Feasibility study: Implementation of a gigabit Ethernet controller using an FPGA

Richard Fält

LiTH-ISY-EX-3222

30 april 2003

Feasibility study: Implementation of a gigabit Ethernet controller using an FPGA

Examensarbete ufört i datorteknik
vid Linköpings Tekniska Högskola av

Richard Fält

Reg nr: LiTH-ISY-EX-3222

Handledare: Åke Andersson
Examinator: Dake Liu
Linköping, 30 april 2003

Avdelning, Institution Division, department Computer Engineering Department of Electrical Engineering Linköping University	**Datum** Date 2003-04-30

Språk Language ☐ Svenska/Swedish ☒ Engelska/English ☐ ______________	**Rapporttyp** Report category ☐ Licentiatavhandling ☒ Examensarbete ☐ C-uppsats ☐ D-uppsats ☐ Övrig rapport ☐ ______________	**ISBN** **ISRN** LITH-ISY-EX-3222-2003 **Serietitel och serienummer** Title of series, numbering / **ISSN** LiTH-ISY-EX-3222

URL för elektronisk version
http://www.ep.liu.se/exjobb/isy/2003/3222/

Titel
Title — Feasibility study:
Implementation of a gigabit Ethernet controller using an FPGA

Författare
Author — Richard Fält

Sammanfattning
Abstract

Background: Many systems that Enea Epact AB develops for theirs customers communicates with computers. In order to meet the customers demands on cost effective solutions, Enea Epact wants to know if it is possible to implement a gigabit Ethernet controller in an FPGA. The controller shall be designed with the intent to meet the requirements of IEEE 802.3.

Aim: Find out if it is feasible to implement a gigabit Ethernet controller using an FPGA. In the meaning of feasible, certain constraints for size, speed and device must be met.

Method: Get an insight of the standard IEEE 802.3 and make a rough design of a gigabit Ethernet controller in order to identify parts in the standard that might cause problem when implemented in an FPGA. Implement the selected parts and evaluate the results.

Conclusion: It is possible to implement a gigabit Ethernet controller using an FPGA and the FPGA does not have to be a state-of-the-art device.

Nyckelord
Keywords — CRC, Data Link Layer, Ethernet, FPGA, gigabit, GMII, MAC, MDI, MII, OSI/BR model, RS, PHY, Physical Layer

ABSTRACT

Background: Many systems that Enea Epact AB develops for theirs customers communicates with computers. In order to meet the customers demands on cost effective solutions, Enea Epact wants to know if it is possible to implement a gigabit Ethernet controller in an FPGA. The controller shall be designed with the intent to meet the requirements of IEEE 802.3.

Aim: Find out if it is feasible to implement a gigabit Ethernet controller using an FPGA. In the meaning of feasible, certain constraints for size, speed and device must be met.

Method: Get an insight of the standard IEEE 802.3 and make a rough design of a gigabit Ethernet controller in order to identify parts in the standard that might cause problem when implemented in an FPGA. Implement the selected parts and evaluate the results.

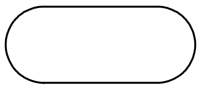
Conclusion: It is possible to implement a gigabit Ethernet controller using an FPGA and the FPGA does not have to be a state-of-the-art device.

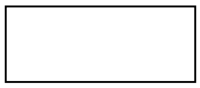
ABBREVIATIONS

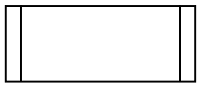
AUI	**A**ttachment **U**nit **I**nterface
b	**b**it(s)
B	**b**yte(s), octet(s) of bits
CRC	**C**yclic **R**edundancy **C**hecksum
DTE	**D**ata **Te**rminal
EDA	**E**lectronics **D**esign **A**utomation
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GMII	**G**igabit **M**edia **I**ndependent **I**nterface
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers, Incorporated
IP	**I**nternet **P**rotocol *or* **I**ntellectual **P**roperty
ISO	**I**nternational **S**tandardization **O**rganization
LLC	**L**ogical **L**ink **C**ontrol
MAC	**M**edia **A**ccess **C**ontrol (sublayer) *or* an electrical device that incorporates the MAC, MAC Control and Reconciliation sublayers
MDI	**M**edia **D**ependent **I**nterface
MII	**M**edia **I**ndependent **I**nterface
OSI/BR	**O**pen **S**ystem **I**nterconnection/**B**asic **R**eference (-model)
PHY	Electrical device that incorporates **Phy**sical layer except Reconciliation sublayer
RS	**R**econciliation **S**ublayer
RTL	**R**egister **T**ransfer **L**evel
TBI	**T**en **B**it **I**nterface
TCP	**T**ransmission **C**ontrol **P**rotocol
UDP	**U**ser **D**atagram **P**rotocol

FLOWCHART SYMBOLS

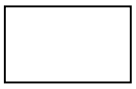
The following three types of symbols are used in flowcharts:

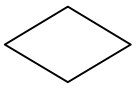
 Process

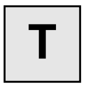
 Function

 Procedure

The following three types of symbols are used in state diagrams:

 Process

 Decision



Delay element

FONTS

- `Pascal` - This style are used for names (processes, functions, procedures, variables, constants, types etc.) that refers to the Pascal-like code in IEEE 802.3
- *VHDL* - This style are used for names (processes, functions, procedures, signals, variables, constants, types, block names etc.) that refers to the VHDL implementation done in this study
- $\boxed{\texttt{code}}$ - This style represents code written in VHDL or Pascal

Most of the names used in the VHDL implementation in this study are defined in the standard IEEE 802.3. E.g. an in standard defined process `foo` are implemented as a process called *foo* but the VHDL implementation may also incorporate other functionality besides that defined in the standard, thereby this distinction is made by the use of different styles.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER

1

Introduction

1.1 Background

This work has been carried out at Enea Epact AB, Linköping, at the Embedded Systems department between September 2001 and February 2002. Enea Epact AB is a consulting company focusing on high-performance systems.

At the Embedded Systems department, some of the systems that have been developed communicate with computers. In order to meet the demands from the customers, Enea Epact wants to know if it is possible or not to implement a gigabit Ethernet controller in an FPGA together with other functions.

1.2 Aim

Is it feasible to implement a gigabit Ethernet controller using an FPGA?

In the meaning of feasible, the following aspects shall be considered:

- Size — A golden rule is to never fill the FPGA more than 80% in order to avoid place & route problems. Besides the Ethernet controller, there must be sufficient place left to implement another, large design.
- Speed — There must be a speed margin in the range of 10 to 20 %, since only parts of the controller will be implemented.
- Device — The selected FPGA shall be a midsize device, which may belong to a high performance family of devices. Further, it is required not to use the highest speed grade available.

In addition, a rough estimation of the power consumption shall be presented.

1.3 General Description of Approach

1. Get an insight and common knowledge about the standard IEEE 802.3® and its associated standards where necessary.
2. Make a design suitable for VHDL implementation of necessary parts in the standard.
3. Implement these block in VHDL using Renoir from Mentor Graphics.
4. Check their functional behavior by simulation using ModelSim from Model Technology.
5. Synthesize the blocks using Leonardo from Exemplar Logic. Implement the design using ISE Alliance and then check the blocks' size and performance.
6. Collect the results from step 5 and decide whether it's feasible or not.
7. Estimate the power dissipation.

1.4 Outline and Reading Instructions

The next chapter introduces the OSI/BR model, which is a commonly used model for describing network communication. Also presented here is the standard IEEE 802.3, which among other techniques describes gigabit Ethernet.

Chapter 3 describes the methodology that was used during this work including which development tools that was used.

Chapter 4 tells about the implementation and how each step of the methodology was carried out in practice. An example is given how code written in the Pascal-like language used in IEEE 802.3 was ported to VHDL.

In **chapter 5,** the results (size, performance and power dissipation) of the implementation are presented.

A discussion regarding limitations, what could have been done better etc. and the conclusions can be found in **chapter 6** and **7** respectively.

Recommendations for future work are presented in **chapter 8**.

Finally, acknowledgements and a reference list are given in **chapter 9** and **10**.

The reader is expected to possess a basic knowledge about HDL languages such as VHDL and fundamental digital building blocks.

CHAPTER

2

Background

This chapter presents the standard IEEE 802.3 and a brief summary of the parts from it that has been used in this study.

In addition, the standard for the OSI/BR model is presented since the architectural description used in IEEE 802.3 is based upon this model.

2.1 Overview of IEEE 802.3 and the OSI/BR Model

This section will give an introduction to the standard IEEE 802.3 and its relationship to the architectural model of networking given by the Open System Interconnection Basic Reference Model, OSI/BR.

2.1.1 The ISO OSI/BR Model

The OSI/BR model is described in the standard ISO/IEC 7498-1:1994. The purpose with this model is to have a standardized model that gives a common basis for the development of different standards for system interconnection.

This model, shown in figure 1, consists of seven layers. Since the model is very adaptable, all layers are optional. Each layer has a dedicated function but because of its very commonly held structure, not every layer will have a counterpart in every standard for interconnection.

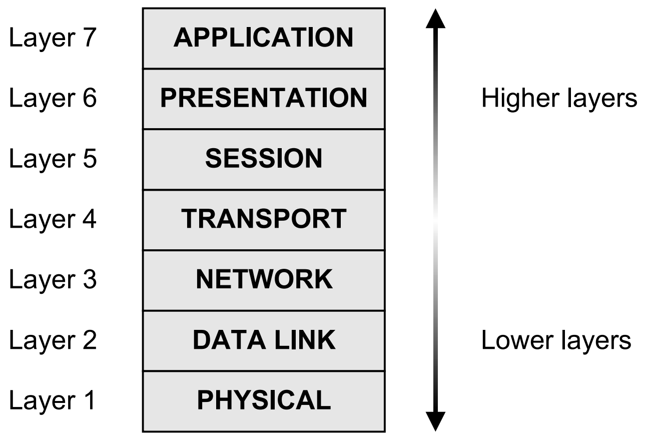


Figure 1: ISO's seven layers OSI/BR model.

In the IEEE 802.3 standard, the two lowest layers are referenced. These are the Physical and Data Link layers. The Physical layer describes the medium that the communication link uses and techniques associated with transmission and reception over the medium, e.g. which type of modulation that is used. The Data Link layer describes how the access to the link is managed, e.g. how a client connection to another client is established.

For orientation, some common protocols and their counterpart to the different layers in the OSI/BR model is presented in appendix A.

2.1.2 The IEEE 802 Standards and their relation to OSI

IEEE 802 is a family of standards for local and metropolitan area networks (LAN and MAN). Their internal relationships and their relation to the OSI/BR model is shown in figure 2. The standards apply to the two lowest layers of the OSI/BR model (Data Link layer and Physical layer) with the exception of IEEE 802.10.

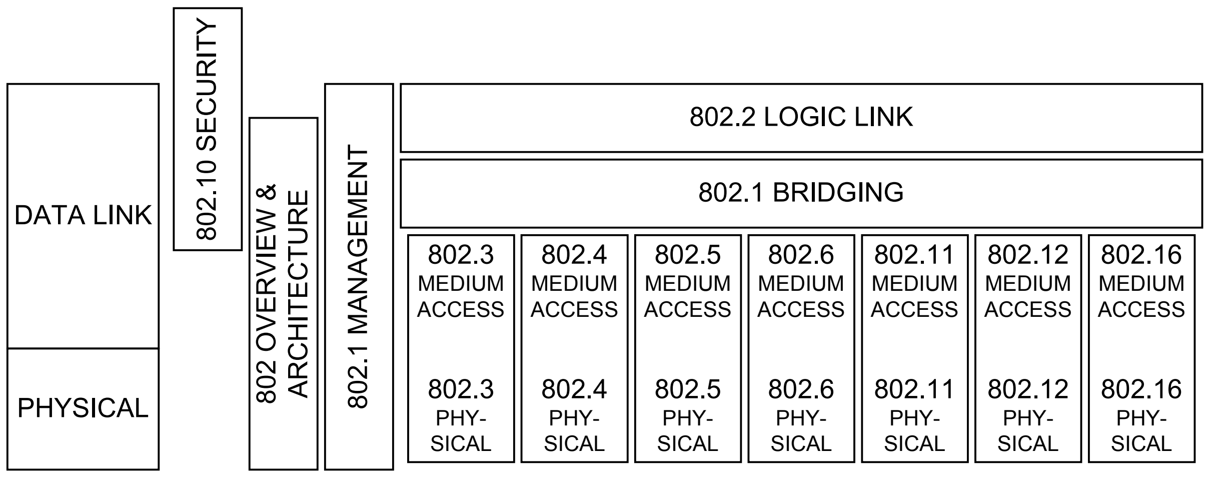


Figure 2: Relationship within the family of IEEE 802 standards.

The standards 802.3-6, 11, 12 and 16 define different medium access technologies and their associated media. As an example, 802.3 defines the access method using Carrier Sense Multiple Access with Collision Detection (CSMA/CD), while e.g. 802.11 defines the access method using Wireless LAN.

2.1.3 The IEEE 802.3 Standard

This standard for LANs employing CSMA/CD as access method supports bit rates from 1 Mbps to 1’000 Mbps. The focus in this report is on 1’000 Mbps systems using copper cabling as physical medium.

The first edition of the 802.3 standard was approved by IEEE itself in 1983. Since then, new parts have been added and old parts revised. Every change to the standard has been given a name, e.g. IEEE 802.3ab. The letters at the end refers to a specific clause that was added or a specific revision of the whole standard. Since the standard has been rather large, it is not often possible to state that a certain product is “IEEE 802.3 compliant”. Instead, the parts of the standard that have been implemented is targeted directly, e.g. “IEEE 802.3ab compliant”.

2.1.4 The IEEE 802.3 Architectural Model

The architecture of IEEE 802.3 corresponds closely to the two lowest layers of the OSI/BR model as shown in figure 3.

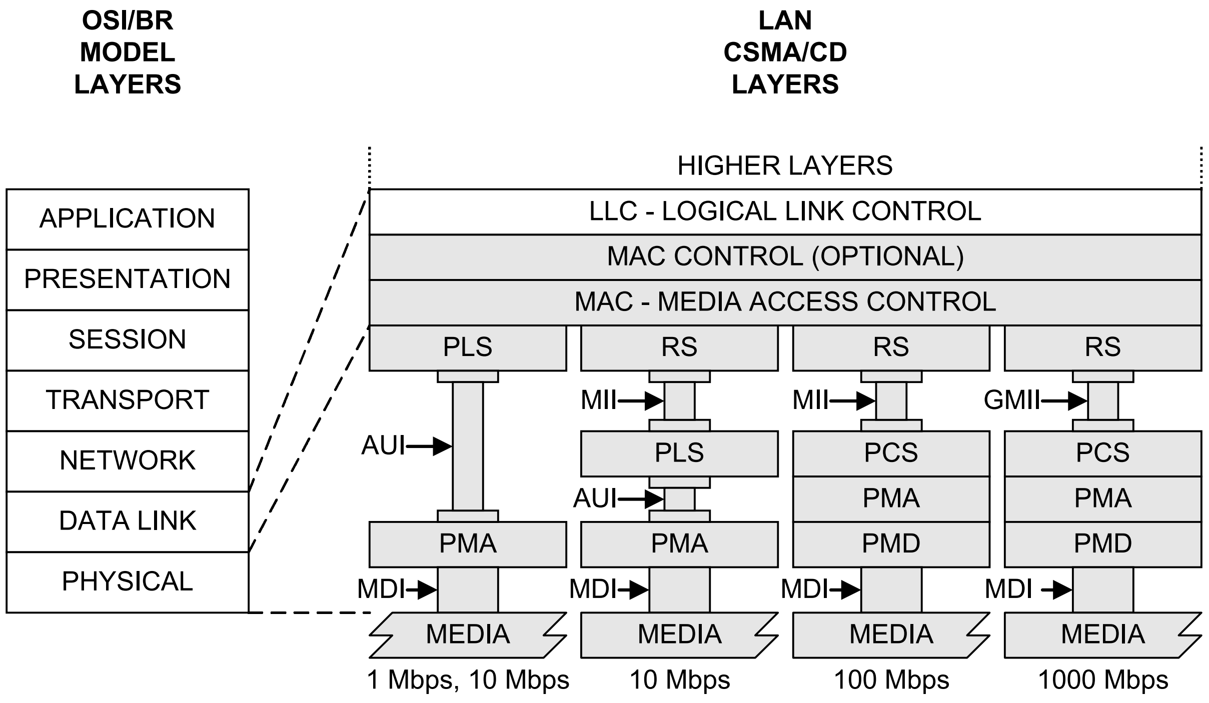


Figure 3: The LAN standard’s relationship to the OSI/BR model.

The Data Link layer in the OSI/BR model is partitioned into three sublayers in the architecture in order to obtain maximum flexibility within the family of IEEE 802 standards. By doing this, various media access methods are allowed since the LLC sublayer is the same for all of them.

Each sublayer in the architectural model provides a set of services that the nearest implemented higher sublayer uses. Service is the gathering name for function, procedure and variable that is made public and used by other parts of a system but the part providing them.

A service is described in its most abstract form by a service primitive. There are two generic types of primitives, REQUEST and INDICATION. The REQUEST primitive is passed from a higher layer to a lower and INDICATION vice versa. The REQUEST primitive requests a service to be initiated while the INDICATION primitive indicates an event.

The architecture also defines five important compatibility interfaces (MII, GMII, AUI, MDI and, not shown in figure 3, TBI). All interfaces, but MDI, are optional and in this study, only MII and GMII are of interest. MII and GMII are further explained in sections 2.3.1 and 2.3.2.

When implemented in hardware the typical solution until today has been to implement the Physical layer except Reconciliation sublayer, RS, in one device, often referred to as a PHY device, and the Data Link layer together with RS into another, often referred to as a MAC device. The MAC device also typically incorporates a bus controller suitable for the intended host system, e.g. PCI if implemented for use in a PC. Another solution that has become more common is to implement both the Data Link layer and the Physical layer together in a single device in order to save space, power and cut costs.

When a two-device constellation is used, the two devices are connected to each other via the MII and/or GMII. A benefit with separate MAC and PHY devices is that one MAC device can be connected to several PHY devices. By doing that the bandwidth can be increased since the links form a single link as seen by the LLC sublayer. This type of link is referred to as aggregated link.

In this work, a PHY device will be used and the FPGA will contain the RS and higher sublayers.

2.2 Referenced Sublayers in IEEE 802.3

Two of the sublayers defined in IEEE 802.3 are referenced in this study. It is the RS and MAC sublayer. Figure 4 shows the services provided by each sublayer. The provider of a service is always the sublayer beneath the arrow. The arrow points from the calling sublayer, e.g. the service collisionDetect is provided by the RS and indicates to the MAC sublayer when a collision has been detected. Another example is the service TransmitBit, also provided by RS, which the MAC sublayer uses to request the transmission of frames.

In figure 4, the optional MAC Control sublayer has been implemented. If this sublayer were not to be implemented, the two service primitives denoted MA_CONTROL.* would not be present. The direction of the arrows for service primitives points upwards if it is of type indication and downwards if it is of type request.

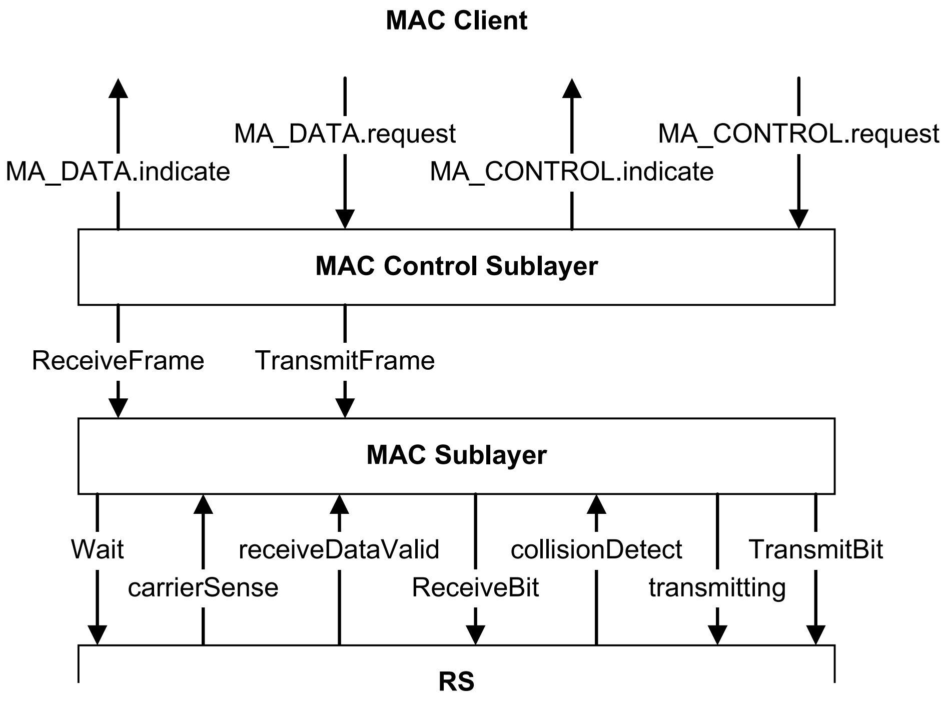


Figure 4: Service primitives' and services' relationships.

