not happen to have a four-port RAM component. `rx_fifo` only uses one read and one write port, which is provided by almost all FPGAs.

## 5.4 Frame check sequence calculation

Calculation of the frame check sequence is in its most basic form performed one bit at a time, but here we would need 8 times the GMII clock frequency of 125 MHz to update the value on-the-fly while data is being sent or received. Doing so in parallel for 8 bits of Ethernet data per clock cycle allows using the same clock at the cost of more logic resources. The boolean equations needed for this can be derived mathematically as seen e.g. in [48] and then adopted into VHDL code one-to-one. The result of this is a long sequence of XOR operations that are virtually impossible to comprehend by looking at them. Sprachmann proposes a different solution in [49] that is vastly superior in understandability by using a simple linear feedback shift register for a one bit update step and letting the VHDL synthesizer figure out the logic gates needed for performing multiple steps in a single clock cycle. His approach is adopted with minor modifications in the MAC core FCS calculation.

This concludes the review of the Ethernet MAC implementation. We have seen not only how some parts particularly relevant to the design goals or the general operation of the MAC core were put into practice, but also a number of alternatives and why they were ultimately rejected. The next topic then is verifying the considerations of both this and the preceding chapter.

# Chapter 6

# Test and Results

The presentation of the IP core would not be complete without also demonstrating that it indeed performs the functionality it was designed to and, equally importantly, that the original goals were reached.

To this end, a sequence of tests was performed that in sum verifies all important aspects of the implementation. We will start with a computer simulation for the basic processes, continue with a hardware test and benchmark on the target platform, show the integration of the core into an existing application, and conclude with statistical information about the FPGA resource utilization and the code.

## 6.1 Functional verification

Before testing the MAC sublayer on actual hardware, its basic functionality is verified by a simulation of the VHDL code. A special VHDL entity, the testbench called `ethernet_mac_tb`, embeds the core including an application that uses its FIFO interface. The testbench emulates an Ethernet physical layer device's media-independent interface and directs the MAC user application to perform specific operations depending on what functionality is being checked.

First, `ethernet_mac_tb` hands a number of test packets to the MAC and expects that they either come back identically or not at all if they were deliberately invalid and thus should be dropped. Any incoming packets received by the MAC user code are looped back by copying them from the RX to the TX FIFO. This way, the complete data path from frame reception on the MII RX interface over reaching the user on the receive FIFO and being written to the transmit FIFO to frame transmission on the MII TX interface is tested. After that, the padding capabilities of the MAC sublayer are tested by having the user application send very small packets.

In detail, the following test cases are conducted at all supported speeds of 10, 100, and 1000 Mb/s (frame sizes measured without the frame check sequence):

1. Frames with each possible size in the range of 60 to 1514 bytes must be looped back identically. No other packets must be transmitted by the MAC.
2. The following malformed frames must not be looped back, but a 100-byte frame must be identically looped back after every frame to verify that the MAC is still functioning:
   (a) Frames with each possible size in the range of 1 to 59 bytes (too small) and 1515 to 1524 bytes (too big)
   (b) A frame with a size of 2118 bytes (overflowing 11 bits of size information)
   (c) A frame with a size of 9999 bytes (larger than the RX memory size)
   (d) A frame with a deliberately mismatching frame check sequence
3. After suspending the loopback process, sending 8 packets with a size of 1024 bytes each, and activating the loopback again, the first 3 packets must be looped back identically and no other packets must be transmitted by the MAC. This makes sure that overflowing the RX FIFO, which has a default size of 4096 bytes, correctly drops later packets and does not corrupt the previously received ones.
4. The loopback user application is replaced with a simple frame generator. Then, the MAC must correctly pad frames with each possible size in the range of 1 to 59 bytes to exactly 60 bytes in transmission without corrupting the contents.

The verification aborts and an error is indicated immediately if the MAC:

- ends transmission of a frame on a half-byte when plain MII is used,
- transmits a frame that does not start with 7 preamble bytes followed by the start frame delimiter,
- transmits a frame with a size that is below the minimum or above the maximum frame length,
- does not respect the minimum interpacket gap of at least 12 byte times on transmission,
- transmits a frame where the frame check sequence does not not match its contents,

- does not transmit a reply to a correct packet or the size and contents of the reply packet do not match the properties of the original one,
- transmits more replies than packets were sent to it,
- transmits a reply to an invalid packet in test case 2,
- transmits a reply to a packet that should have been dropped in test case 3,
- transmits a reply that is not exactly 60 bytes in size or does not match the expected contents in test case 4, or
- does not transmit a required packet within 20 ms.