The services for the MAC sublayer and RS are further described in sections 2.2.1 and 2.2.2 respectively.

2.2.1 MAC Service Specification

The MAC sublayer performs the access control for the shared media (i.e. the physical cable). Besides the access control it also performs, among other things, checksum generation for outgoing frames and checks incoming ones, assemble and dissemble frames.

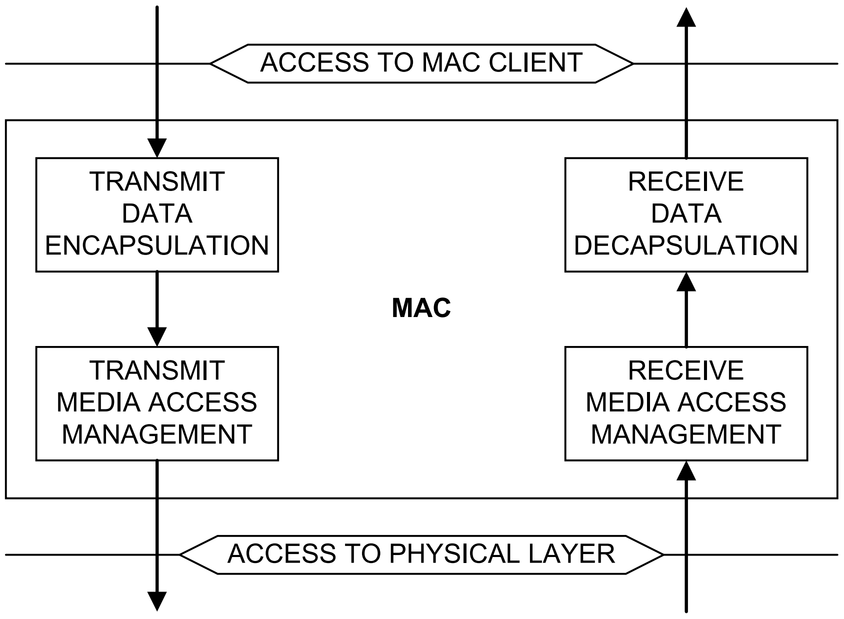

ACCESS TO MAC CLIENT
TRANSMIT DATA ENCAPSULATION
RECEIVE DATA DECAPSULATION
MAC
TRANSMIT MEDIA ACCESS MANAGEMENT
RECEIVE MEDIA ACCESS MANAGEMENT
ACCESS TO PHYSICAL LAYER

Figure 5: MAC functions.

The services provided by the MAC sublayer allow the MAC client entity to exchange LLC data with peer LLC sublayer entities.

The MAC sublayer is described in several levels of abstraction. The highest level is the specification of service primitives, notated as “MA_*” in figure 4. These service primitives are translated to services, e.g. the service primitive MA_DATA.request is an abstraction of the service TransmitFrame. The services provided by the MAC sublayer are presented in sections 2.2.1.1 and 2.2.1.2.

How the services are obtained is then given by a functional specification of the MAC sublayer presented in IEEE 802.3, Clause 4.2.8. This functional specification, written in a Pascal-like code, is later used in chapter 4 where the implementation of the sublayer is performed.

2.2.1.1 TransmitFrame

The MAC client (i.e. MAC Control or LLC sublayer) transmits a frame by invoking `TransmitFrame` (IEEE 802.3, Clause 4.3.2).

```
function TransmitFrame(
              var destinationParam: AddressValue;
              var sourceParam: AddressValue;
              var lengthOrTypeParam:LengthOrTypeValue;
              var dataParam: DataValue
              ): TransmitStatus;

type TransmitStatus = (transmitDisabled, transmitOk,
              excessiveCollisionError,
              lateCollisionErrorStatus);
```

The TransmitFrame operation is synchronous and lasts the entire attempt to transmit the whole frame and when finished, it reports success or failure via TransmitStatus.

TransmitStatus can also take the underlined values, but only if Layer Management is implemented.

2.2.1.2 ReceiveFrame

The MAC client (i.e. MAC Control or LLC sublayer) accepts to receive a frame by invoking ReceiveFrame (IEEE 802.3, Clause 4.3.2).

```
function ReceiveFrame(
              var destinationParam: AddressValue;
              var sourceParam: AddressValue;
              var lengthOrTypeParam: LengthOrTypeValue;
              var dataParam: DataValue
              ): ReceiveStatus;

type ReceiveStatus = (receiveDisabled, receiveOk,
              frameTooLong, frameCheckError, lengthError,
              alignmentError);
```

The ReceiveFrame operation is synchronous and lasts the entire attempt to receive the whole frame and when finished, it reports success or failure via ReceiveStatus.

ReceiveStatus can also take the underlined values, but only if Layer Management is implemented.

2.2.2 RS Service Specification

The interface through which the MAC sublayer uses the facilities of the Physical layer consists of a function, a pair of procedures and four Boolean variables.

The services that RS provides are defined by the service primitives for the Physical Layer Signaling (PLS) sublayer, notated "PLS_*" in figure 6. The RS maps these service primitives to electrical signals that form the interfaces MII and GMII.

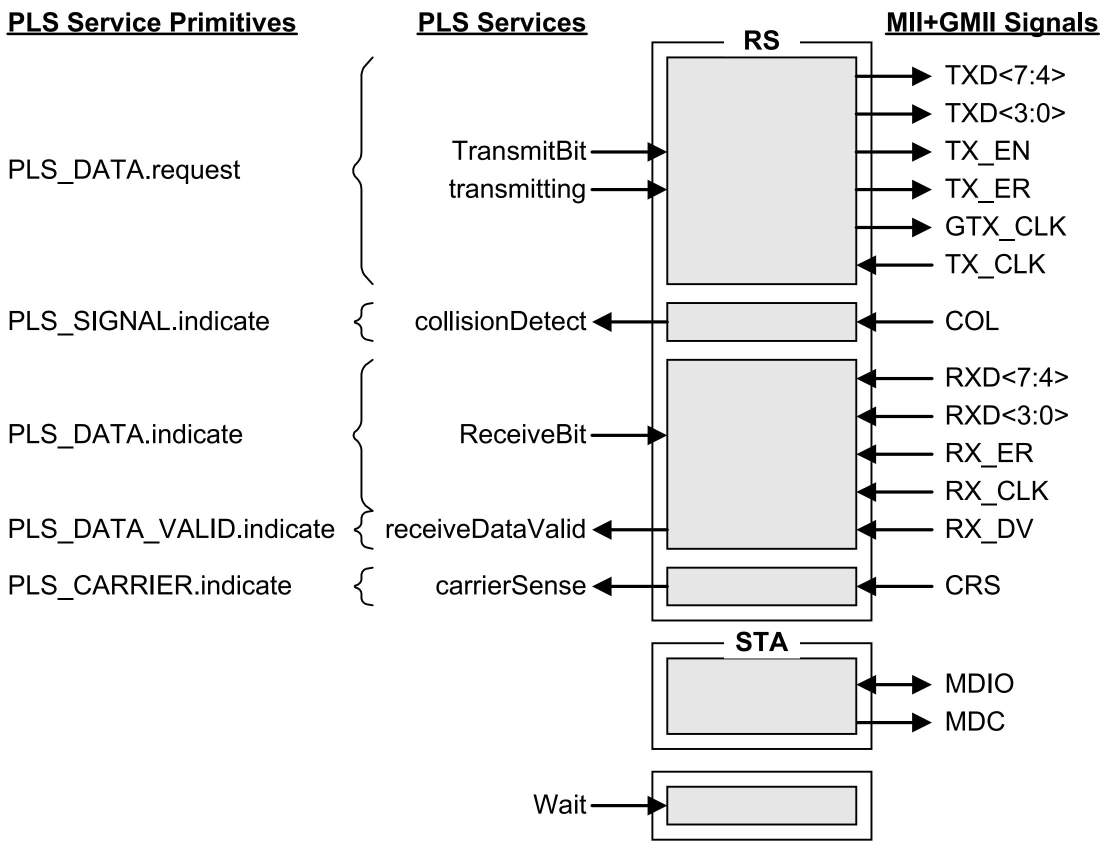


Figure 6: RS services' and STA's connections to MII/GMII.

2.2.2.1 carrierSense

The variable `carrierSense` signals to the MAC sublayer if there is any activity or not on the physical medium.

```
var carrierSense: Boolean;
```

The variable is set to true immediately upon detection of activity and set to false as soon as the activity ceases. The transitions of the variable are not synchronous with any of the clocks defined.

The behavior of the variable is only specified for half duplex mode, meaning that it shall be omitted in full duplex mode.

2.2.2.2 receiveDataValid

The variable `receiveDataValid` signals to the MAC sublayer if there is data being received by the physical layer.

```
var receiveDataValid: Boolean;
```

When the variable `receiveDataValid` is set to true by the physical layer, the

MAC sublayer shall immediately begin receiving the incoming data by using the function `ReceiveBit`. The function will be called repeatedly until `receiveDataValid` becomes false.

2.2.2.3 collisionDetect

The variable `collisionDetect` signals to the MAC sublayer if a collision occurs in the physical medium.

```
var collisionDetect: Boolean;
```

The variable `collisionDetect` remains true during the duration of the collision. It can only be true during transmission, not during reception.

The behavior of the variable is only specified for half duplex mode, meaning that it shall be omitted in full duplex mode.

2.2.2.4 transmitting

The variable `transmitting` signals to the Physical sublayer if data is being transmitted.

```
var transmitting: Boolean;
```

Prior to the first bit of data to be transmitted is passed from the MAC sublayer to the Physical layer, `transmitting` is set to true to inform that a stream of bits will be presented via the procedure `TransmitBit`.

When the last bit has been transferred, `transmitting` is set to false in order to indicate the end of the frame.

2.2.2.5 TransmitBit

During transmission, the outgoing frame is passed bit by bit to the Physical layer by repeated use of the procedure `TransmitBit`.

```
procedure TransmitBit(var bitParam: PhysicalBit);
```

Each invocation of the procedure passes one new bit and the duration of the operation is one bit time. Prior to the first invocation of the procedure, the variable `transmitting` has to be set to true.

A `PhysicalBit`, when transmitting, is a bit that can take the values `0`, `1`, `extensionBit` or `extensionErrorBit`. An `extensionBit` is a non-data value used for carrier extension and interframe during bursts.

An `extensionErrorBit` is a non-data value used to jam during carrier extension.

2.2.2.6 ReceiveBit

During reception, the incoming frame is passed bit by bit to MAC sublayer by repeated use of the function `ReceiveBit`.

```
function ReceiveBit( ): PhysicalBit;
```

Each invocation of the function passes one new bit and the duration of the operation is one bit time. The function is invoked every time that `receiveDataValid` is set to true.

A `PhysicalBit`, when receiving, is a bit that can take the values `0`, `1` or `extensionBit`. An `extensionBit` is a non-data value used for carrier extension and interframe during bursts.

2.2.2.7 Wait

The procedure `Wait` waits for a specified number of bit times, which allows the MAC sublayer to measure time in units of bit times.

```
procedure Wait(var bitTimes: Integer);
```

A bit time is the period it would takes to transmit one bit on the physical medium, e.g. when sending in 100 Mbps 1 bit time is 10 ns.

2.3 Referenced Interfaces in IEEE 802.3

There are totally five electrical interfaces defined within IEEE 802.3. Only two of them are relevant for this study and they are MII and GMII. Their location in the architecture can be seen in figure 3.

On many counts, the two interfaces are identical. The difference is the bit width of the data signals, and the encoding of the same is extended for GMII. The transmit clock signal also differ between MII and GMII. This allows the interfaces to be merged together, which often is done in integrated circuits that provide both a MII and a GMII interface.

The MII and GMII interfaces are further describe in sections 2.3.1 and 2.3.2 respectively, where the latter one only describes the signals in the cases where the definition differs from the one for MII.

2.3.1 MII

The interface through which the PHY device communicates with higher layers at speeds of 10 or 100 Mbps is called MII (fig. 7).

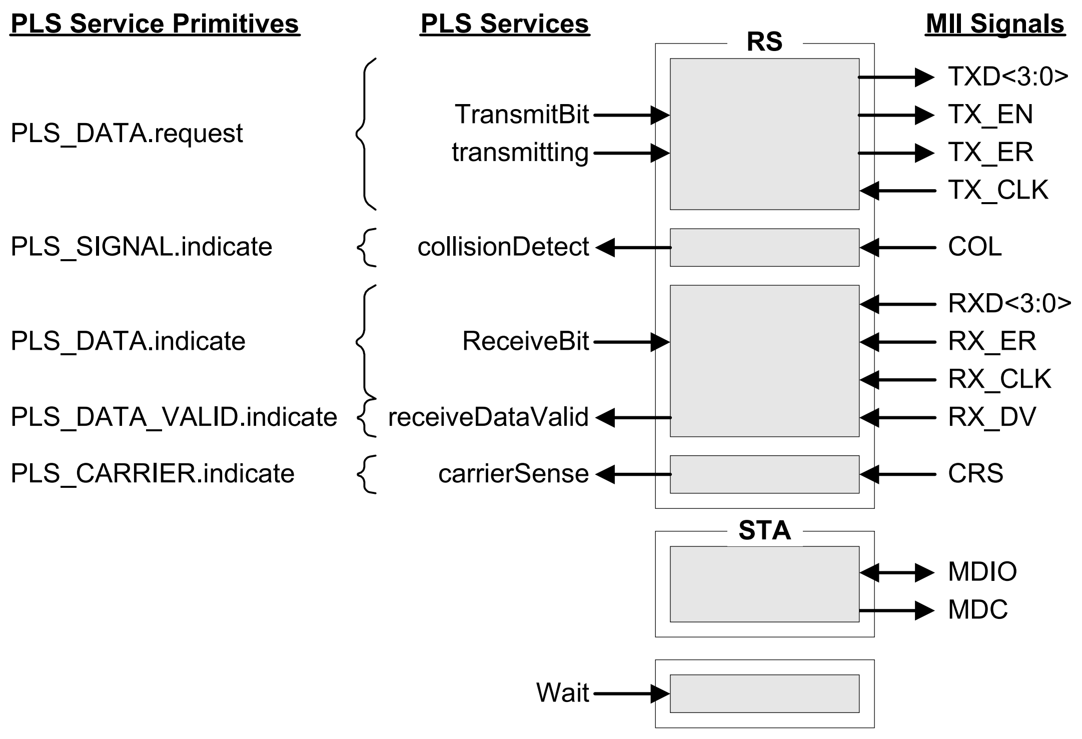


Figure 7: RS services' and STA's connections to MII.

The grayed boxes in figure 7 indicate domains for which all associated signals and services have a specified timing relationship in the standard. There is no specified timing relationship between any of the boxes.

2.3.1.1 TX_CLK (transmit clock)

`TX_CLK` is a continuous clock, sourced by the PHY, that provides the timing reference for the transfer of the `TXD`, `TX_EN` and `TX_ER` signals from the RS to the PHY.

The clock will have a frequency equal to one-fourth of the data rate (i.e. 25 MHz for 100 Mbps).

2.3.1.2 TX_EN (transmit enable)

`TX_EN` is driven by the RS to indicate that nibbles for transmission is presented on `TXD` and transitions synchronously with `TX_CLK`. `TX_EN` shall be asserted from the first nibble of Preamble through the whole frame and then de-asserted.

2.3.1.3 TX_ER (transmit coding error)

`TX_ER` is driven by the RS to indicate to the PHY that the RS, or any higher layer, has encountered problems during transmission and the frame currently being transmitted is not correct. `TX_ER` transitions synchronously to `TX_CLK`.

2.3.1.4 TXD (transmit data)

`TXD<3:0>` is a bundle of four data signals, where `TXD<0>` is the least significant bit. `TXD` is driven by the RS and transitions synchronously to `TX_CLK`. For each `TX_CLK` period in which `TX_EN` is asserted, `TXD` is accepted for transmission by the PHY.

Table 1: Permissible encoding of TX_EN, TX_ER and TXD

TX_EN	TX_ER	TXD<3:0>	Indication
0	0	0 – F	Normal inter-frame
0	1	0 – F	Reserved
1	0	0 – F	Normal data transmission
1	1	0 – F	Transmit error propagation

2.3.1.5 RX_CLK (receive clock)

`RX_CLK` is a continuous clock, sourced by the PHY, that provides the timing reference for the transfer of the `RXD`, `RX_DV` and `RX_ER` signals from the PHY to the RS.

The clock will have a frequency equal to one-fourth of the data rate (i.e. 25 MHz for 100 Mbps).

2.3.1.6 RX_DV (receive data valid)

RX_DV is driven by the PHY to indicate that the PHY is presenting data on RXD<3:0> and transitions synchronously with RX_CLK. RX_DV shall be asserted continuously from the first recovered nibble (starting no later than SFD) through the whole frame and then de-asserted.

2.3.1.7 RX_ER (receive error)

RX_ER is driven by the PHY to indicate to the RS that an error was detected somewhere in the frame presently being transferred over the MII. RX_ER transitions synchronously to RX_CLK.

2.3.1.8 RXD (receive data)

RXD<3:0> is a bundle of four data signals, where RXD<0> is the least significant bit. RXD is driven by the PHY and transitions synchronously to RX_CLK. For each RX_CLK period where RX_DV is asserted, PHY transfers a nibble of recovered data bits to the RS.

During a “Normal data reception”, the service ReceiveBit will transfer a “0” or a “1”. During a “False Carrier indication”, the service ReceiveBit will transfer an extensionBit (sec. 2.2.2.6). Table 2 summarizes the permissible encoding of RX_DV, RX_ER and RXD.

Table 2: Permissible encoding of RX_DV, RX_ER and RXD

RX_DV	RX_ER	RXD<3:0>	Indication
0	0	0 – F	Normal inter-frame
0	1	0	Normal inter-frame
0	1	1 – D	Reserved
0	1	E	False Carrier indication
0	1	F	Reserved
1	0	0 – F	Normal data reception
1	1	0 – F	Data reception error

2.3.1.9 CRS (carrier sense)

CRS is driven by the PHY and asserted when there is activity on the medium. In other cases, CRS will be de-asserted. The transition of CRS is not required to be synchronous with either TX_CLK or RX_CLK.

The behavior of CRS is unspecified for full duplex operation.

2.3.1.10 COL (collision detect)

`COL` is driven by the PHY and asserted as long as there is a collision on the transmit medium, otherwise `COL` is de-asserted. The transition of `COL` is not required to be synchronous with either `TX_CLK` or `RX_CLK`.

The behavior of `COL` is unspecified for full duplex operation.

2.3.1.11 MDC (management data clock)

`MDC` is sourced by the Station Management entity (STA) and used as the timing reference for the signal `MDIO`. Further, `MDC` is an aperiodic clock signal that has no maximum high or low times. This clock is not related to the clocks `TX_CLK` or `RX_CLK`.

2.3.1.12 MDIO (management data input/output)

`MDIO` is a bidirectional signal between the PHY and the STA. It is used to transfer control and status information between the PHY and the STA. Control information is driven by the STA synchronously with respect to `MDC` and is sampled synchronously by the PHY. Status information is driven by the PHY synchronously with respect to `MDC` and is sampled synchronously by the STA.

2.3.2 GMII

The interface through which the PHY device communicates with higher layers at speeds of 1’000 Mbps is called GMII (fig. 8).

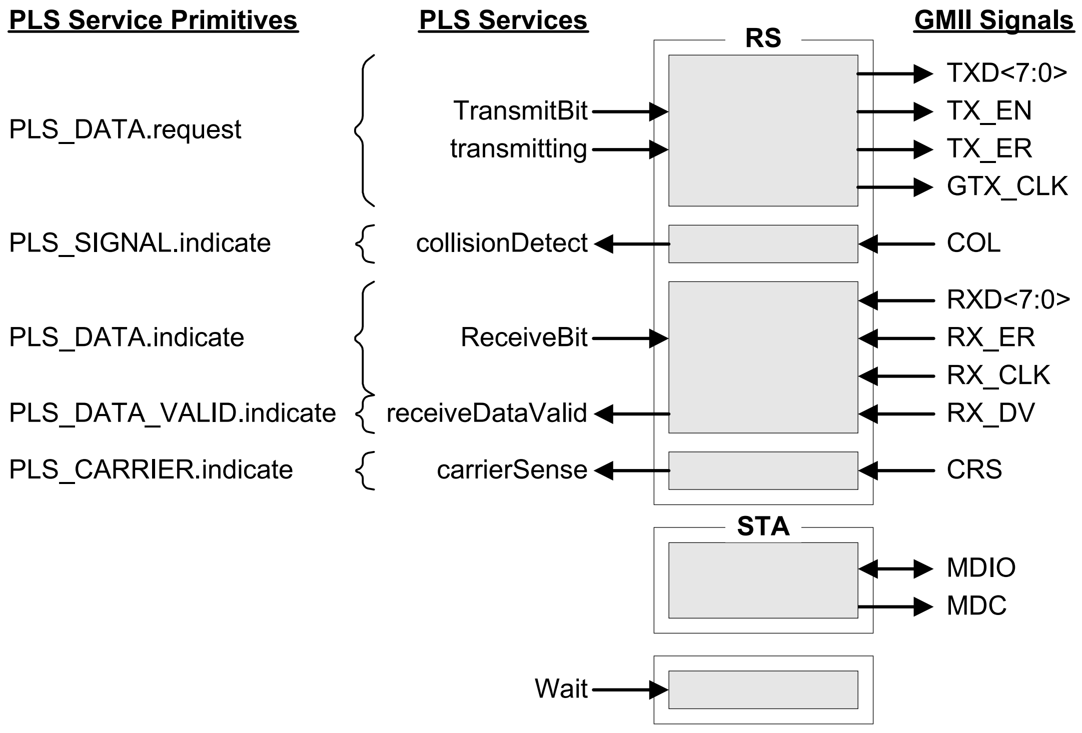


Figure 8: RS services’ and STA’s connections to GMII.

The grayed boxes in figure 8 indicate domains for which all associated signals and services have a specified timing relationship in the standard. There is no specified timing relationship between any of the boxes.

2.3.2.1 GTX_CLK (transmit clock)

`GTX_CLK` is a continuous clock, sourced by RS, that provides the timing reference for the transfer of the `TXD`, `TX_EN` and `TX_ER` signals from RS to the PHY.

The clock will have frequency equal to one-eighth of the data rate (i.e. 125 MHz for 1’000 Mbps).

2.3.2.2 TX_EN (transmit enable)

(Same behavior as for MII, see section 2.3.1.2)

2.3.2.3 TX_ER (transmit error)

(Same behavior as for MII, see section 2.3.1.3)

2.3.2.4 TXD (transmit data)

`TXD<7:0>` is a bundle of eight data signals, where `TXD<0>` is the least significant bit. `TXD` is driven by the RS and transitions synchronously to `GTX_CLK`. For each `GTX_CLK` period in which `TX_EN` is asserted, `TXD` is accepted for transmission by the PHY.

During a Normal data transmission, the service TransmitBit will transfer a "`0`" or a "`1`". During a "Carrier Extend" or a "Carrier Extend error", the service `TransmitBit` will transfer respectively an `extensionBit` or an `extensionErrorBit` (sec. 2.2.2.5). Table 3 summarizes the permissible encoding of `TX_EN`, `TX_ER` and `TXD`.

Table 3: Permissible encoding of TX_EN, TX_ER and TXD

TX_EN	TX_ER	TXD<7:0>	Indication
0	0	00 – FF	Normal inter-frame
0	1	00 – 0E	Reserved
0	1	0F	Carrier Extend
0	1	10 – 1E	Reserved
0	1	1F	Carrier Extend Error
0	1	20 – FF	Reserved
1	0	00 – FF	Normal data transmission
1	1	00 – FF	Transmit error propagation

2.3.2.5 RX_CLK (receive clock)

`RX_CLK` is a continuous clock, sourced by the PHY, that provides the timing reference for the transfer of the `RX_DV`, `RXD` and `RX_ER` signals from the PHY to the RS.

The clock will have frequency equal to one-eighth of the data rate (i.e. 125 MHz for 1'000 Mbps).

2.3.2.6 RX_DV (receive data valid)

(Same behavior as for MII, see section 2.3.1.6)

2.3.2.7 RX_ER (receive error)

(Same behavior as for MII, see section 2.3.1.7)

2.3.2.8 RXD (receive data)

RXD<7:0> is a bundle of eight data signals, where RXD<0> is the least significant bit. RXD is driven by the PHY and transitions synchronously to RX_CLK. For each RX_CLK period where RX_DV is asserted, PHY transfers a nibble of recovered data bits to the RS.

During a "Normal data reception", the service ReceiveBit will transfer a "0" or a "1". During a "False Carrier indication", a "Carrier Extend" or a "Carrier Extend error", the service ReceiveBit will transfer an extensionBit (sec. 2.2.2.6). Table 4 summarizes the permissible encoding of RX_DV, RX_ER and RXD.

Table 4: Permissible encoding of RX_DV, RX_ER and RXD

RX_DV	RX_ER	RXD<7:0>	Indication
0	0	00 – FF	Normal inter-frame
0	1	00	Normal inter-frame
0	1	01 – 0D	Reserved
0	1	0E	False Carrier indication
0	1	0F	Carrier Extend
0	1	10 – 1E	Reserved
0	1	1F	Carrier Extend error
0	1	20 – FF	Reserved
1	0	00 – FF	Normal data reception
1	1	00 – FF	Data reception error

2.3.2.9 CRS (carrier sense)

(Same behavior as for MII, see section 2.3.1.9)

2.3.2.10 COL (collision detect)

(Same behavior as for MII, see section 2.3.1.10)

2.3.2.11 MDC (management data clock)

(Same behavior as for MII, see section 2.3.1.11)

2.3.2.12 MDIO (management data input/output)

(Same behavior as for MII, see section 2.3.1.12)

2.4 Referenced Protocols in IEEE 802.3

There are two protocols defined in IEEE 802.3 and both of them are used in this project. The first one is the MAC frame structure, which is the frame format used for transmission of data over the shared medium. Remember, one of the tasks for the MAC sublayer is to assemble the data and build a frame according to this structure.

The other frame structure is used for communication between a MAC and one or more connected PHYs.

2.4.1 MAC Frame Structure

The MAC frame structure is defined in IEEE 802.3, Clause 3. There exist several names for this frame format. One of the most common names is Ethernet II, which refers to the MAC frame when type interpretation is used upon the Length/Type field. Another common name is Ethernet frame or Ethernet 802.3 Raw frame. Both these names refer to a MAC frame when length interpretation is used upon the Length/Type field. The Length/Type field is further explained in sec. 2.4.1.4. Figure 9 shows the MAC frame structure and, depending of the value of the Length/Type field, it can be either an Ethernet or an Ethernet II frame.

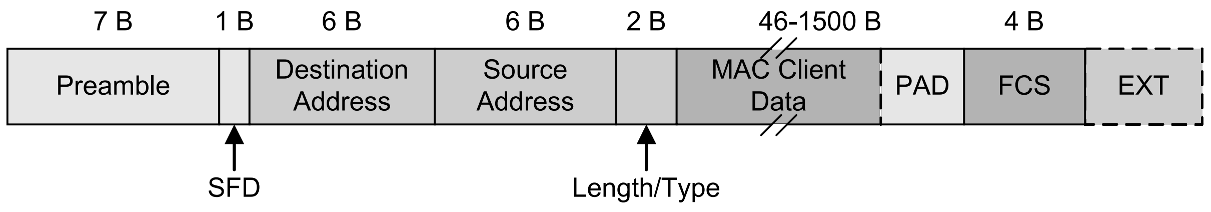


Figure 9: The IEEE 802.3 MAC frame structure.

The fields in the frame are transmitted from left to right. The byte(s) within each field are transmitted from left to right. Each byte in the frame, with the exception of the FCS, is transmitted with low-order bit first.

The extension field, EXT, is only needed for 1’000 Mbps half duplex operation.

2.4.1.1 Preamble field

The Preamble field is used for synchronization of the receiver with respect to the transmitter.

The preamble pattern is:
{10101010 10101010 10101010 10101010 10101010 10101010 10101010}

The bits are transmitted from left to right.

2.4.1.2 Start Frame Delimiter (SFD) field

This field denotes the start of the frame.

The pattern is:
{10101011}

The bits are transmitted from left to right.

2.4.1.3 Destination and Source Address fields

The address fields should be 48 bits. IEEE 802 states that one can use either 16- or 48-bit addresses but in IEEE 802.3, 16-bit addresses have been excluded.

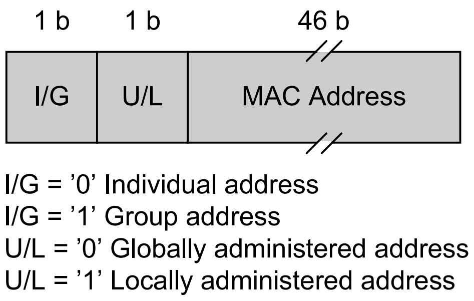

1 b
1 b
46 b
I/G
U/L
MAC Address
I/G = '0' Individual address
I/G = '1' Group address
U/L = '0' Globally administered address
U/L = '1' Locally administered address

Figure 10: Address field format.

The Destination Address (DA) field specifies the station(s) for which the frame is intended. While the Source Address (SA) field specifies the station that sends the frame.

If all (48) bits in the DA field are set to '1', a broadcast will be performed.

The last 46 bits of the address field contains the MAC Address (sometimes referred to as the Ethernet Address). The MAC Address is unique for each station and the allotment of addresses is managed by the IEEE Registration Authority, i.e. a manufacturer of network controllers has to get the MAC Addresses directly from IEEE. If two stations on the same network would use the same MAC Address, it would cause a collapse of the network.

Each byte in the Address field shall be transmitted with least significant bit first.

2.4.1.4 Length/Type field

This field has two meanings depending of its value. The first byte in the field is the most significant one.

If the value is less than 0x0600 then the field indicates the number of MAC Client Data bytes contained in the subsequent data field of the frame (length interpretation).

If the value is greater than or equal to 0x0600 then the field indicates the type of the MAC Client Protocol (type interpretation).

2.4.1.5 Data and PAD fields

The data field contains a sequence of n bytes. Full data transparency is provided in the sense that any arbitrary sequence of byte values may appear in the data field up to a maximum number specified by the implementation of the standard that is used. A minimum frame size is required for correct operation and is specified by the particular implementation of the standard.

If necessary, the data field is extended by appending extra bits (that is, a pad) in units of bytes after the data field but prior to calculating and appending the FCS. The size of the pad, if any, is determined by the size of the data field supplied by the MAC client and the minimum frame size and address size parameters of the particular implementation.

The maximum size of the data field is determined by the maximum frame size and address size parameters of the particular implementation.

2.4.1.6 Frame Check Sequence (FCS) field

The FCS field contains a 32-bit checksum of the frame. The checksum is of the type cyclic redundancy check, CRC (in this case, CRC-32). This value is computed as a function of the contents of the Source Address, Destination Address, Length/Type, Data and PAD fields (that is, all fields except the preamble, SFD, FCS and EXT).

The 32 bits of the CRC value are placed in the FCS field so that the x^{31} term is the left most bit of the first byte, and the x^0 term is the right most bit of the last byte (the bits of the CRC are thus transmitted in the order x^{31}, x^{30}, …, x^1, x^0).

2.4.1.7 Extension (EXT) field

The Extension field follows the FCS field, and it is made up of a sequence of extension bits (described in sec. 2.3.2.4 and 2.3.2.8).

The contents of the Extension field are not included in the FCS computation.

The Extension field may have a length of greater than zero when sending in half duplex mode above 100 Mbps. The length of the Extension field will be zero under all other conditions.

2.4.2 Management Frame Structure

Frames transmitted on the MII/GMII Management Interface shall have the frame structure shown in figure 11. The order of bit transmission shall be from left to right.

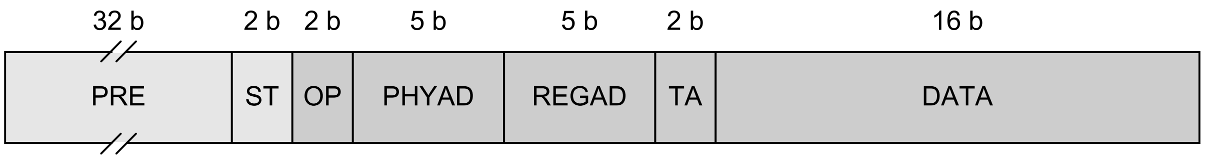


Figure 11: Management frame structure.

2.4.2.1 PRE (preamble)

At the beginning of each transaction, the STA shall send a sequence of 32 contiguous logic one bits on MDIO in order to establish the synchronization with the PHY.

If every PHY that is connected to the MAC is able to accept frames that are not preceded by the preamble, the STA may suppress the generation of it.

2.4.2.2 ST (start of frame)

The start of the frame is indicated by a “01” pattern.

2.4.2.3 OP (operation code)

When STA shall set a bit in the register of the PHY, a write transaction will be carried out, which is indicated by a “10” pattern. When STA whishes to read the value in the PHY’s status register, a read transaction is performed, which is indicated by a “01” pattern.

2.4.2.4 PHYAD (PHY Address)

The PHY Address is five bits, allowing 31 PHYs to be connected to one MAC. PHY address zero (“00000”) is a broadcast address that every connected PHY shall respond.

2.4.2.5 REGAD (Register Address)

The Register Address is five bits, allowing 32 individual registers to be addressed within each PHY. The address is transmitted with MSB first. The PHY’s registers are defined in IEEE 802.3, Clause 22.2.4.

2.4.2.6 TA (turnaround)

The turnaround is a 2-bit-time spacing between the REGAD field and the DATA field. During a write transaction, STA shall drive a logic one bit for the first bit time and a logic zero during the second. For a read transaction, both the STA and the PHY shall be in high-impedance state during the first bit time. During the second bit time, the PHY shall drive a logic zero bit.

2.4.2.7 DATA (data)

The data field is 16 bits. The first bit transmitted and received corresponds to bit 15 of the addressed register.

2.4.2.8 IDLE (IDLE condition)

The IDLE condition on MDIO is a high-impedance state.

CHAPTER

3

Design Methodology

Design methodology, as concept, can be interpreted in different ways. Some might think of it as a specific way of working through the design phase only (e.g. the commonly known “top-down” method sometimes used for software development). There is also a wider conception where one means the whole workflow from idea to a working product. In this chapter, the latter interpretation is used.

The design methodology describes the different actions taken under the development process. These actions can be carried out in different ways (and with different tools). This chapter describes a design methodology that is rather common today and which has been used in this work. The workflow is shown in figure 12. The choice of methodology in this study was not done exclusively, but the one implied by the selection of tools that were used.

An important thing to remember about all design methodologies is that the methodology shall be a tool in order to make the work easier, it should not become an end in itself.

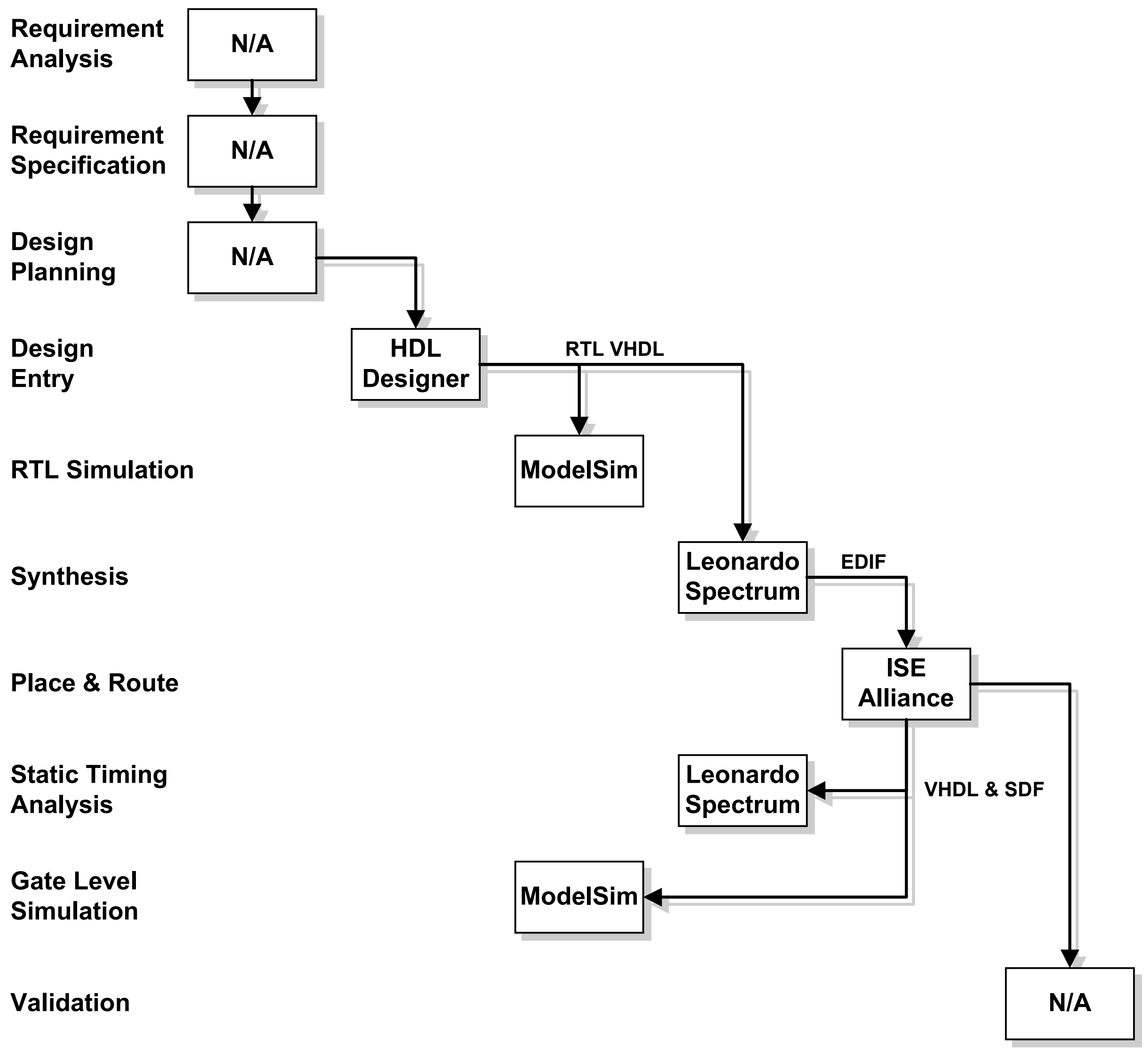


Figure 12: The workflow with addressed tools.

3.1 Requirement Analysis

This is the phase where one decides what the system actually should do. The environment surrounding the system is analyzed and the demands on the system are identified. The resulting document is written in prose:

> *"By mounting the network controller on a sensor, the sensor can transmit measurement data in gigabit rate."*

Producing the analysis document and specification document is an iterative process with many loops in order to cover all aspects of the system. To miss something in these first two steps can turn out to be very costly, which can be seen in table 1.

Table 5: Relative cost to fix an error

Phase	Cost ratio	Step (sections)
Requirements	1	3.1, 3.2
Design	3 – 6	3.3
Coding	10	3.4, 3.6, 3.7
Development testing	15 – 40	3.5, 3.8, 3.9
Validation	30 – 70	3.10
Operation	40 – 1000	

The figures in table 5 applies to software development but since we are dealing with hardware development in terms of programming HDL these figures can be considered to be relevant even for this case. One should also know that these figures are considered conservative [Boehm 1980].

3.2 Requirement Specification

The specification defines in natural language what the system is supposed to do. The difference between this document and the analysis document is that the components, actors, services etc. are identified (and named) and the demands on each of these parts are condensed from the previous document. During this process, missing parts can be analyzed and corrected. System components and their associated characteristics are marked in the requirement analysis document:

*"By mounting the **network controller** on a sensor, the sensor can transmit measurement data in **gigabit** rate."*

The requirement specification document is then written in the form:

*"The <system component> **shall** <required characteristic>"*

Using the fragment from the previous section would result in:

*"The network controller **shall** support gigabit Ethernet."*

There are both functional and non-functional requirements. A functional requirement specifies what the system should do, i.e. its functionality and features, of which the line above is an example.

A non-functional requirement specifies how the functionality is obtained, and under which constraints, e.g.:

*"The network controller's power dissipation **shall** be less than 4 W."*

The specification describes all the requirements regarding the system, for example standards that the system has to fulfill, timing and power constraints and all features it has to possess.

The specification document shall not discuss different implementation techniques (e.g. "*All state machines shall be of type Mealy*") or how to reach the solution (e.g. "*Use the design entry tool Renoir*").

3.3 Design Planning

At this point, the requirement specification is partitioned into functional blocks. The functional blocks are further broken down until a complete hierarchy is created were the function, interface and constraints of each block is well defined. At this stage, one has to choose how to implement state machines, memories etc.

This step in the design flow is very crucial. Making a bad planning can cause many problems later. For example: collate similar functions in the same functional block, be very careful when crossing clock domain boundaries etc. This is the step where the experienced designer takes use of his whole knowledge.

3.4 Design Entry

There are several ways how to entry the design. In this work HDL Designer (former Renoir) from Mentor Graphics has been used which allows the user to graphically enter the hierarchy, connect blocks and then enter HDL code in the blocks using a text-editor (e.g. Emacs), this method is called block diagram entry. HDL Designer can also generate HDL code from state diagrams, flow charts and truth tables.

When the design is completed, the HDL code is compiled, either towards the simulator or towards the synthesis tool.

3.5 RTL Simulation

First, interesting test cases must be identified and test vectors written describing those cases.

Then a behavioral model is created. This model is the "truth", in other words how the block being tested should behave if it is correct. The behavioral model can be automatically generated, e.g. by using Matlab when verifying an algorithm.