If, on the contrary, the simulation concludes normally, all results are as expected. The behavior verified by the automatic testbench as listed above covers all functions and failure modes relevant for normal use except the rather simple operation of the MII management interface.

The open-source VHDL simulator GHDL [50] has successfully run the testbench, proving that the basic functionality is correctly working. The importance of this verification is underlined by the fact that most of the test cases above can not be replicated on hardware without an expensive Ethernet protocol testing device because network interface controllers do not commonly have the ability to send invalid packets. Also, stress tests with millions of packets that can detect intermittent failures are generally unfeasible in simulation due to high processing overhead. The manual check and benchmark in the following section will cover both this and MIIM.

Additionally, when the MAC sends a burst of multiple frames in test case 3, the duration of inactivity on the MII between two consecutive frames is measured. This value corresponds to the minimum interpacket gap that the sublayer is able to transmit. It was measured to be 14 byte times when using GMII and 12.5 byte times when using plain MII. The difference is a consequence of having an additional clock cycle per transmitted byte available in the state machines with MII. Substituting these values for $n_{IPG}$ in the equations for the achievable data rate devised in Section 2.1.3 results in a simulated maximum TX data rate of 974.0 Mb/s for Gigabit Ethernet and 97.50 Mb/s for Fast Ethernet.

For request-response communication schemes, the latency between a request packet and its corresponding response is a significant characteristic of the MAC core. In test case 1, where the packet is looped back identically and not processed in any way, the time required for a complete transaction (start of request transmission until end of response reception) was measured to be 1.76 µs at 1000 Mb/s link speed for a minimum size packet of 60 bytes and 125 MHz user clock. The actual processing inside the FPGA takes 0.63 µs or 36 percent of the total time, the rest is occupied by the data transmission on the media-independent interface. The values were determined by manually analyzing the MII waveforms produced by the simulation and allow for a maximum of 568,181 request-response transactions per second, which is usually more than sufficient. If necessary, bypassing the user FIFOs will minimize processing latency at the cost of design complexity.

Behavioral simulation generally operates on the basis of full clock cycles and does not account for the component and routing delays in FPGAs. It is possible to include these factors in a post-synthesis simulation by transforming the fully routed design netlist back into VHDL code. This code then contains only FPGA primitives like flip-flops, LUTs, and buffers and their interconnections, including the actual delays incurred. Consequently, this form of simulation is more accurate and can additionally detect timing violations. In contrast to the behavioral test, only a selection of corner cases and intermediate sizes is checked in test cases 1 and 2 instead of verifying all possible packet sizes. This allows the verification to complete in a reasonable amount of time despite the increase in processing time caused by the more accurate model. Furthermore, the testbench was extended to simulate and check the setup and hold times for GMII. The resulting test cases were successfully run on the commercial HDL simulator ModelSim by Mentor Graphics [51]. The free GHDL simulator could not be used again because it does not support the extended VHDL features needed for accurate timing simulation.

## 6.2 Benchmark

Both behavioral and post-synthesis simulation do not model the exact function of external ICs and circuitry. It is possible to repeat a false assumption e.g. about MII that was made while programming the core in the testbench and get a successful result. To show that the MAC is working in practice, we will look into performing a hardware test of the full core and an encompassing benchmark design on the GigaBee platform introduced in Section 3.2. Apart from simple packet reception and transmission, the maximum achievable data rate is of particular interest.

The general idea is similar to the simulation, although both link partners are now actual devices and not emulated any more. A personal computer is connected directly to the Ethernet connector of the GigaBee baseboard via a standard cable and sends packets to the device which the FPGA then returns identically, again testing the complete data path. Instead of verifying single packets of specific lengths, packets of maximum size are sent continuously, thereby allowing the peak data rate to be measured.

A custom benchmark application was developed for the Linux operating system in C++ that puts data packets on the link as fast as the network interface and operating system allow. The application stamps each packet with a continuously increasing 4-byte sequence number and checks that this number does not skip any values in the mirrored packets received from the network. If this happens, the MAC implementation is either malfunctioning or not fast enough as packets were definitely lost. Each packet carries different payload data that is compared for equality between transmitted and received packets to ensure that no packet contents were corrupted. To check for intermittent failures, the test is performed without interruption for 8 hours each at all three supported link speeds. For every second the amount of bits received from the FPGA is recorded. Basic MII management operation is also tested by this procedure because all data transfer depends on `miim_control` providing accurate PHY speed information.