The test vectors and the behavioral model together form the so-called test bench. Its interface is identical with the block's being tested but mirrored. The test bench is then connected to the block and then compiled together towards the simulator. Note that it is only needed to write the interface in HDL, both the test vectors and behavioral model can be read from a file, which allows several tests to be carried out without the need of re-compiling the design in between.

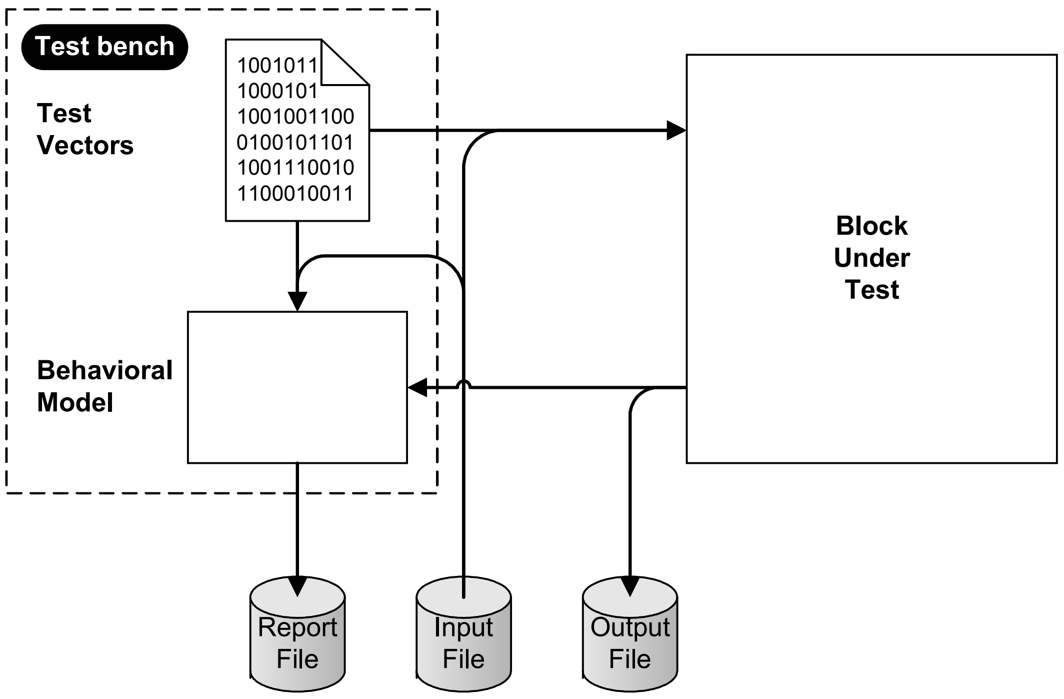


Figure 13: Test bench.

The test bench and its components are illustrated in figure 13 where the "Input File" represents an external test vector source. The "Output File" stores the output signals from behavioral model and the block being tested, this in order to easier discovers differences. A "Report File" can also be a good idea to generate, which logs messages from the behavioral model such as passed breakpoints etc.

Before RTL simulation, a functional simulation can be carried out. The difference between the two is that there is no delay element introduced in the latter one. When compiling the design for RTL simulation, delay elements are introduced that are of the same size and based on a typical wire length.

A simulator, in this case ModelSim by Model Technology, then uses the compiled data. The user both gets a graphical view of all signals and there transitions and, if certain commands are written in the test bench, textual messages in the form of warnings or passed breakpoints etc.

3.6 Synthesis

3.6.1 Choosing Target Device

If it has not been done earlier, it is time to choose which FPGA to use. Often it is hard to predict the size and speed grade needed, to get a feeling of these figures the simplest way is to synthesize one time using a large and fast device which gives a hint of the resources needed.

Besides size and speed other important issues to consider are of both functional and non-functional nature.

Functional characteristics:

- Hard blocks (e.g. processors, arithmetic units)
- Memory (size and type)

Non-functional characteristics:

- Package (e.g. size, heat-tolerance, BGA)
- Speed (e.g. number of clock drivers, maximum speed, routing resources)
- Size (number of CLBs)
- Number of I/Os
- I/O Signaling (e.g. TTL, LVDS)
- Supply voltage
- Power consumption
- Price

There are also some aspects that one should consider but which are not covered in above listings, e.g. the fact that a tool perhaps do not support a specific device. Mostly it is not possible for a company to have all design tools needed in order to cover all devices. There can also be a good idea to choose a smaller and slower device in a family of devices with the identical footprint and pinout to ease possible future upgrades. Support from the manufacturer can be a very important issue, especially if the device contains, for the developer, new functionality. If the developer have much experience of a certain device family, this also can be important to have in mind when selecting device in order to gain time in the project. Selecting an old device family can result in unnecessary costs since most manufacturers raise the price for older families when a new one is launched. Also, remember the fact that a family is not manufactured forever. Eventually it will be impossible to get a device, which can cause problem if the product is going to be manufactured for a long period.

3.6.2 Choosing Synthesize Method

The synthesis step begins within the design entry tool HDL Designer where the HDL is generated and compiled. In this study the synthesis tool Leonardo Spectrum from Exemplar Logic has been used. The synthesis tool can be invoked from the HDL Designer and a pre-optimization is carried out. The user must decide which device to use, which type of optimization to perform (area or delay) and whether the hierarchy should be preserved or flattened. There is also a third alternative, “auto”. The “auto” choice leaves the system to decide whether to preserve or flatten the hierarchy.

It is worth mentioning some words regarding the hierarchy options. If a single block is to be synthesized, this option has of course no impact on the result but with several blocks and bad design practice (i.e. not registered outputs) it does. When selecting “flatten” the block borders are removed and the whole design is treated as one single block. The optimization will not be very good if the design is big since the algorithms have difficulties dealing with large designs. Instead one have to use “preserve hierarchy” where each block will be optimized individually and then treated as black boxes when combined. This way of optimizing is much faster than flattening the design.

What about “bad design practice” and “preserve hierarchy”? If the outputs of a block are not registered but instead consists of logic and then connected to a second block that has logic on its inputs, the two logic nets will be optimized separately and later, when it is routed, the timing will be wretched. The resulting optimization is illustrated in figure 14, where a white cloud illustrates logic before optimization and a black cloud after optimization.

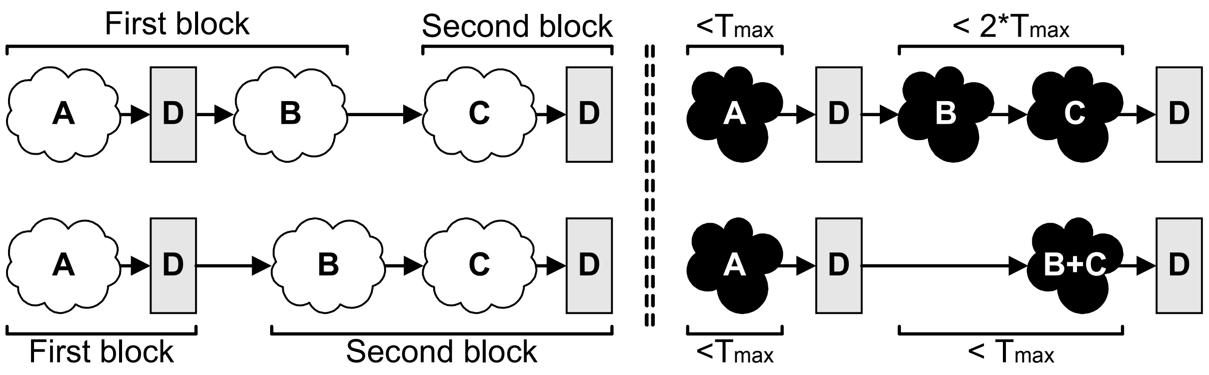


Figure 14: Bad and good design practice when preserve hierarchy is used.

In the upper case the maximum clock frequency might be halved since the logic nets B and C will not be optimized together when preserve hierarchy is used. This problem would be avoided if the design was flattened and/or all outputs of all blocks were registered.

Leonardo Spectrum allows the user to set several timing criteria, e.g. false path (a signal that does not has to fulfill the timing requirement) and multiple cycle path (a signal that has several clock cycles before it has to be stable). These options may be very useful when synthesizing a design but has not been used in this study.

The synthesis process is much of a “trial and error” one because of all degrees of freedom (selecting device, area/delay optimization, preserve or flatten hierarchy etc.). The synthesis tool also estimates the size and performance of the final implementation.

3.7 Place & Route

The outcome from the synthesis step is forwarded into the place & route tool ISE Alliance. Like in the preceding step, there are several choices how the action shall be performed.

When the place & route step is finished the exact figures of size, performance etc. can be found in the generated report files. The figures are very close to reality and can differ a lot from the estimations done by the synthesis tool.

3.8 Static Timing Analysis

The place & route tool computes the delay and time skew for all paths, which gives the maximum possible clock-frequency under worst-case condition.

If the block does not manage the timing constraints, the failing path can be analyzed in the synthesis tool Leonardo Spectrum. To solve the problem either the block can be modified or perform the synthesis and Place & route step with other preferences and/or constraints. Later versions of ISE Alliance contain a "Place & route Assistant" that gives suggestions to improvements when the timing constraints are not met.

3.9 Gate Level Simulation

This simulation can be carried out after that the synthesized design has gone through the place & route step. The test bench from the RTL simulation can be re-used. The difference from the earlier RTL simulation is that the average delay in the transmission lines is changed to exact figures since the delay in each line now is known.

In practice, a new architecture of the block is generated by the place & route tool and then imported into HDL Designer. This result becomes, that one entity is described by two architectures, the original architecture written in HDL and the one generated from the place & route tool. By selecting the latter architecture and then compiling the block towards ModelSim, the gate-level simulation can be carried out.

3.10 Validation

As was pointed out in section 3.2 (Requirement Specification), all of the requirements shall be possible to validate. The validation process takes place in-board under realistic conditions.

The functional requirements are validated by comparing the behavior of the system with the one specified and with the results from the gate-level simulation.

The non-functional requirements are validated by measurements of e.g. power dissipation, supply voltages etc.

CHAPTER

4

Implementation

This chapter describes how each step presented in chapter 3 was carried out in this project.

Since a specific standard is used and the focus is only set on the core function (i.e. the MAC), the first two steps in the design methodology were omitted.

The last two steps, validation and gate level simulation, were also omitted since no hardware was used in this work. The preceding step, gate level simulation, was not performed since the step turned out to be quite time consuming.

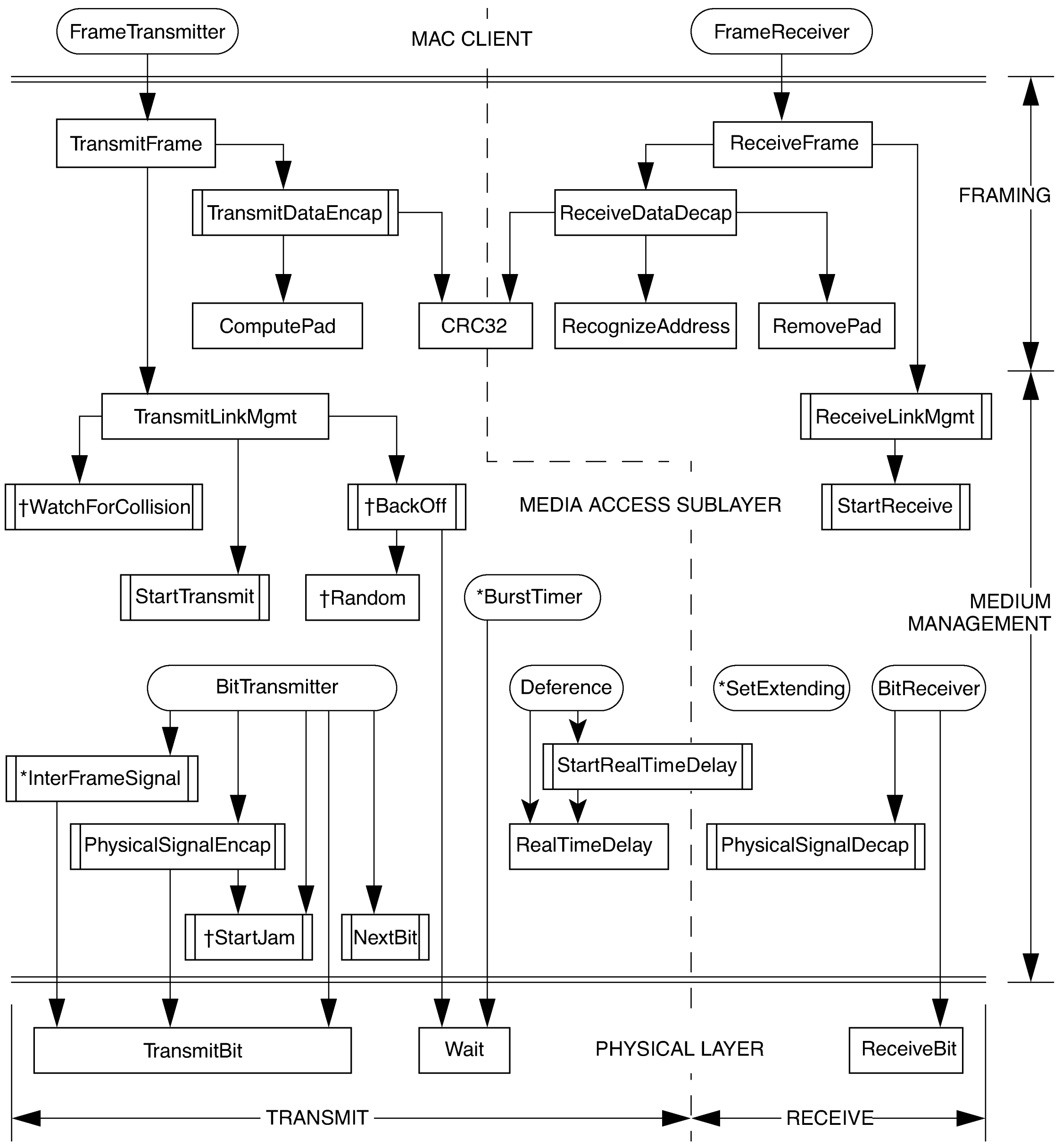
An extra step was inserted within the Design Planning where an analysis was carried out in order to minimize the number of blocks necessary to implement in the project.

4.1 Design Planning

In this planning clearness has been prioritized before reaching an optimal design. Clearness has been reached by adopting as much as possible of the structure used in the standard. The reason was simply to ease for future readers to take use of the conclusions made in this work. If optimal design should be the target, there is a risk that the structure of the implementation would differ a lot from the one in the standard, which would force the reader to learn two different descriptions of the same system.

4.1.1 Partitioning of the Standard

The precise definition of the MAC in the standard is written in a Pascal-like language. It is therefore necessary to port the Pascal-like code to the HDL language (e.g. VHDL). The standard assumes the presence of a nearly infinitely fast processor, which can handle parallel processes as well, executing the Pascal program. The need of parallelism is the main reason why to use an FPGA since it is parallel by nature.



† Not applicable to full duplex operation
* Applicable only to half duplex operation at > 100 Mb/s

Figure 15: Relationship among CSMA/CD processes, procedures and functions as defined in standard.

Like VHDL, the Pascal-like language allows the declaration of process, function and procedure. A short repetition regarding just mentioned terms in the case of VHDL code:

- Process: Executed sequentially
  Processes are executed in parallel.
- Function: Executed sequentially
  Returns one value
- Procedure: Executed sequentially
  Returns zero or more values
  Can change its input arguments

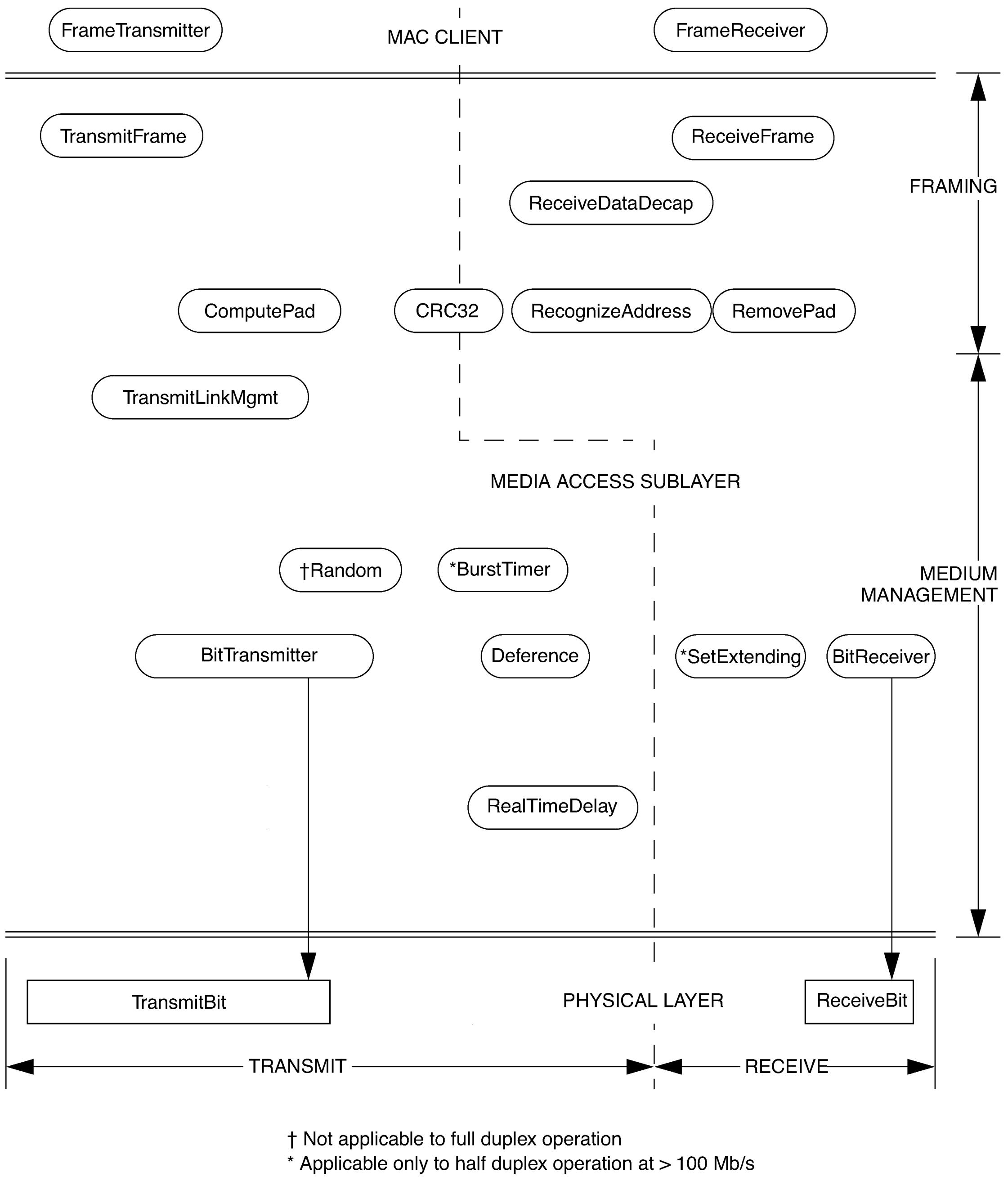


Figure 16: Relationship among CSMA/CD processes in the implementation.

When porting, the following rules have been used:

- Processes in the standard are implemented as processes.
- Functions in the standard are implemented as processes that are idling until called, then executed one time and then returns to idling.
- Procedures in the standard are implemented by being incorporated in the processes and/or functions that take use of it.

The application of above rules will result in the structure presented in figure 16. The original structure in the standard is shown in figure 15.

The data path in the standard is bit-oriented but in the implementation, it is byte-oriented. The reason is simple; since the maximum bit-rate is 1000 Mbps choosing a bit-oriented data-path would result in a clock-frequency of 1 GHz, instead by choosing byte-oriented data-path the clock-frequency is lowered to 125 MHz. This is in line with the standards suggestion of implementing the data path on bit, byte or word basis [IEEE 802.3, Clause 4.2.2.1.c].

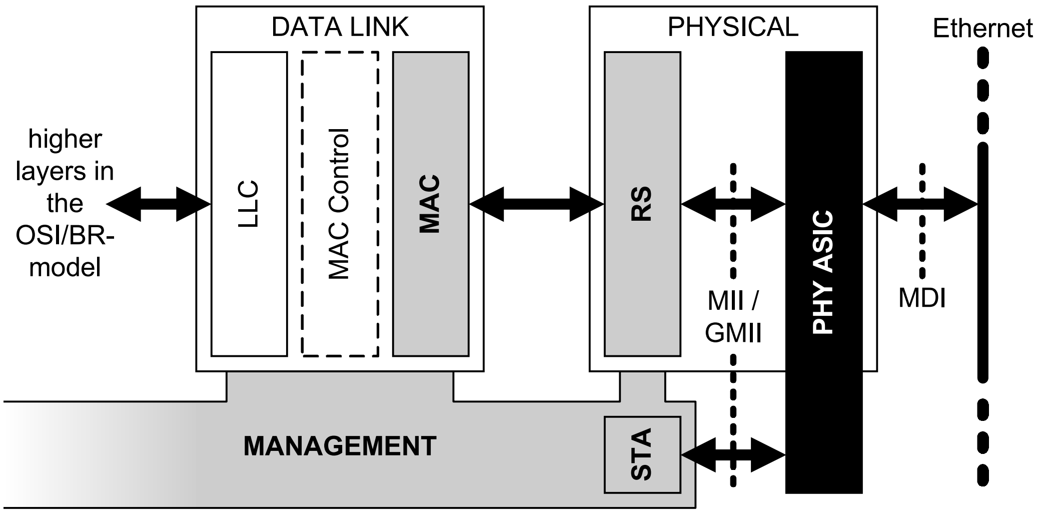


Figure 17: The architecture of the implementation.

The complete implementation, shown in figure 17, is a direct mapping of the OSI/BR model to the implementation. Grayed areas are those that were implemented. The PHY ASIC implements the remaining sublayers that were not implemented in the FPGA.

There is no focus on the higher layers since these are not considered difficult to implement because the data width can be expanded in order to lower the clock-frequency. This, however, is not complete true. If the implementation should support the commonly used protocol TCP (often referred to as TCP/IP) there may be problem, since the protocol stack requires much resources. Instead, TCP frames should be forwarded to an external unit for further processing. This implementation is done having another often-used protocol in mind, UDP, which does not contain a stack like the one used in TCP.

Beneath the MAC sublayer (when referring to the OSI/BR model), RS is located. RS provides the major part of the MII/GMII interface except the MII/GMII Management Interface, which is provided by the Station Management entity, STA. The blocks RS and STA are the only parts of the Physical layer that are implemented in the FPGA. These blocks are further explained in sections 4.1.3 and 4.1.4 respectively.

The MAC sublayer consists of five blocks (fig. 18), the transmitter (*TxMAC*), the receiver (*RxMAC*), the Management Information Base (*MIB*) and the buffer managers *TxBufMgr* and *RxBufMgr*.

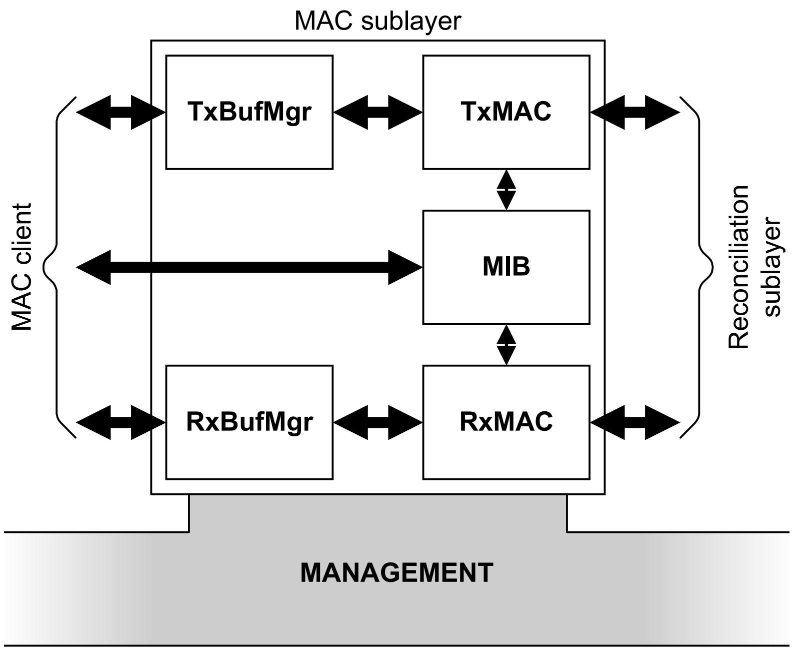


Figure 18: Implementation of MAC sublayer.

The *TxMAC* is further explained in sec. 4.1.2. The *MIB* provides Layer Management [IEEE 802.3, Clause 5]. The *MIB* itself is not a critical part of the system and thereby not included in the project. The same yields for the buffer managers, which manages the transmit and receive queues.

The implementation of the MAC sublayer contains three different clock-domains. *TxMAC* and *RxMAC* constitute one domain each and the rest of the blocks as well as the management part belong to the same. By having several clock-domains, the complexity increases and extra logic has to be inserted in order to be able to cross the domains. This is also discussed in sec. 4.1.2.3.

4.1.2 Precise Design: TxMAC

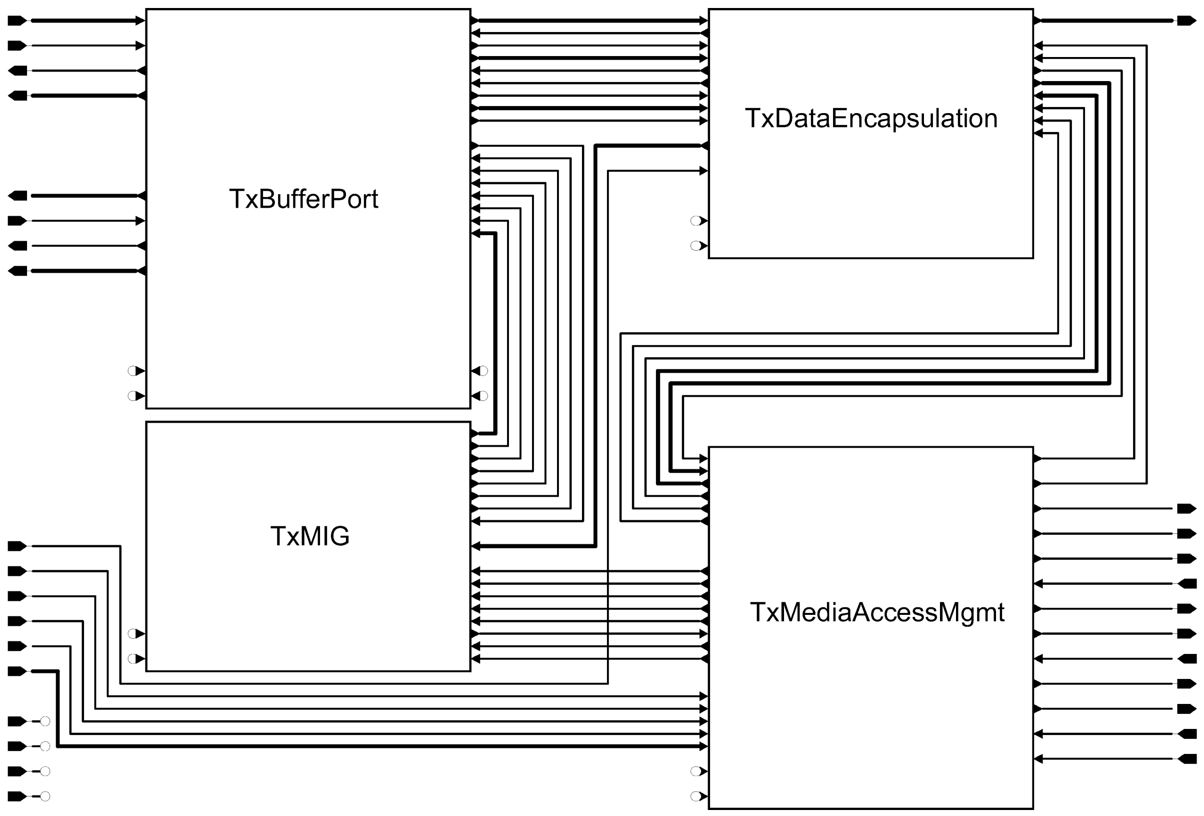


Figure 19: The structure of TxMAC.

The block *TxMAC* (fig. 19) implements the transmitter, which is defined by the service-primitive MA_DATA.request [IEEE 802.3, Clause 2.3.1] and by the precise specification [IEEE 802.3, Clause 4.2.8].

As can be seen in appendix C the transmit part is the most complex part of the whole Ethernet controller because of the great amount of shared signals in combination with a high clock-frequency.

TxMAC consists of four blocks:

- *TxDataEncapsulation* collates the framing functionality, i.e. the building of the frame. This block is described in sec. 4.1.2.1.
- *TxMediaAccessMgmt* collates the Medium Management functionality. This block is described in sec. 4.1.2.2.
- The buffer (*TxBufferPort*) is needed in order to be able to cross the clock domain since *TxMAC* belongs to the transmitter clock-domain and it interfaces towards the system clock-domain.
  This block is described in sec. 4.1.2.3.
- The block *TxMIG* generates data for the *MIB* and updates fields in the descriptor register *TxDescReg*. This block is described in sec. 4.1.2.4.

4.1.2.1 TxDataEncapsulation

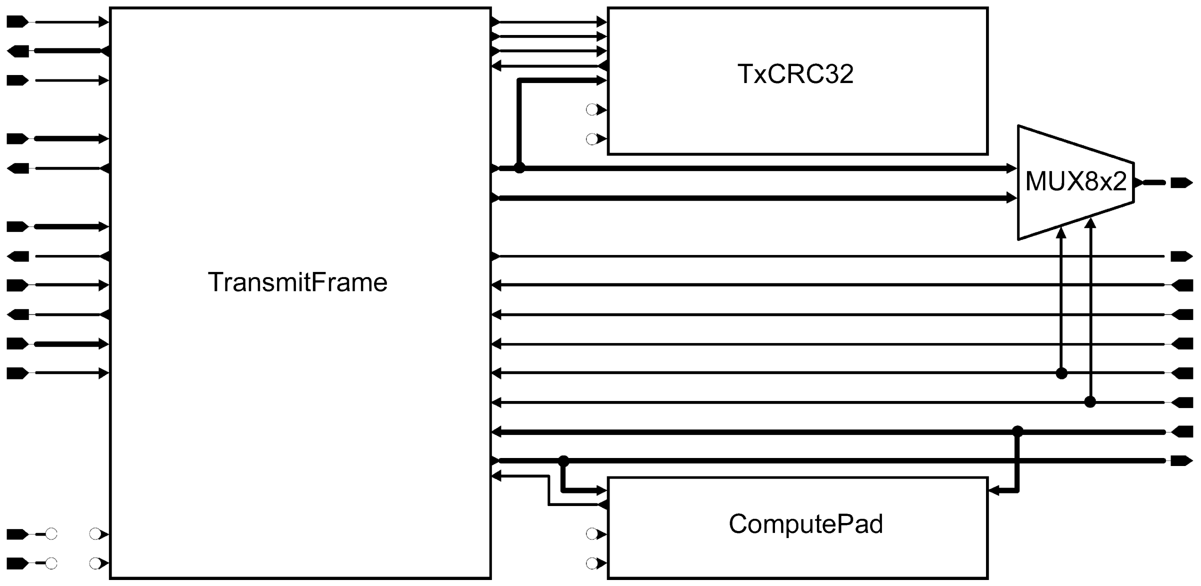


Figure 20: The structure of TxDataEncapsulation.

The functionality that shall be provided by the block *TxDataEncapsulation* (fig. 20) is described in IEEE 802.3, Clause 4.2.3.1. This block shall assemble the frame from the values provided by the MAC client, check if the frame has to be extended by insertion of extra data (i.e. padding) to fulfill the demands of minimum frame size and append a 32-bits CRC checksum.

4.1.2.1.1 TransmitFrame

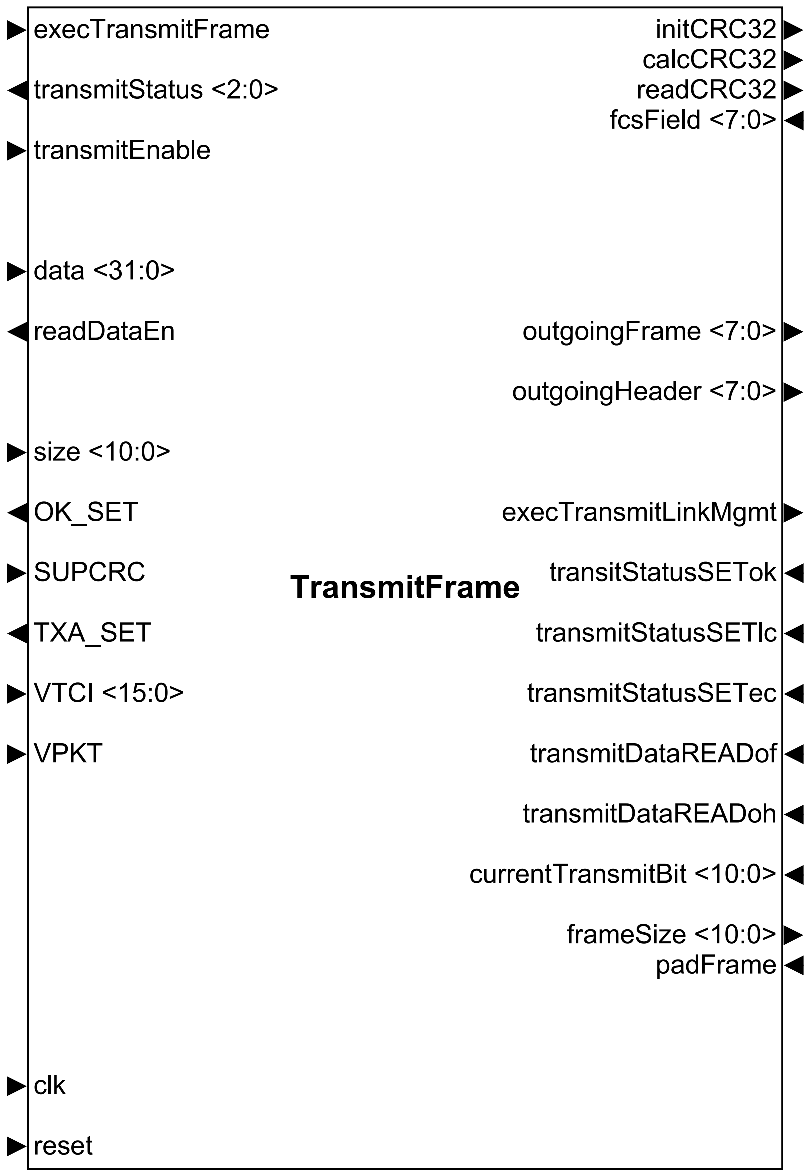


Figure 21: The symbol for TransmitFrame.

If transmission is enabled, *TransmitFrame* (fig. 21) use the incorporated procedure `TransmitDataEncap` to construct a frame by inserting values such as DA, SA, frame size etc. at appropriate places in the byte stream.

TransmitFrame uses *ComputePad* to check if it is necessary to pad the frame and it uses the process *TxCRC32* to compute the frame's checksum. In the contrary what the standard implies, the frame in this implementation is constructed "on-the-fly" in order to minimize the need of memory and to maximize the throughput in the system.

4.1.2.1.2 TxCRC32

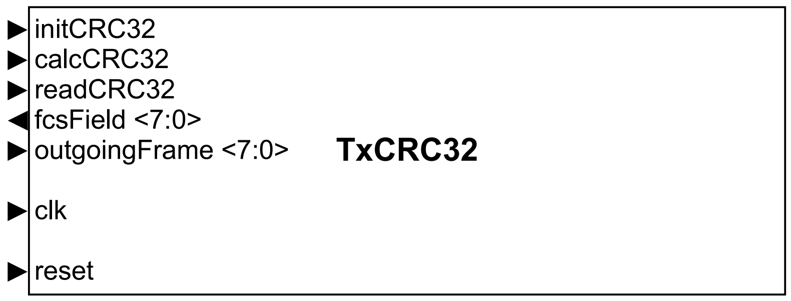


Figure 22: The symbol for TxCRC32.

The *TxCRC32* block (fig. 22) computes the 32-bits CRC checksum. The block is a modified version of the free IP block “IEEE 802.3 Cyclic Redundancy Check” provided by Xilinx [XAPP209] as a reference design.

The simplest way to compute the CRC value is by using a linear feedback shift-register (LFSR) as shown in figure 23.

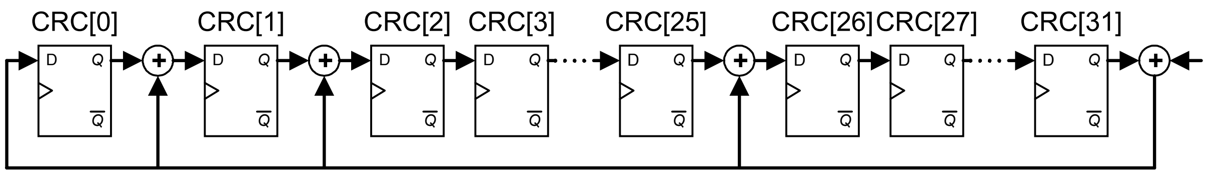


Figure 23: LFSR implementation of CRC-32.

Using an LFSR is, however, not feasible since the application yields that it would have to work in 1 GHz. Instead, the CRC value has to be computed in parallel. This can be done using a state machine, as shown in figure 50, where the part of the logical net that computes the CRC-value is obtained from the IP core.

Before the checksum generation begins, the register has to be loaded with ones, which is done by assertion of *initCRC32*. As soon as *calcCRC32* is asserted the generation of the checksum begins. When *calcCRC32* later is de-asserted, the value in the register is shifted one byte at a time. The inverted value of the lowest byte is at any time present at the output *outgoingFrame*.

The input *readCRC32* is not used and shall be ignored.

4.1.2.1.3 ComputePad

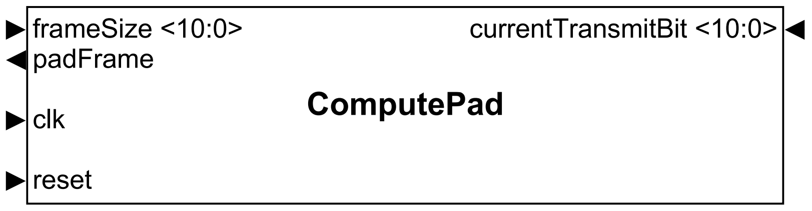


Figure 24: The symbol for ComputePad.

If the frame size is less then minimum allowed value, arbitrary data has to be appended to the frame. The number of bytes that have to be appended, if needed, is computed by the block *ComputePad* (fig. 24), which simply compares the frame size given by the MAC client with the allowed minimum value defined in the standard [IEEE 802.3, Clause 4.4.2.1, 3 & 4].

4.1.2.2 TxMediaAccessMgmt

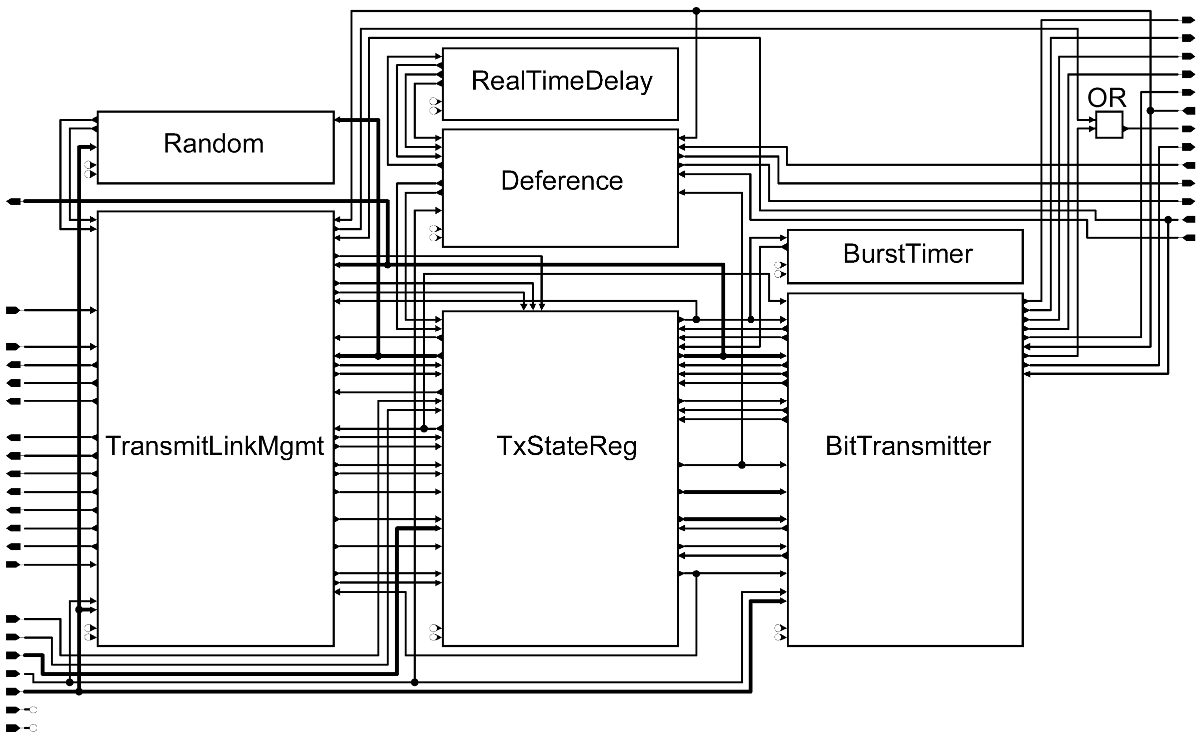


Figure 25: The structure of TxMediaAccessMgmt.