Running the benchmark produced no errors and the measurement results we see in Table 6.1: the average data rate was always very close to the theoretical maximum with negligible deviation and no sequence numbers were skipped. However, the peak data rate shows that the maximally possible values (e.g. 975.3 Mb/s for Gigabit Ethernet) were never reached. If we assume that the MAC layer was not fast enough to mirror the incoming packets, it would have had to drop some of them. That this is not the case can only indicate that the device or application initially sending the packets is unable to saturate the physical link to the FPGA board due to e.g. processing overhead in the network interface controller or the operating system. Without enterprise-grade hardware and preferably a real-time operating system, the absolute maximum data rate that the MAC sublayer is able to process cannot be measured this way.

Nonetheless, this does not cause the test to become meaningless. It has shown that the MAC implementation is correctly working on the target platform, adopts to the physical link speed, and is able to continuously both send and receive at 99.2 percent of the theoretical maximum speed allowed by the Ethernet standard.

Even though the computer system used for the benchmark was clearly not able to *send* packets out at full Gigabit Ethernet rate, a short experiment revealed that it was able to *receive* at this speed. This fact was utilized for measuring the maximum TX data rate of the MAC core. Instead of transmitting packets on the personal computer and having them looped

**Table 6.1:** Results of the MAC loopback benchmark. Data rates are measured in the direction from the FPGA to the test computer.

| Link speed | Peak data rate | Avg. data rate | Std. deviation | Lost packets |
|---:|---:|---:|---:|:---:|
| 10 Mb/s | 9.7440 Mb/s | 9.7307 Mb/s | 0.0043 Mb/s | 0 |
| 100 Mb/s | 97.332 Mb/s | 97.307 Mb/s | 0.011 Mb/s | 0 |
| 1000 Mb/s | 968.24 Mb/s | 967.70 Mb/s | 0.93 Mb/s | 0 |

back, a process on the FPGA continuously fills the TX FIFO with maximum-size packets. The benchmarking application on the computer only needs to monitor the incoming interface data rate. As intermittent failures in the general operation of the core were already tested in the previous benchmark, a shorter duration of 60 seconds can be used now.

The measured average TX data rates were 974.0 Mb/s for Gigabit Ethernet, 97.50 Mb/s for Fast Ethernet and 9.750 Mb/s for 10 Mb/s Ethernet. They are consistent with the values calculated from the simulation results in the preceding section.

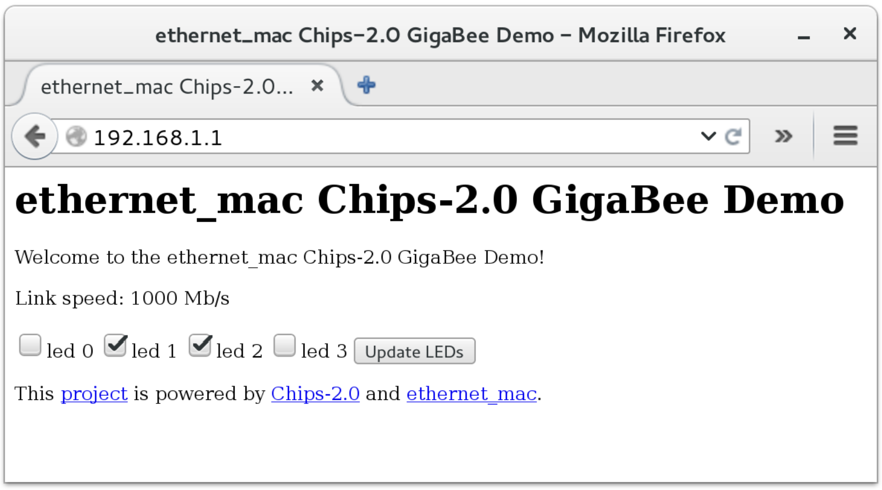
## 6.3 Example application

The benchmark code already serves as an example for the way users can interact with the MAC sublayer implementation presented in this thesis, but showing a more elaborate application will further prove that the core can be easily integrated.

Higher-layer network protocols such as IP and TCP can be implemented more easily in software programming languages than in hardware description languages. The open-source project Chips [52] allows compilation of specially crafted code written for sequential execution in the C programming language to synthesizable Verilog-HDL code. It interacts with the rest of the hardware design using 16-bit wide data buses. The author also provides a demo [53] that implements basic IP functionality and a simple HTTP server on a specific development board. It supports only Gigabit Ethernet connections via GMII.

To use this project to demonstrate the functionality of the MAC sublayer, the core must replace the prior implementation of Ethernet connectivity, an adaptor must be installed as MAC user application to convert between the MAC FIFO and the Chips interface, and the existing code must generally be adopted to the pinout and infrastructure of the GigaBee platform used in this thesis. Writing the adaptor was very straightforward as it primarily needs to copy each two Ethernet data bytes into one word on reception and split each word into two bytes on transmission. Circa 150 lines of VHDL code were necessary. The complete result of the customization is available at [54].