The function that shall be provided by the block *TxMediaAccessMgmt* (fig. 25) is described in IEEE 802.3, Clause 4.2.3.2.

This block shall be able to handle collision detection, collision enforcement (i.e. jam, back off and retransmission), carrier extension and frame bursting.

4.1.2.2.1 TransmitLinkMgmt

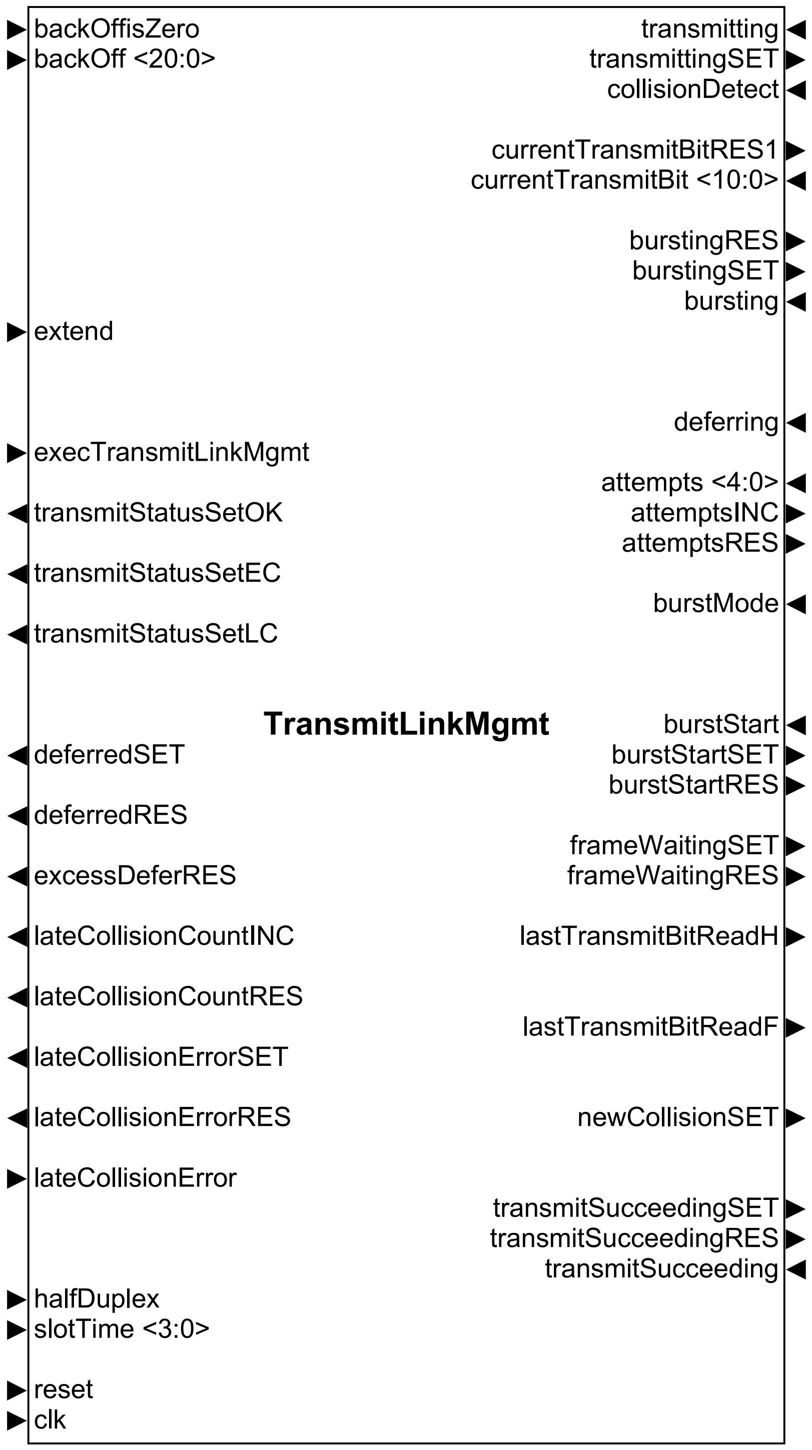


Figure 26: The symbol for TransmitLinkMgmt.

TransmitLinkMgmt (fig. 26) attempts to transmit the frame. In half duplex mode, it first defers to any passing traffic. When a frame transmission is initiated, the internal procedure *StartTransmit* alerts the process *BitTransmitter* that transmission is to begin. If a collision occurs, the transmission is aborted and retransmission is scheduled using a suitable back off interval, which is computed by the block *BackOff*. Collisions are detected by the in standard defined procedure `WatchForCollision`, which is incorporated into the block *TransmitLinkMgmt*.

4.1.2.2.2 Random

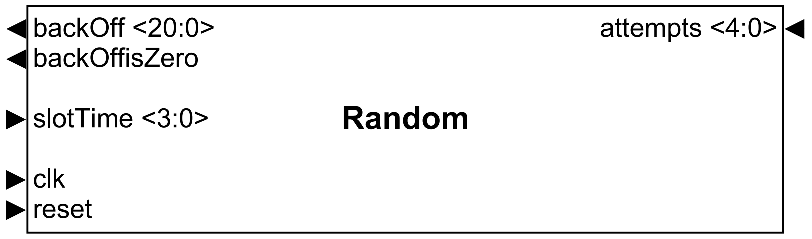


Figure 27: The symbol for Random.

The function `Random` [IEEE 802.3, Clause 4.2.3.2.5] is implemented as a process (fig. 27) with the same name. The standard implies that the function, on request, shall return a uniformly distributed random integer between (and including) zero and `maxBackOff` i.e. {0, 1, 2, …, `maxBackOff`-1}. The variable `maxBackOff` can take the values 2^1, 2^2, 2^3, …, 2^{10}.

In the implementation, this functionality is obtained by letting the random number generator produce a new number every clock cycle in the interval zero to 2^{10}-1. The bits are then masked with a vector that is `maxBackOff`-1 to obtain the correct interval. The masking is done in the calling block TransmitLinkMgmt.

The standard implies that a random number only should be computed when needed but this gives rise to timing problems since the number then has to be computed with zero latency. Instead, a number is computed every clock cycle and ignored if not needed. The drawback with this solution is slightly higher power consumption.

4.1.2.2.3 BurstTimer

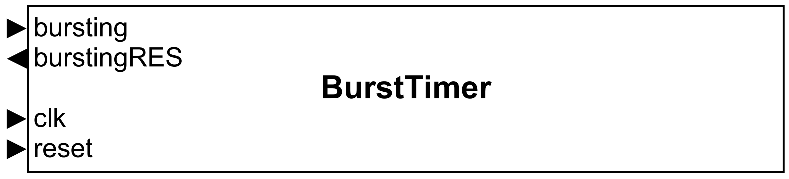


Figure 28: The symbol for BurstTimer.

In gigabit half duplex burst mode, the `BurstTimer` process [IEEE 802.3, Clause 4.2.3.2.7], see figure 28, clears the signal bursting when the `burstLimit` is reached. In any other modes, this block has no function. The constant `burstLimit` is the maximum size of a frame transmitted in this mode. This type of frame is sometimes referred to as “Jumbo frame”.

4.1.2.2.4 Deference

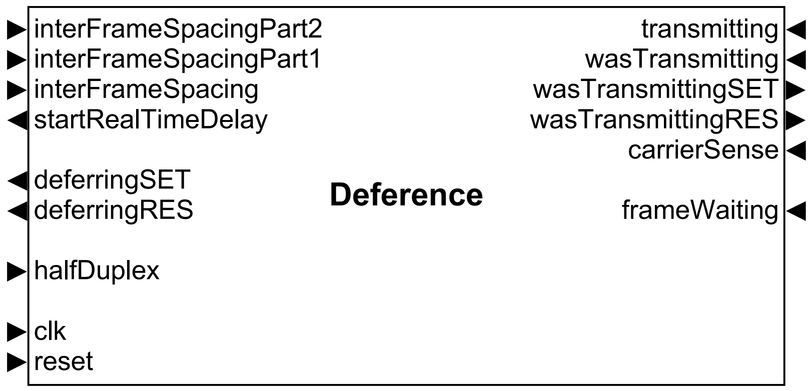


Figure 29: The symbol for Deference.

This block implements the `Deference` process [IEEE 802.3, Clause 4.2.3.2.1], see figure 29. It continuously computes the proper value of the signal `deferring`, which indicates that any pending transmission must wait for the medium to clear. It assures that the minimum inter frame gap is obtained. When transmitting in half duplex burst mode, the signal is true throughout the whole burst and ignored by other parts of the system.

4.1.2.2.5 RealTimeDelay

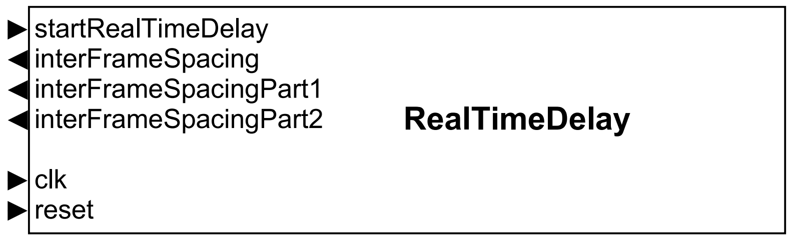


Figure 30: The symbol for RealTimeDelay.

The block *RealTimeDelay* (fig. 30) implements the function `RealTimeDelay` and the procedure `StartRealTimeDelay` as a process. The timer is reset by assertion of the signal *startRealTimeDelay*. The timer starts counting when *startRealTimeDelay* is de-asserted. The outputs are then asserted or not depending of how many microseconds that have elapsed since the recent invocation (i.e. assertion and de-assertion of *startRealTimeDelay*) of the timer, see figure 31.

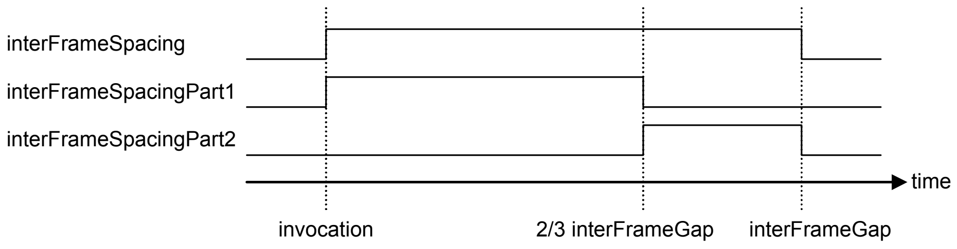


Figure 31: The signal pattern of the outputs of RealTimeDelay.

4.1.2.2.6 BitTransmitter

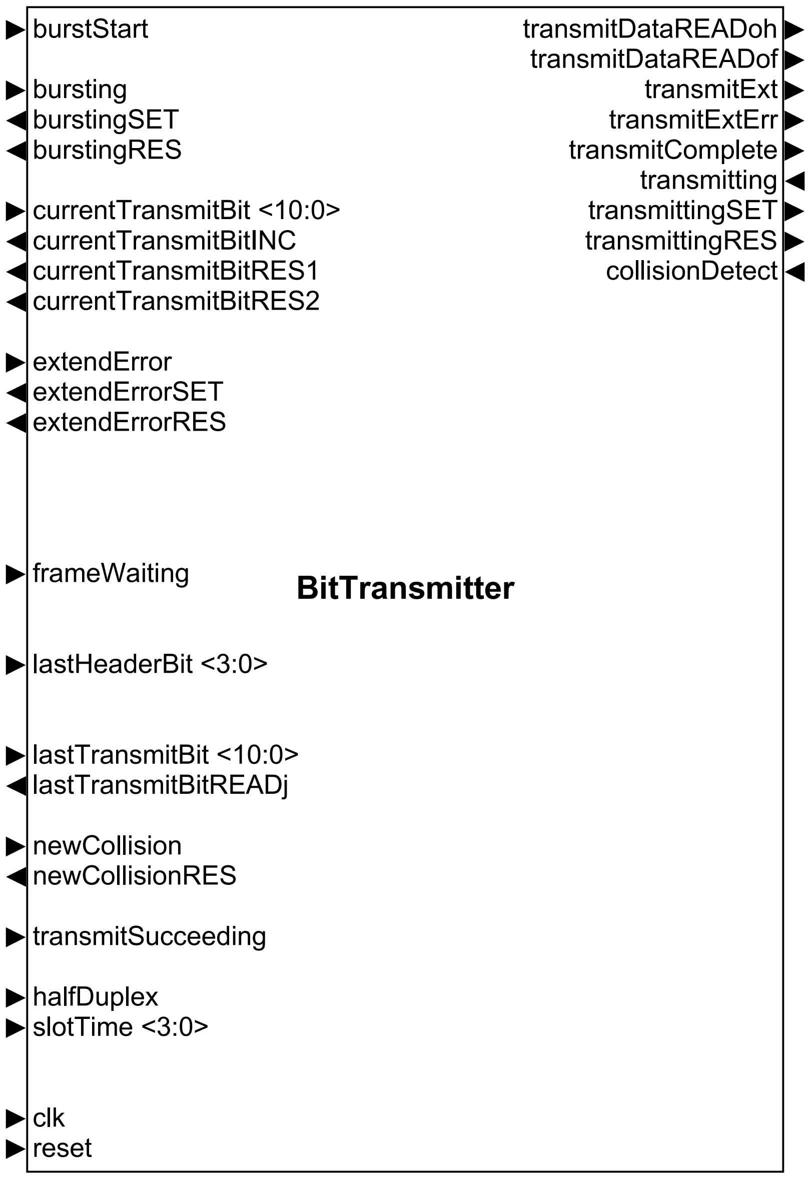


Figure 32: The symbol for BitTransmitter.

The *BitTransmitter* (fig. 32) process is enabled by the *TransmitLinkMgmt* process. The process incorporates four procedures defined in the standard, `PhysicalSignalEncap`, `NextBit`, `StartJam` and `InterFrameSignal`.

`PhysicalSignalEncap` transmits the header, i.e. Preamble (PA) and Start Frame Delimiter (SFD). When the header has been transmitted without collisions *BitTransmitter* transmits the data. Between each transmission of a byte, the counter *NextBit* is incremented.

If a collision is detected by the transmitting station during transmission, the procedure `StartJam` will be called that will cause the *BitTransmitter* to send arbitrary data for sufficient time so all stations on the network will detect the collision.

When sending in burst mode, `InterFrameSignal` fills the interval between two following frames with extension bits.

4.1.2.2.7 TxStateReg

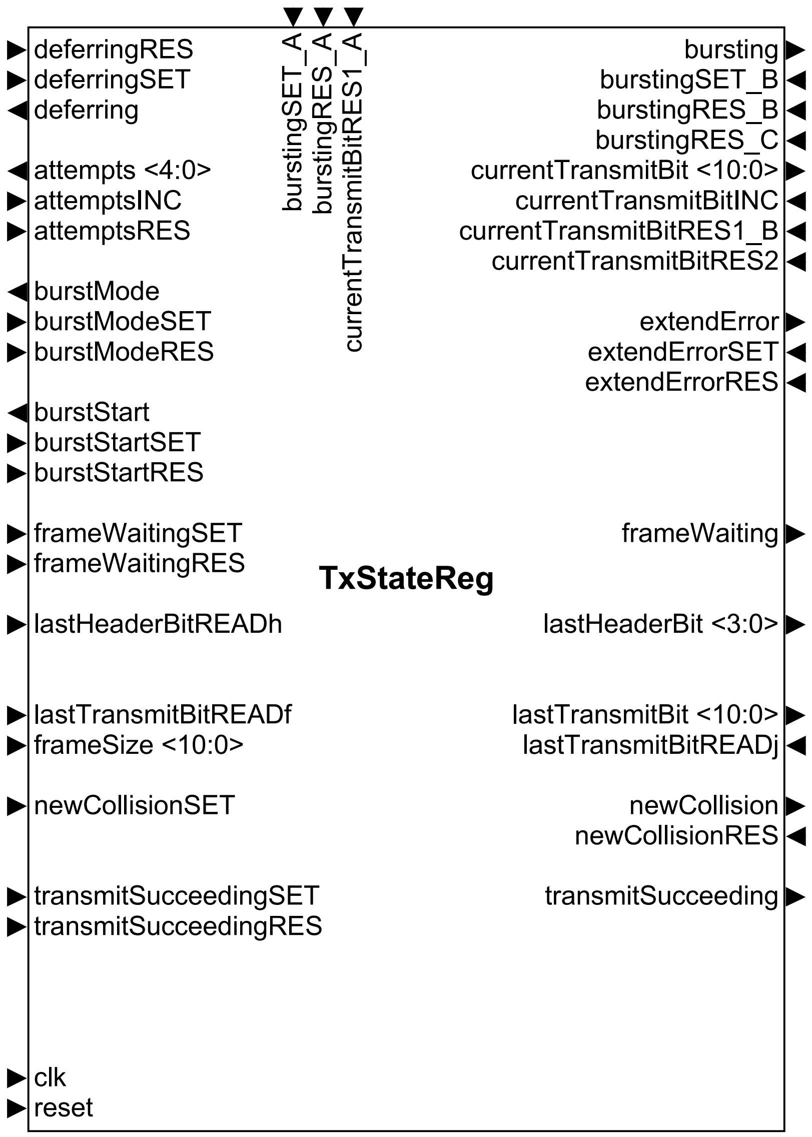


Figure 33: The symbol for TxStateReg.

The register *TxStateReg* (fig. 33) holds the signals that are shared within the *TxMAC*, see appendix C. The register consists of ordinary D-flip-flops, which typically have one (or more) set and clear input(s), and an output each.

4.1.2.3 TxBufferPort

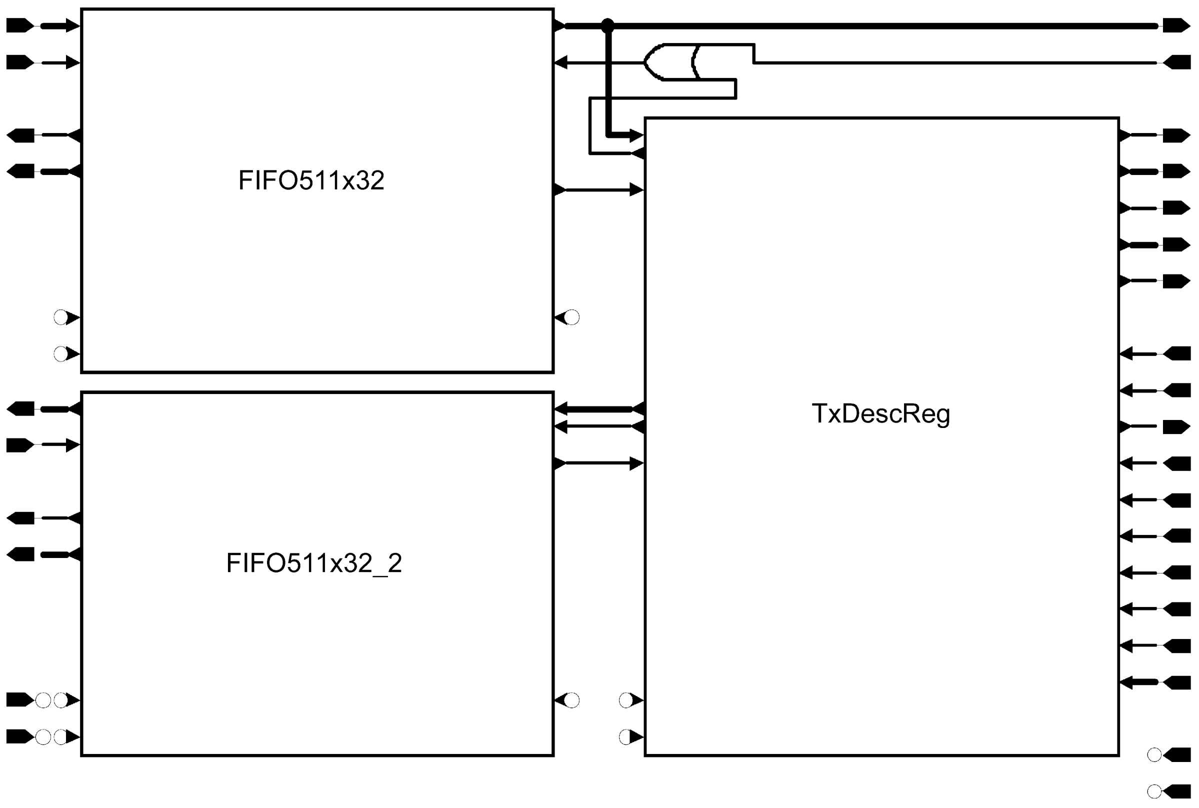


Figure 34: The structure of TxBufferPort.

When data is ready for transmission, *TxBufferPort* (fig. 34) receives a descriptor from *TxBufMgr* via *TxDataFIFO*. When the descriptor is received and stored into the *TxDescReg*, the *TxDataFIFO* is filled with the beginning of the packet that is to be sent. At this moment, *TxDescReg* asserts the signal *execTransmitFrame*. When a link has been established the transmitting of the packet begins and the *TxDataFIFO* transfers data from *TxBufMgr* further to *TxDataEncapsulation*. When the transmission is finished, or aborted, the ownership of the descriptor together with updated fields of it is returned to *TxBufMgr* via *TxDescFIFO*. As soon as *TxBufMgr* returns the ownership, a new descriptor can be transferred to *TxBufferPort* to start another transmission.

4.1.2.3.1 TxDataFIFO

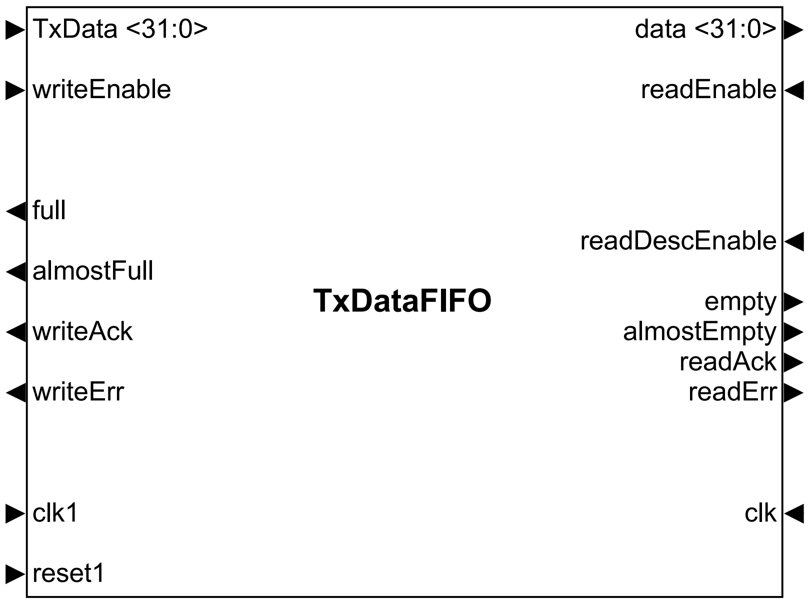


Figure 35: The symbol for TxDataFIFO.

The *TxDataFIFO* (fig. 35) is an IP core from Xilinx [XAPP258]. The FIFO uses two independent clocks, which makes it ideal to use when crossing clock domains.

The functionality of a FIFO is considered fundamental and not discussed further.

4.1.2.3.2 TxDescFIFO

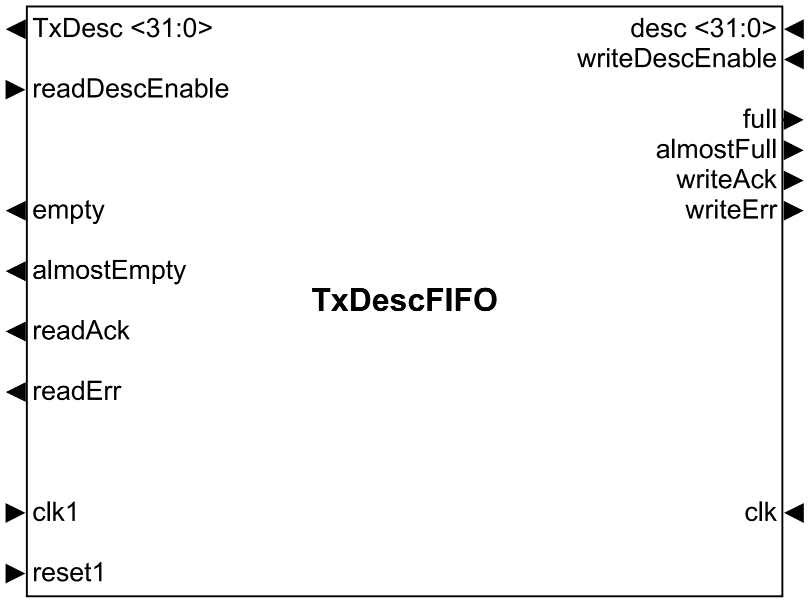


Figure 36: The symbol for TxDescFIFO.

The FIFO *TxDescFIFO* (fig. 36) transfers the content of the register *TxDescReg* to *TxBufMgr*. It is identical with *TxDataFIFO*, but mirrored.

4.1.2.3.3 TxDescReg

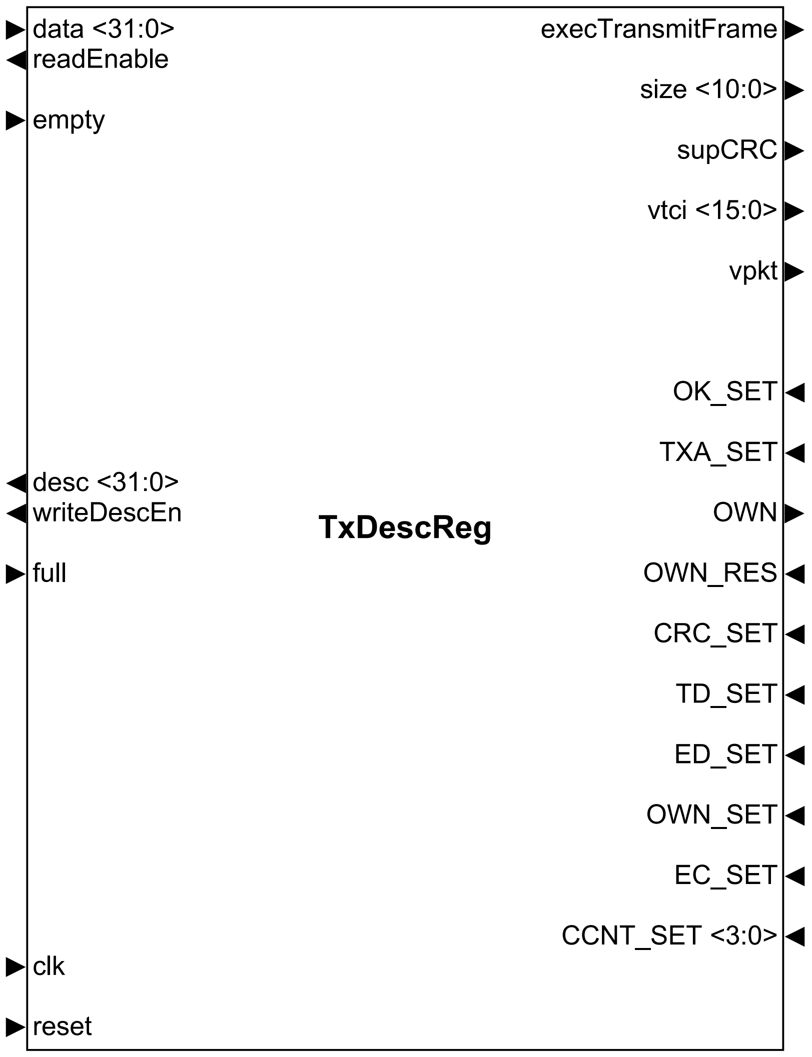


Figure 37: The symbol for TxDescReg.

The register *TxDescReg* (fig. 37) holds the transmit descriptor associated with the packet currently being transmitted.

Before a packet is being transmitted, the descriptor is transferred to the register via *TxDataFIFO* and the input data<31:0>. *TxDescReg* requests new data until signal empty is asserted.

When the descriptor is received, *execTransmitFrame* will be asserted which cause the underlying layers to attempt to transmit the packet.

After the transmission is finished, or aborted, and fields in the descriptor has been updated, e.g. via *ok_set*, the ownership is returned to *TxBufMgr*. By asserting *own_res*, not only the ownership is returned but also the transfer of the content of the register is initiated, i.e. assertion of *writeDescEn*. The data is transmitted via the bus *desc<31:0>*. The signal full reports if the FIFO is full and the transmission is paused until full is de-asserted.

4.1.2.4 TxMIG

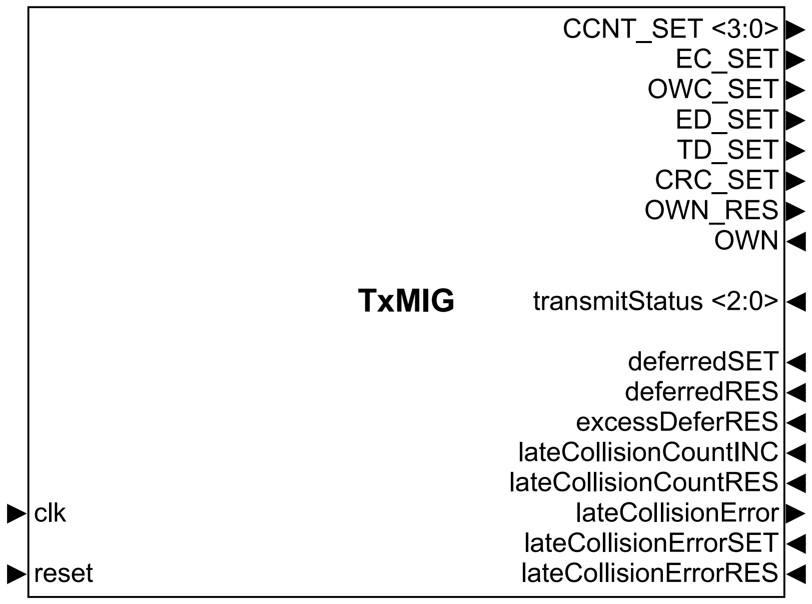


Figure 38: The symbol for TxMIG.

This implementation is prepared for providing DTE Layer Management [IEEE 802.3, Clause 5.2.4]. The Layer Management service consists of a set of counters and actions. The counters together form the Management Information Base (*MIB*), which is located in the block *MAC*. Some of the services provided by the Layer Management are implemented in the block Management Information Generator (*TxMIG*) in the transmitter and in the corresponding location in the receiver (*RxMIG*). Other services will be placed in the *MIB* itself.

TxMIG (fig. 38) monitors different status signals in the transmitter and updates appropriate counters, either automatically or by the blocks within the transmitter.

After each transmission, *TxMIG* updates the management information in the *MIB* and then resets all counters.

The design shown in figure 38 is not complete, since it lacks the interface towards the *MIB*.

4.1.3 Precise Design: Reconciliation Sublayer (RS)

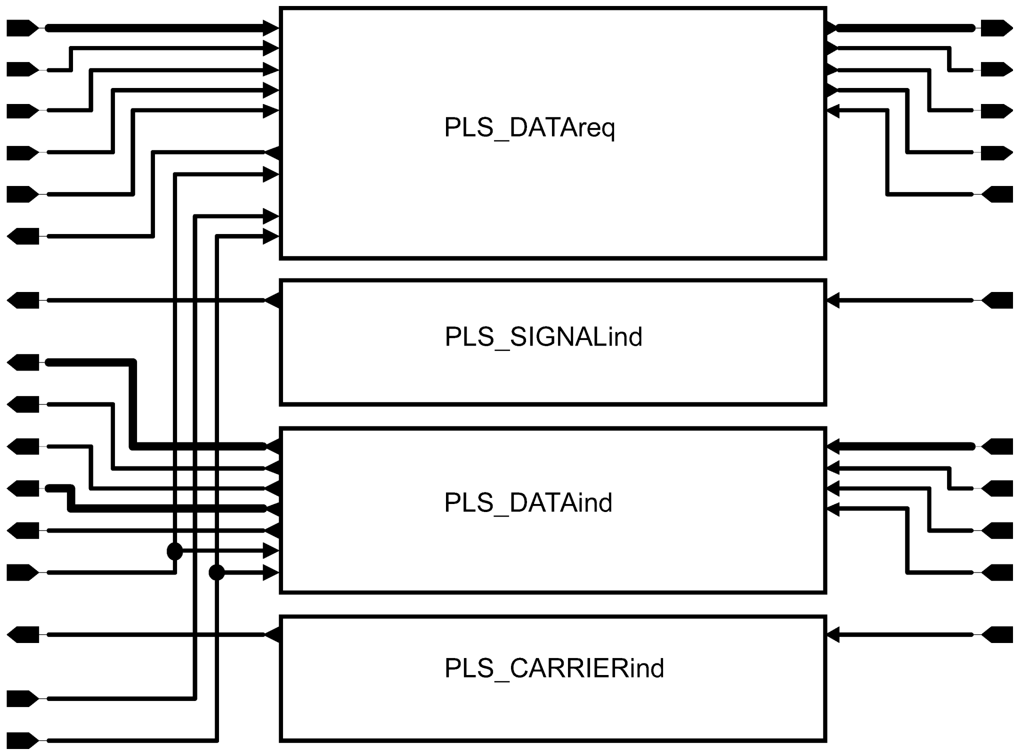


Figure 39: The structure of RS.

The MII and GMII consist of two parts, the RS interface and the station management interface (provided by STA).

RS (fig. 39) maps the signals provided by the MII and GMII to the PLS service primitives. Further, RS maps the variables, procedures and functions provided to the Mac sublayer by the Physical layer to the PLS service primitives. The mapping of the latter is shown in table 6.

Table 6: Mapping of PLS service primitives to physical layer signals as presented to MAC sublayer

Variables	
ReceiveDataValid	PLS_DATA_VALID.indicate
CarrierSense	PLS_CARRIER.indicate
Transmitting	PLS_DATA.request
WasTransmitting	PLS_DATA.request
CollisionDetect	PLS_SIGNAL.indicate
Procedures	
TransmitBit	PLS_DATA.request
Wait	N/A
Functions	
ReceiveBit	PLS_DATA.indicate

The variables *transmitting* and *wasTransmitting* are controlled by the MAC sublayer. The signals needed to set and clear these signals are not shown in figure 39.

The, by Physical layer provided, procedure `TransmitBit` is implemented by the signals *transmitData<7:0>*, *transmitExt*, *transmitExtErr* and *transmitDataValid* (where *transmitDataValid* is the signal that performs the execution of the procedure).

The, by physical layer provided, function `ReceiveBit` is implemented by the signals *receiveData<7:0>*, *receiveExt* and *receiveDataValid* (where *receiveDataValid* is the signal that performs the execution of the function).

The design of MII and GMII allows the two interfaces to share many of the signals and the behavior needs only to be slightly modified for a few of them depending of which interface to provide.

The signals *transmitComplete*, *receiveExtErr* are never used and shall be omitted.

4.1.3.1 PLS_DATAreq

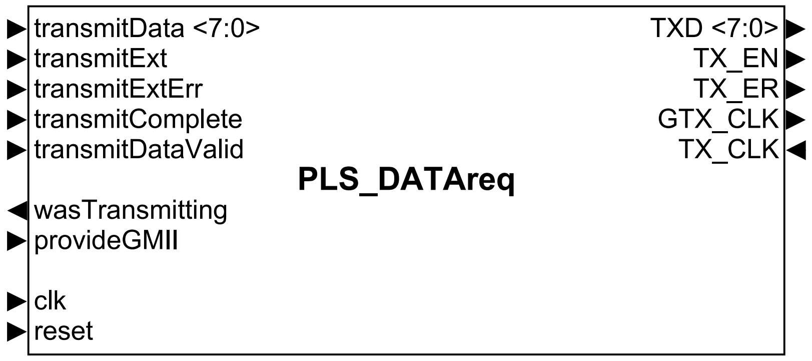


Figure 40: The symbol for PLS_DATAreq.

The block *PLS_DATAreq* (fig. 40) provides the Physical layer interface procedure `TransmitBit` [IEEE 802.3, Clause 4.3.3], which is the instantiation of the service primitive PLS_DATA.request [IEEE 802.3, Clause 22.2.1.1 (MII) and 35.2.1.1 (GMII)].

The mapping of PLS_DATA.request to MII and GMII is not identical. In order to obtain the different behavior depending of which type of interface to present, i.e. MII or GMII, the control signal provideGMII is used.

In MII-mode, the signals *transmitExt*, *transmitExtErr*, *TX_ER* and *GTX_CLK* are not used. While *transmitDataValid* is asserted, *transmitData* will be stored in a double-clocked FIFO synchronously to *clk*. As long as the FIFO is not empty, *TX_EN* will be asserted and data read out, four bits at a time, to *TXD<3:0>* synchronous to *TX_CLK*. *TX_CLK* is generated by the PHY.

In GMII-mode, the signal *TX_CLK* is not used. While *transmitDataValid* is asserted, *transmitExtErr*, *transmitExt* and *transmitData* are read and the values of the output signals will be as shown in table 7.

Table 7: Encoding of the GMII signals TXD, TX_EN and TX_ER

transmitExtErr	transmitExt	transmitData<7:0>	TXD<7:0>	TX_EN	TX_ER
0	0	00 – FF	transmitData	1	0
0	1	00 – FF	0F	0	1
1	0	00 – FF	1F	0	1
1	1	00 – FF	1F	0	1

The signals *TX_EN* and *TX_ER* are not supposed to be asserted at the same time. When *transmitDataValid* is not asserted, all outputs will be zero.

The signal *transmitComplete* is never used and shall be omitted.

4.1.3.2 PLS_SIGNALind

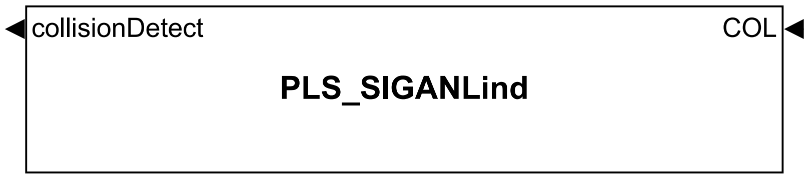


Figure 41: The symbol for PLS_SIGNALind.

The block PLS_SIGNALind (fig. 41) provides the Physical layer interface variable `collisionDetect` [IEEE 802.3, Clause 4.3.3], which is the instantiation of the service primitive PLS_SIGNAL.indicate [IEEE 802.3, Clause 22.2.1.4 (MII) and 35.2.1.4 (GMII)]

The behavior of the signal *COL* is identical for both MII and GMII and is specified in [IEEE 802.3, Clause 22.2.1.4 (MII) and 35.2.1.4 (GMII)].

In the standard, *COL* is specified to be asynchronous. By reasons explained in section 4.3.1, it is not wise to allow asynchronous signals. Instead, *COL* is sampled with respect to transmit clock domain and renamed to *collisionDetect*. This can be done without violating the standard since it does not place any emphasis on suitability to a particular implementation technology [IEEE 802.3, Clause 4.2.2].

4.1.3.3 PLS_DATAind

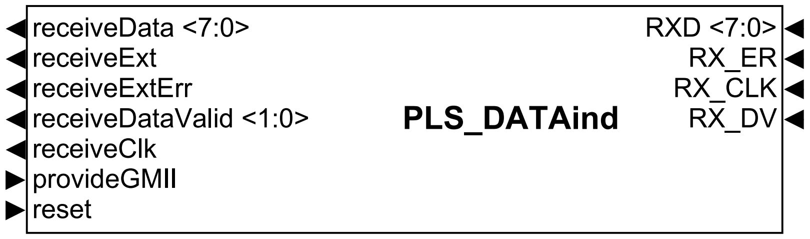


Figure 42: The symbol for PLS_DATAind.

The block PLS_DATAind (fig. 42) provides the Physical layer interface function `ReceiveBit` [IEEE 802.3, Clause 4.3.3], which is an instantiation of the two service primitives PLS_DATA.indicate [IEEE 802.3, Clause 22.2.1.2 (MII) and 35.2.1.2 (GMII)] and PLS_DATA_VALID.indicate [IEEE 802.3, Clause 22.2.1.7 (MII) and 35.2.1.7 (GMII)].

Neither the mapping of PLS_DATA.indicate nor PLS_DATA_VALID.indicate to MII and GMII are identical. In order to obtain the different behavior depending of which type of interface to present, i.e. MII or GMII, the control signal *provideGMII* is used. All MII/GMII signals in this block are synchronous to *RX_CLK*.

In MII-mode, the signals *receiveExt* and *receiveExtErr* are not used. The decoding of the input signals is presented in table 8.

Table 8: Decoding of the MII signals RX_DV, RX_ER and RXD

RX_DV	RX_ER	RXD<3:0>	receiveData<3:0 / 7:4>	receiveDataValid<0 / 1>
0	0	0 – F	0	0
0	1	0 – F	0	0
1	0	0 – F	RXD<3:0>	1
1	1	0 – F	INV (RXD<3:0>)	1

When both *RX_DV* and *RX_ER* are asserted, the RS must ensure that the MAC sublayer will detect a `FrameCheckError`, i.e. wrong checksum. This is obtained by inverting the data as long as the condition persists.

The direct mapping between *RX_DV* and *receiveDataValid* in MII-mode is allowed only if the process *BitReceiver* is implemented to receive a nibble of data on each cycle. This is fulfilled by using two signals that corresponds to first and second nibble in *receiveData<7:0>*.

In GMII-mode, all signals are used. The decoding of the input signals is presented in table 9.