After downloading the design onto the FPGA, a standard web browser was pointed at the static IP address set in the C code. Figure 6.1 shows a screenshot of the resulting web page. Pressing the “Update LEDs” button changes the state of the light emitting diodes on the baseboard as expected. The test was successfully repeated at 10 and 100 Mb/s link speed.


ethernet_mac Chips-2.0 GigaBee Demo – Mozilla Firefox
ethernet_mac Chips-2.0...
192.168.1.1

ethernet_mac Chips-2.0 GigaBee Demo

Welcome to the ethernet_mac Chips-2.0 GigaBee Demo!

Link speed: 1000 Mb/s

☐ led 0 ☑ led 1 ☑ led 2 ☐ led 3 Update LEDs

This project is powered by Chips-2.0 and ethernet_mac.

**Figure 6.1:** Index page of the modified Chips webserver demo project using the MAC sub-layer presented in this thesis

## 6.4 FPGA resource and code statistics

On the test platform, the fully routed design including the loopback benchmark application consumed the resources we see in Table 6.2. The amount of available units is different for most other members of the Spartan-6 family and is included only as an indicator for how much logic is left for actual user applications on the smallest GigaBee micromodule. Also, the number of used resource units will vary greatly when the MAC is ported to other FPGA families.

Although it was decisively not a goal of the present thesis, the utilization of FPGA resources is quite small as a consequence of the minimal feature set.

The complete core of the MAC sublayer (without any user application) was implemented in 2,275 lines of actual VHDL code not including blank lines and 531 additional lines of comments. The testbench additionally needs 869 lines of code and 149 lines of comments. In other words, comments make up approx. 18 percent of all lines and verification code accounts for about 28 percent of the lines of code.

**Table 6.2:** MAC sublayer resource usage (including minimal encompassing user design) on the Xilinx Spartan-6 family FPGA XC6SLX45-2FGG484

| Resource | Used | Total available |
|---|---:|---:|
| Slice registers | 603 | 54,576 |
| Slice LUTs | 882 | 27,288 |
| Occupied slices | 349 | 6,822 |
| RAMB16BWERs (block RAM) | 3 | 116 |
| BUFG/BUFGMUXs | 3 | 16 |
| DCMs | 1 | 8 |

# Chapter 7

# Conclusion

This thesis presented an Ethernet MAC sublayer design and corresponding implementation in VHDL that was devised primarily with simplicity in mind, both in external usage and in internal operation. Smaller applications that want to use Ethernet directly from the HDL code such as the transmission of measurement data in a network of ultra-wideband sensors will benefit from the simple FIFO interface offered while nevertheless being able to send and receive at almost full Gigabit Ethernet speed.

The MAC supports communicating with a physical layer integrated circuit via the standard media-independent interface at 10, 100, and 1000 Mb/s link speed and automatic detection of the current speed. Prior FPGA implementations of tri-mode Ethernet MAC sublayers were laden with nonessential features such as CSMA/CD for half-duplex connections, energy-efficient Ethernet support, or complex on-chip bus interfaces. While beneficial for e.g. using them in combination with embedded processors, an increasing amount of features is accompanied by increasing complexity and cost. In contrast, only essential Ethernet functionality was considered for the present thesis.

The design discussion demonstrated how ease of use and understanding were achieved in practice. We have seen that the tasks the Ethernet MAC has to perform were clearly separated into a concise structure of VHDL entities. The user interface is comprised of two FIFO buffers with well-known and simple interfaces for sending and receiving packets respectively. To avoid requiring the user to process invalid data, broken packets received from the network are dropped internally. All functionality is wrapped in a single entity called `ethernet_with_fifos` that the user can instantiate into his or her design with little effort.

We then reviewed the implementation of MII I/O functionality on the Xilinx Spartan-6 family FPGA used in the primary target platform, the Trenz Electronic GigaBee micromodule. Architectures were presented for both transmission and reception that allow dynamic switching between MII and GMII and provide sufficient performance even at 1000 Mb/s link speed.

For the FIFO buffers facing the user, a generated standard implementation was chosen for the TX direction and a custom one for the RX direction because it was the best way to ensure a consistent, symmetrical interface. By synchronizing the read and write pointers per packet transaction, the complexity of using grey counters usually needed for asynchronous FIFOs was avoided.

A self-checking VHDL testbench confirmed that all basic functionality was implemented correctly. To ensure that no major speed limitations emerged as a consequence of keeping the design simple, a benchmark was performed which clearly proved that performance very close to the theoretical maximum is reached at all three supported link speeds. Finally, integrating the MAC sublayer into an existing web-server demo project with little effort has shown us that applications can make use of the core easily.

The resulting source code is published on the online collaboration platform GitHub [55] under an open-source license. The preceding chapters specifically described the state found at [56], but further updates to the code might be made in the future. To the knowledge of the author, this is the first VHDL tri-mode Ethernet MAC implementation available without any license fee. One free Verilog alternative exists at OpenCores [34], but its design goals are severely different.

While attention was paid to the secondary goal of making large parts of the IP core device-independent, the TX FIFO is not completely satisfactory in this regard. It relies on the proprietary LogiCORE generator that will only work with Xilinx FPGAs. Although this was a deliberate choice for the sake of simplicity, it becomes a significant limitation when porting the MAC to FPGAs of other vendors. Future work should provide a custom implementation similar to the RX FIFO directly in VHDL code as an alternative where using the Xilinx FIFO is unfeasible.

Although all basic Ethernet functionality is included, some secondary features were left out. Half-duplex operation is highly unlikely to be encountered in modern networks, but flow control functionality is certainly desirable when operating at high data rates. The IEEE standard even recommends that every Gigabit Ethernet component supports flow control [6]. However, care must be taken to reach a fair compromise between the benefit of additional features and the complexity they introduce. While it should be possible to integrate flow control into the existing design with only minor modifications to the current entities by adding a component that handles the transmission and reception of the corresponding control frames, doing so is expected to compromise the clarity of the present MAC sublayer too much. In contrast, additional interfaces to the PHY such as the reduced gigabit media-independent interface (RGMII) should not pose a problem since they can be implemented by replacing the `mii_gmii` and `mii_gmii_io` entities without adding additional components at all.

An aspect of media access control functionality that has received little attention in the present thesis is the addressing of nodes on the network. All packets received from the PHY are handed to the user, no matter whether they were ultimately destined for the device or not. He or she is then responsible for checking the destination address and acting accordingly. The amount of non-matching data received will be small because Ethernet switches generally forward packets only to the switch port corresponding to the destination address anyway. Nevertheless, dropping packets meant for other network nodes inside the MAC sublayer already should be considered to further improve user-friendliness. Similarly, outgoing packets need a valid source address that could be inserted automatically by the IP core.

When we take a look at the bigger picture, we observe that Ethernet is rarely used by itself without at least network and transport protocols such as IP and TCP. The ultra-wideband sensor networks which this Ethernet MAC was primarily developed for are no exception. Consequently, most users will have an additional demand for an FPGA core that realizes this functionality, so that only the actual application layer functionality is left for them to implement. Quite similar to the original problem statement that led to this thesis, simple and understandable hardware TCP/IP implementations for FPGAs are scarce. Future work should investigate which features of which higher-layer protocols are essential for basic applications and how they can be implemented in cooperation with the MAC sublayer presented in this thesis in a straightforward manner to offer a comprehensive solution for Ethernet communication.

# Appendix A

# Timing Diagrams

The following diagrams show the detailed timing when interacting with the `framing` entity from the user (FIFO) side. 4 bytes of data are transmitted or received respectively. The `mii_gmii` side is identical for reception and only differs in the missing `tx_busy` signal for transmission. The meaning of the signals is provided in Section 4.3.2 and, more specifically, in Table 4.1.

## A.1 Transmission

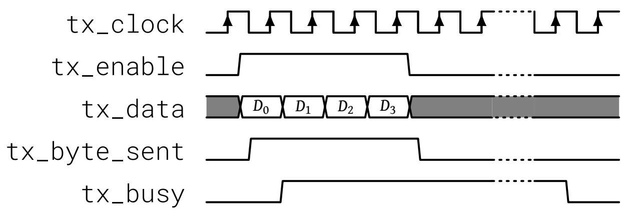


**Figure A.1:** Transmission of a frame when GMII is used

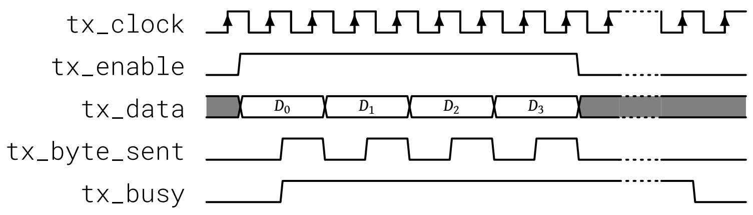


**Figure A.2:** Transmission of a frame when MII is used

## A.2 Reception

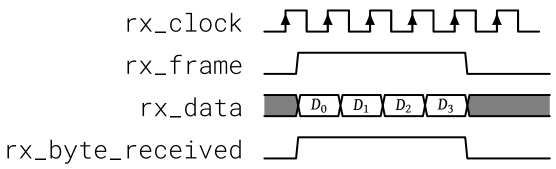


**Figure A.3:** Reception of a frame when GMII is used

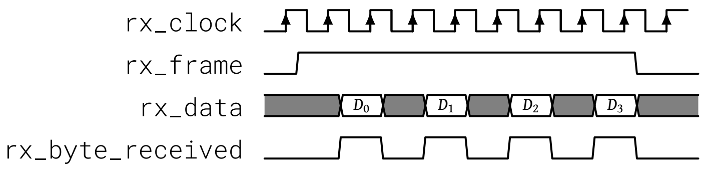


**Figure A.4:** Reception of a frame when MII is used

# Appendix B

# Coding Style

A consistent coding style makes the code considerably easier to grasp both locally and in its entirety. To that effect, the following principles have been applied consistently throughout the source code:

- Expressive names for signals, variables, constants etc.
- Prefixing type names with `t_` and suffixing all entity port names with their respective direction (`_i` for input, `_o` for output, or `_io` for both)
- Usage of the standard IEEE package `ieee.numeric_std` instead of vendor-specific ones for mathematical functions
- Putting types, constants, and functions common to multiple entities into shared packages instead of repeating them
- Distinct usage of signals for storage and variables for intermediate values
- Usage of generics and type attributes for flexibility
- Simple one-process unified coding style for finite state machines by default, usage of two processes where they provide an easier representation of the functionality
- Comments where they add meaningful information beyond the obvious operation performed by the code
- Indentation with tabulator characters

# Bibliography

[1] Marko Helbig et al. “Design and test of an imaging system for UWB breast cancer detection.” In: *Frequenz: journal of RF-engineering and telecommunications* (2012), pp. 387–394. ISSN: 2191-6349. DOI: 10.1515/freq-2012-0103.

[2] R. Salman et al. “On buried weapon detection by means of scattering matrix decomposition for quad-polarized ultra-wideband Radar.” In: *Ultra-Wideband (ICUWB), 2013 IEEE International Conference on*. Sept. 2013, pp. 113–119. DOI: 10.1109/ICUWB.2013.6663832.

[3] J. Sachs and R. Herrmann. “M-sequence-based ultra-wideband sensor network for vitality monitoring of elders at home.” In: *Radar, Sonar Navigation, IET* 9.2 (2015), pp. 125–137. ISSN: 1751-8784. DOI: 10.1049/iet-rsn.2014.0214.

[4] *Universal Serial Bus Specification*. Version 2.0. Compaq Computer Corporation et al. Apr. 2000. URL: http://www.usb.org/developers/docs/usb20_docs/usb_20_0702115.zip (visited on 2015-08-24).

[5] *IEEE Standard for Ethernet*. IEEE Std 802.3-2012 (Revision of IEEE Std 802.3-2008). The Institute of Electrical and Electronics Engineers, Inc. Dec. 2012. URL: http://standards.ieee.org/about/get/802/802.3.html (visited on 2015-07-07).

[6] Jörg Rech. *Ethernet. Technologien und Protokolle für die Computervernetzung*. 3rd ed. Hannover: Heise Zeitschriften Verlag GmbH & Co. KG, 2014. ISBN: 978-3-944099-04-0.

[7] *EZ-USB FX2LP™ | Cypress*. Cypress Semiconductor Corporation. URL: http://www.cypress.com/products/ez-usb-fx2lp (visited on 2015-08-24).

[8] *Trenz Electronic - Trenz Electronic TE0600 Series*. Trenz Electronic GmbH. URL: http://www.trenz-electronic.de/products/fpga-boards/trenz-electronic/te0600.html (visited on 2015-08-30).

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

[10] *IEEE Standard VHDL Language Reference Manual*. IEEE Std 1076-1993 (Revision of IEEE Std 1076-1987). The Institute of Electrical and Electronics Engineers, Inc. 1994. DOI: 10.1109/IEEESTD.1994.121433.

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

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

[13] Shuang Yu. *IEEE 802.3 "Standard for Ethernet" Marks 30 Years of Innovation and Global Market Growth*. IEEE Computer Society. 2013. URL: http://standards.ieee.org/news/2013/802.3_30anniv.html (visited on 2015-07-07).

[14] *Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*. 2nd ed. ISO/IEC 7498-1:1994. International Organization for Standardization and International Electrotechnical Commission. 1994. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip (visited on 2015-07-07).

[15] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach*. 4th ed. San Francisco: Elsevier, Inc., 2007. ISBN: 978-0-12-370490-0.

[16] *DP83848 Single 10/100 Mb/s Ethernet Transceiver Reduced Media Independent Interface™ (RMII™) Mode*. AN-1405. Texas Instruments. Apr. 2013. URL: http://www.ti.com/lit/an/snla076a/snla076a.pdf (visited on 2015-07-09).

[17] Yi-Chin Chu. *Serial-GMII Specification*. Version 1.7. Cisco Systems. July 2001. URL: http://www.angelfire.com/electronic2/sharads/protocols/MII_Protocols/sgmii.pdf (visited on 2015-07-09).

[18] *Reduced Gigabit Media Independent Interface (RGMII). Reduced Pin-count Interface For Gigabit Ethernet Physical Layer Devices*. Version 2.0. Broadcom, hp Invent, and Marvell. Jan. 2002. URL: http://www.hp.com/rnd/pdfs/RGMIIv2_0_final_hp.pdf (visited on 2015-07-09).

[19] Steve Kilts. *Advanced FPGA Design. Architecture, Implementation and Optimization*. New Jersey: John Wiley & Sons, Inc., 2007. ISBN: 978-0-470-05437-6.

[20] Andrew S. Tanenbaum and David J. Wetherall. *Computernetzwerke*. 5th ed. Munich: Pearson Deutschland GmbH, 2012. ISBN: 978-3-86894-137-1.

[21] Fred Halsall. *Computer Networking and the Internet*. 5th ed. Harlow: Pearson Education Limited, 2005. ISBN: 0-321-26358-8.

[22] Clive M. Maxfield. *FPGAs: Instant Access*. Burlington: Elsevier Inc., 2008. ISBN: 978-0-7506-8974-8.

[23] Gina R. Smith. *FPGAs 101*. Burlington: Elsevier Inc., 2010. ISBN: 978-1-85617-706-1.

[24] *Spartan-6 FPGA Clocking Resources*. UG382. Version 1.10. Xilinx, Inc. June 2015. URL: http://www.xilinx.com/support/documentation/user_guides/ug382.pdf (visited on 2015-07-22).

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

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

[27] *LogiCORE IP FIFO Generator*. PG057. Version 9.3. Xilinx, Inc. Dec. 2012. URL: http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v9_3/pg057-fifo-generator.pdf (visited on 2015-07-22).

[28] *SCFIFO and DCFIFO IP Cores User Guide*. UG-MFNALT_FIFO. Version 2014.12.17. Altera Corporation. Dec. 2014. URL: https://www.altera.com/literature/ug/ug_fifo.pdf (visited on 2015-07-31).

[29] *Tri-Mode Ethernet Media Access Controller (TEMAC)*. Xilinx, Inc. URL: http://www.xilinx.com/products/intellectual-property/temac.html (visited on 2015-08-07).

[30] *Interface Protocols - Triple-Speed Ethernet MegaCore Function*. Altera Corporation. URL: https://www.altera.com/products/intellectual-property/ip/interface-protocols/m-alt-ethernet-mac.html (visited on 2015-08-07).

[31] *DesignWare Ethernet MAC 10/100/1000 Universal*. Synopsys, Inc. URL: http://www.synopsys.com/dw/ipdir.php?ds=dwc_ether_mac10_100_1000_unive (visited on 2015-08-07).

[32] *10/100/1GMAC Ethernet Controller*. Mobiveil Inc. 2013. URL: http://mobiveil.com/download/2378/ (visited on 2015-08-07).

[33] *10/100/1000 Ethernet MAC*. MorethanIP GmbH. URL: http://www.morethanip.com/mbps_10_100_1000_mac.htm (visited on 2015-08-07).

[34] *10_100_1000 Mbps tri-mode ethernet MAC :: Overview :: OpenCores*. ORSoC AB. URL: http://www.opencores.org/project,ethernet_tri_mode (visited on 2015-08-30).

[35] Jon Gao. *10_100_1000 Mbps Tri-mode Ethernet MAC Specification*. Version 0.1. Jan. 2006. URL: http://opencores.org/websvn,filedetails?repname=ethernet_tri_mode&path=/ethernet_tri_mode/trunk/doc/Tri-mode_Ethernet_MAC_Specifications.pdf (visited on 2015-07-23).

[36] Sanket Suresh Naik Dessai. "Design and Implementation of an Ethernet MAC IP Core for Embedded Applications." In: *International Journal of Reconfigurable and Embedded Systems (IJRES)* 3.3 (Nov. 2014). ISSN: 2089-4864. DOI: 10.11591/ijres.v3i3.6245.

[37] Digital picture. Trenz Electronic GmbH. URL: http://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/TE0600-GigaBee_series/TE0603/pictures/TE603+TE600.angle.jpg (visited on 2015-07-22).

[38] *GigaBee XC6SLX Series User Manual. Industrial-Grade Xilinx Spartan-6 LX FPGA Micromodules*. UM-TE0600-02. Version 2.07. Trenz Electronic GmbH. June 2013. URL: http://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/TE0600-GigaBee_series/TE0600/documents/UM-TE0600-02.pdf (visited on 2015-07-22).

[39] *88E1111 Product Brief. Integrated 10/100/1000 Ultra Gigabit Ethernet Transceiver*. MV-S105540-00. Version A. Marvell International Ltd. Oct. 2013. URL: http://www.marvell.com/transceivers/assets/Marvell-Alaska-Ultra-88E1111-GbE.pdf (visited on 2015-07-22).

[40] *TE0603 GigaBee Header Baseboard. User Manual*. UM-TE0603. Version 1.00. Trenz Electronic GmbH. Sept. 2012. URL: http://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/TE0600-GigaBee_series/TE0603/documents/UM-TE0603.pdf (visited on 2015-07-22).

[41] *Spartan-6 Family Overview*. DS160. Version 2.0. Xilinx, Inc. Oct. 2011. URL: http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf (visited on 2015-07-22).

[42] *Spartan-6 FPGA SelectIO Resources*. UG381. Version 1.6. Xilinx, Inc. Feb. 2014. URL: http://www.xilinx.com/support/documentation/user_guides/ug381.pdf (visited on 2015-07-22).

[43] *AMBA Specifications - ARM*. ARM Limited. URL: http://www.arm.com/products/system-ip/amba-specifications.php (visited on 2015-08-07).

[44] *Wishbone :: OpenCores*. ORSoC AB. URL: http://opencores.org/opencores,wishbone (visited on 2015-08-07).

[45] *Spartan-6 FPGA Data Sheet: DC and Switching Characteristics*. DS162. Version 3.1.1. Xilinx, Inc. Jan. 2015. URL: http://www.xilinx.com/support/documentation/data_sheets/ds162.pdf (visited on 2015-08-02).

[46] *LogiCORE IP Tri-Mode Ethernet MAC*. UG138. Version 4.5. Xilinx, Inc. Mar. 2011. URL: http://www.xilinx.com/support/documentation/ip_documentation/tri_mode_eth_mac_ug138.pdf (visited on 2015-08-05).

[47] *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices*. UG687. Version 14.5. Xilinx, Inc. Mar. 2013. URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/xst_v6s6.pdf (visited on 2015-08-03).

[48] Ying Wu and Yuehong Qiu. “The 8-bit Parallel CRC-32 Research and Implementation in USB 3.0.” In: *Computer Science Service System (CSSS), 2012 International Conference on*. Aug. 2012, pp. 1079–1082. DOI: 10.1109/CSSS.2012.273.

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

[50] Tristan Gingold. *GHDL Main/Home Page*. URL: http://ghdl.free.fr/ (visited on 2015-08-07).

[51] *ModelSim ASIC and FPGA Design - Mentor Graphics*. Mentor Graphics. URL: http://www.mentor.com/products/fv/modelsim/ (visited on 2015-08-25).

[52] Jonathan P Dawson. *dawsonjon/Chips-2.0 · GitHub*. 2014. URL: https://github.com/dawsonjon/Chips-2.0 (visited on 2015-08-30).

[53] Jonathan P Dawson. *dawsonjon/Chips-Demo · GitHub*. 2015. URL: https://github.com/dawsonjon/Chips-Demo (visited on 2015-08-30).

[54] Philipp Kerling. *pkerling/Chips-Demo · GitHub*. 2015. URL: https://github.com/pkerling/Chips-Demo (visited on 2015-08-30).

[55] Philipp Kerling. *pkerling/ethernet_mac · GitHub*. 2015. URL: https://github.com/pkerling/ethernet_mac (visited on 2015-08-30).

[56] Philipp Kerling. *pkerling/ethernet_mac at 08e539d774687cc2ee318903ad3428299b6b3692 · GitHub*. 2015. URL: https://github.com/pkerling/ethernet_mac/tree/08e539d77 (visited on 2015-08-30).