Table 9: Decoding of the GMII signals RX_DV, RX_ER and RXD

RX_DV	RX_ER	RXD<7:0>	receiveData<7:0>	receiveDataValid	receiveExt
0	0	00 – FF	00	0	0
0	1	00 – 0E	00	0	0
0	1	0F	0F	1	1
0	1	10 – 1E	00	0	0
0	1	1F	INV (1F)	1	0
0	1	20 – FF	00	0	0
1	0	00 – FF	RXD<7:0>	1	0
1	1	00 – FF	INV (RXD<7:0>)	1	0

When a “Carrier Extend” is received, i.e. *RX_DV* is de-asserted, *RX_ER* is asserted and *RXD* is “`0F`”, the RS shall notify the MAC sublayer that extension bits have been received.

When a “Carrier Extend Error” is received, i.e. *RX_DV* is de-asserted, *RX_ER* is asserted and *RXD* is “`1F`”, the RS shall ensure that the MAC sublayer will detect a `FrameCheckError` [IEEE 802.3, Clause 35.2.1.5].

When both *RX_DV* and *RX_ER* are asserted RS shall ensure that the MAC sublayer will detect a `FrameCheckError` [IEEE 802.3, Clause 35.2.1.5].

The signal *receiveExtErr* is never used and shall be omitted.

4.1.3.4 PLS_CARRIERind

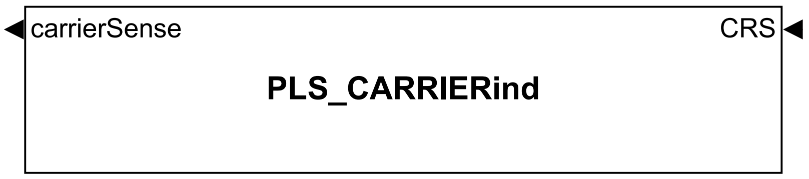


Figure 43: The symbol for PLS_CARRIERind.

The block PLS_CARRIERind (fig. 43) provides the Physical layer interface variable `carrierSense` [IEEE 802.3, Clause 4.3.3], which is an instantiation of the service primitive PLS_CARRIER.indicate [IEEE 802.3, Clause 22.2.1.3 (MII) and 35.2.1.3 (GMII)].

The mapping of PLS_CARRIER.indicate to MII and GMII is not identical. However, which is also is stated in [IEEE 802.3, Clause 22.2.1.3], the mapping is carried out in the same way in practice.

The signal *CRS* is asynchronous and therefore sampled into the transmit clock domain to avoid timing problems. The actual sampling occurs within the process *Deference*, see section 4.1.2.2.4.

4.1.4 Precise Design: Station Management (STA)

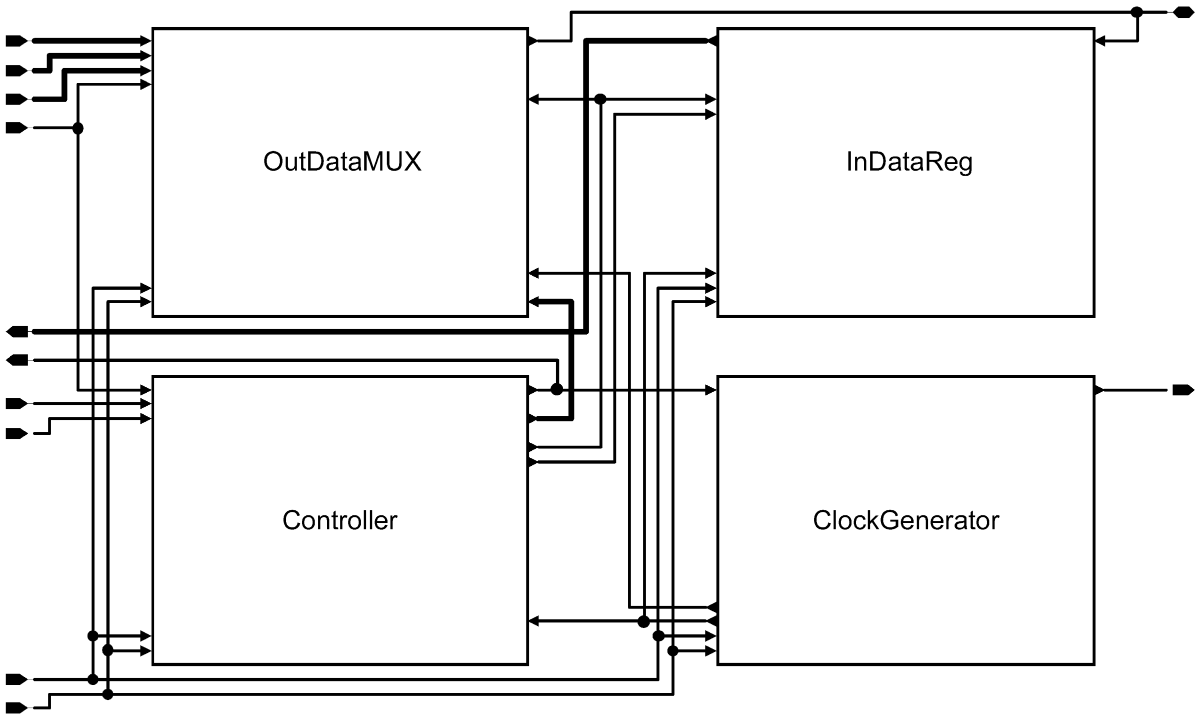


Figure 44: The structure of STA.

The Station Management entity (STA) (fig. 44) provides the MII/GMII Management Interface, which is a part of the MII/GMII. The management interface is the same regardless if it is a part of the MII or the GMII.

The management interface [IEEE 802.3, Clause 22.2.4] is a simple, two-wire, serial interface that connects a management entity and a managed PHY for the purposes of controlling the PHY and gathering status from the PHY. In addition to the signals, the interface also defines a frame format and protocol [IEEE 802.3, Clause 22.2.4.5] and a register set in the PHY [IEEE 802.3, Clause 22.2.4, Table 22-6].

The host interface of the STA is not covered in the standard, whether the implemented interface is suitable or not has not been confirmed.

4.1.4.1 InDataReg

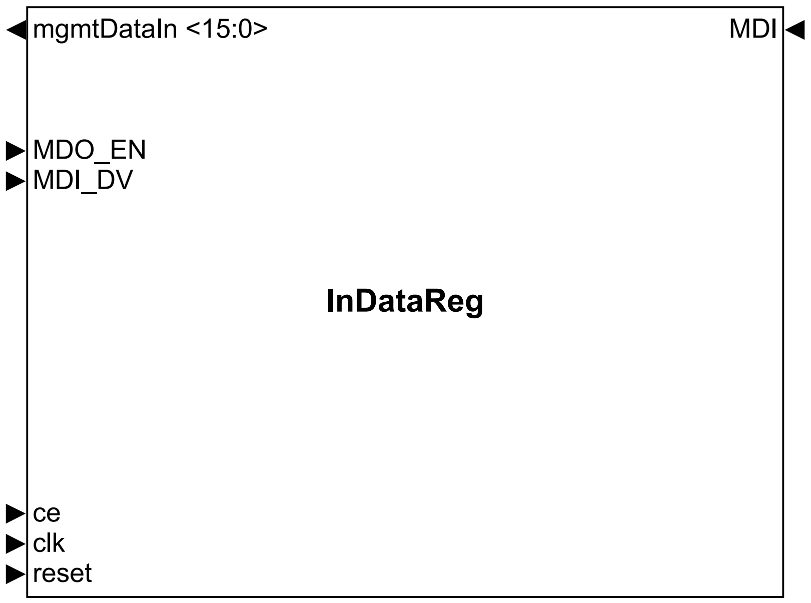


Figure 45: The symbol for InDataReg.

The register *InDataReg* (fig. 45) is managed by the *Controller*. When *MDI_DV* is true and *MDO_EN* is false, the bit present at *MDI* is shifted into the register. When the shift-register is full, the data is present in parallel form at the output *mgmtDataIn*.

MDI is one of the two channels that form the signal *MDIO*, which is explained in section 4.1.5.4.

4.1.4.2 ClockGenerator

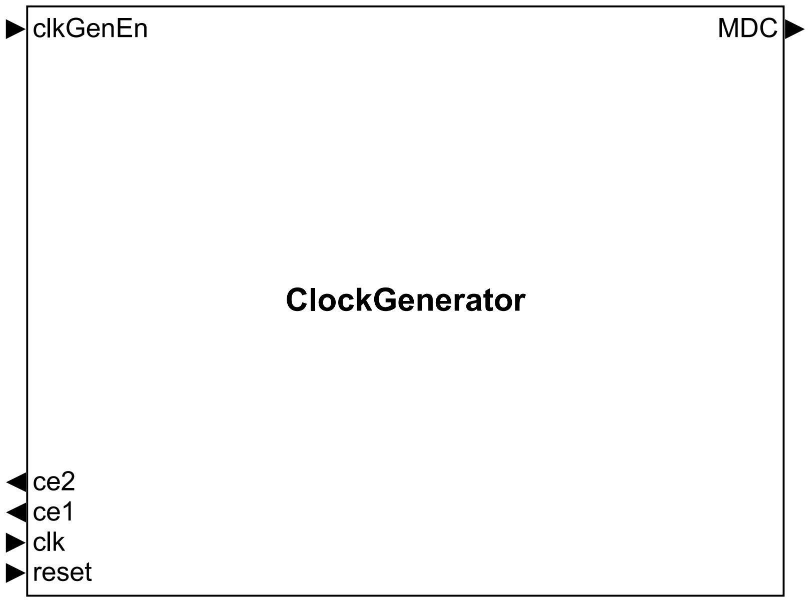


Figure 46: The symbol for ClockGenerator.

The *ClockGenerator* (fig. 46) generates the clock signal *MDC*. The block also generates two clock enable signals, *ce1* and *ce2*. These signals are used by the other blocks to determine when *MDC* makes a transition. When the *ClockGenerator* is disabled, i.e. *clkGenEn* is de-asserted, all outputs of the block will be de-asserted.

The MII/GMII Management Interface signal `MDC` [IEEE 802.3, Clause 22.2.2.11] is sourced by the Station Management entity to the PHY as the timing reference for transfer of information on the `MDIO` signal. `MDC` is a non-periodic signal that has no maximum high or low times. The minimum high and low times for `MDC` shall be 160 ns each, and the minimum period for `MDC` shall be 400 ns (2.5 MHz), regardless of the nominal period of `TX_CLK` and `RX_CLK`.

4.1.4.3 OutDataMUX

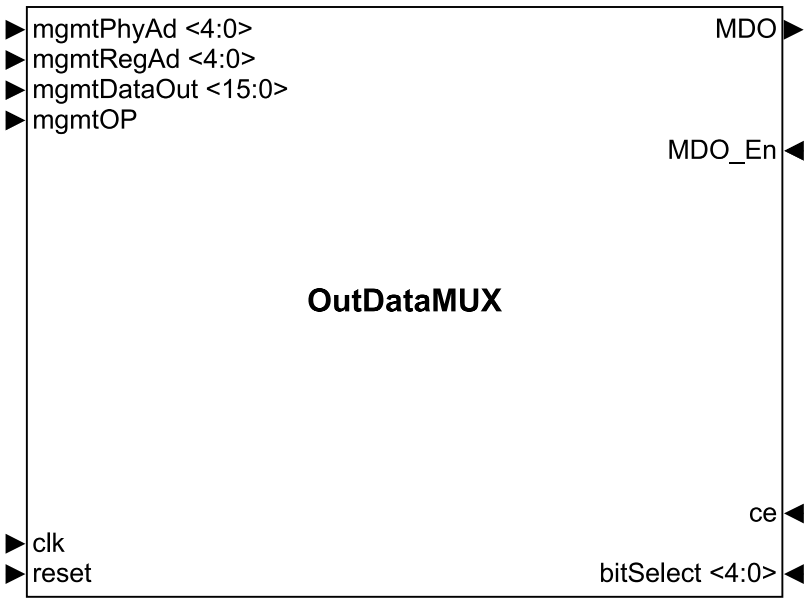


Figure 47: The symbol for OutDataMUX.

The *OutDataMUX* (fig. 47) builds the management frame and sends the frame in serialized form via *MDO* when *MDO_EN* is asserted. The signals *mgmtPhyAd*, *mgmtRegAd*, *mgmtDataOut* and *mgmtOP* represents the field values of the frame.

MDO is one of two channels that form the signal *MDIO*, which is explained in section 4.1.5.4.

4.1.4.4 Controller

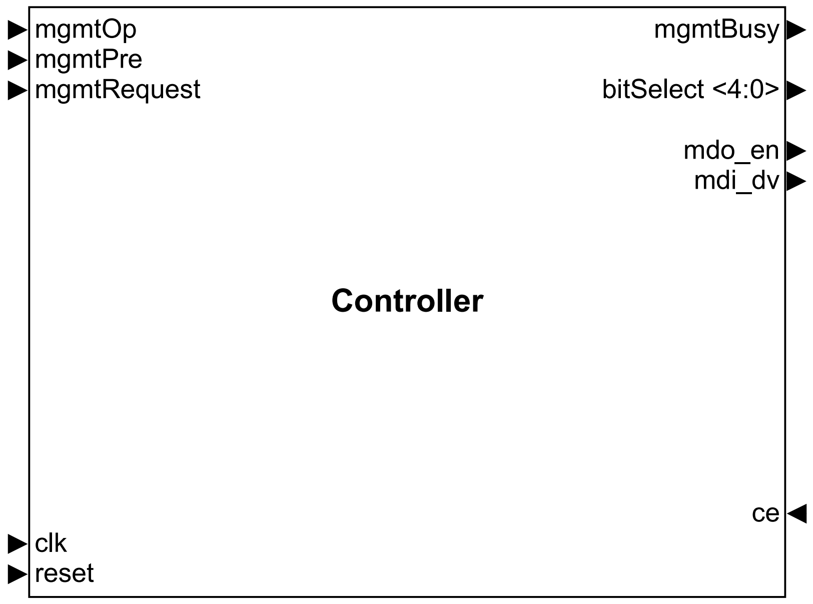


Figure 48: The symbol for Controller.

The *Controller* (fig. 48) manages the communication between the STA and one or more connected PHYs. The STA is enabled by assertion of *mgmtRequest*. If the controller can respond to the request, *mgmtBusy* is asserted. Assertion of *mgmtRequest* when *mgmtBusy* is asserted is ignored.

The signal *mgmtOP* determines if the STA shall perform a read or write transaction. Each frame begins with a preamble. This preamble is not always needed and can be omitted by de-assertion of *mgmtPre*. The signal *bitSelect* is used by the *OutDataMUX* to select which bit from which filed to transmit.

`MDIO` [IEEE 802.3, Clause 22.2.2.12] is a bidirectional signal between the PHY and the STA. It is used to transfer control information and status between the PHY and the STA. When `MDO_EN` is asserted, control information is driven by the STA synchronously with respect to `MDC` and is sampled synchronously by the PHY. When `MDI_DV` is true, status information is driven by the PHY synchronously with respect to `MDC` and is sampled synchronously by the STA. `MDO_EN` and `MDI_DV` cannot be true at the same time.

4.2 Analysis of Possible Critical Blocks

Reasons that could vindicate an implementation of a specific block can be:

- Speed:
  There are reasons to believe that the block should fail to pass the delay and timing constraints. Such reason can be a large counter that will be updated each clock-cycle, a large multiplier, signals used by many blocks (propagation delay and skew) etc.
- Complexity:
  If the block is very complex it can be hard to predict its performance and behavior.
- Simplify verification:
  If the block is simple, meaning little room for mistakes, it can be a reason to rather implement it to simplify the verification process of other blocks instead of constructing test vectors that simulates the block.
- Unspecified interface:
  Since the standard do not specify electrical interfaces between each block, there can be a reason for implementation to achieve a fully specified electrical interface.

With the reasons stated above and after studies of the Pascal-code and the interconnection diagrams in appendix C, the following blocks have been selected for implementation:

- `TransmitLinkMgmt`
- `CRC32`
- `BitTransmitter`

The motivation for each block selected can be found in section 4.2.1-3.

The block `CRC32` will not be shared between the transmitter and receiver since, which can be read in section 4.2.2, the block will process the data in real-time. In that case, the transmitter and receiver have to have their own checksum generator in order to be able to support full duplex communication.

As can be seen in above listing all blocks can be found in the transmitter. This is not surprising since the transmitter consists of both more blocks and more signals than the receiver does.

4.2.1 Critical Block: TransmitLinkMgmt

The implementation of this block will include the, in standard defined, blocks `TransmitLinkMgmt`, `WatchForCollision`, `StartTransmit` and `BackOff`.

As can be seen in appendix C this block involves the counter `currentTransmitBit`, which can cause delay problems since it is 11 bits wide, it may be updated each clock cycle and it is distributed to several other blocks. The block is also complex since it incorporates much functionality, which performance and behavior is hard to predict. Further, the interface is not specified in the standard.

4.2.2 Critical Block: TxCRC32

There are two ways that the checksum can be generated, either it is done before the transmission begins, either do it "on-the-fly". The problem with the first solution is that a buffer is needed to store the frame before the transmission begins and the delay from the transmission is initiated at higher layers until the actual transmission begins. The other way to generate the checksum is better but two factors have to be verified, the throughput and the latency. The critical of the two is the throughput, but when that is solved, an eventual latency problem is easily solved by insertion of delay elements in the data path.

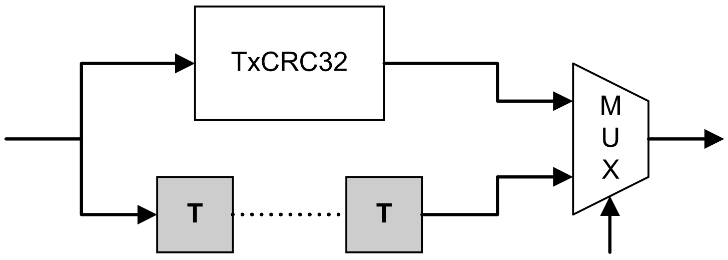


Figure 49: Insertion of delay elements to overcome latency problems.

The interface of the block is simple so there is no need to implement any adjacent block for simplifying the verification.

4.2.3 Critical Block: BitTransmitter

The implementation of this block will include the, in standard defined, blocks `BitTransmitter`, `InterFrameSignal`, `PhysicalSignalEncap`, `StartJam` and `NextBit`.

As can be seen in appendix C this block involves the counter `currentTransmitBit`, which can cause delay problems since it is 11 bits wide, it may be updated each clock cycle and it is distributed to several other blocks. The block is also complex since it incorporates much functionality, which performance and behavior is hard to predict. Further, the interface is not specified in the standard.

4.3 Design Entry

The development environment (i.e. HDL Designer) allows several ways to enter the design as mentioned in section 3.4. In this project, the block diagram entry method has been used. There was no crucial factor in the choice of method, but rather by force of habit of the author.

4.3.1 Different Styles

The implementation is done with synchronous logic. The problems associated with asynchronous logic (e.g. temperature and voltage dependency, speed variations through different paths, difficult to verify) outnumber the benefits (e.g. higher speed, smaller implementations).

One very common construction is the state machine. In this implementation, a synchronous type of the commonly known Mealy machine has been used consistently (fig. 50**)**. In the synchronous type, the output signals are stored in a separate register, which are reloaded at each clock edge. Consequently, if an output signal is used for feedback it will be duplicated and stored in both registers.

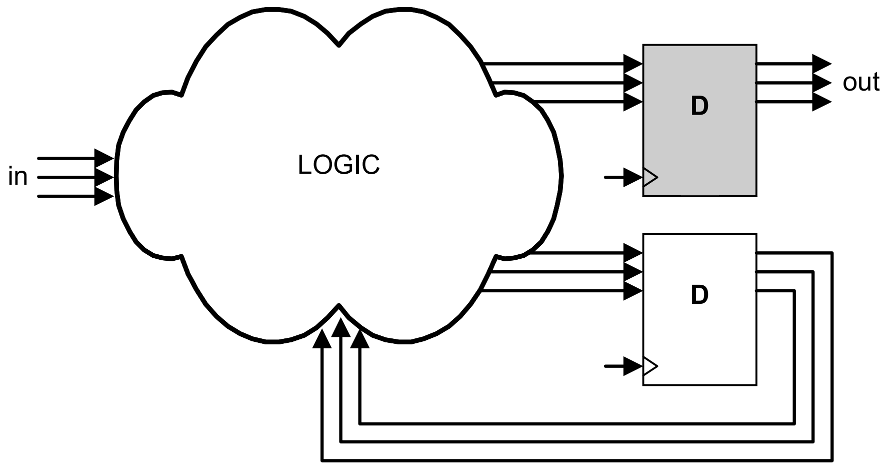


Figure 50: Mealy state machine. Grayed-out register makes it synchronous.

The difference between the original and the synchronous Mealy machine is that the outputs in the latter one do not suffer from the glitches that arise in the logical net. The trade-off is that the synchronous Mealy has a latency of one clock cycle while the original Mealy machine has no latency in terms of clock cycles (there will always be some delay).

4.3.2 Implementation Example: BitTransmitter

In order to illustrate how the Pascal-like code in the standard IEEE 802.3 was converted to VHDL the following example is used. The program is not a part of the standard.

The program consists of a process (*Counting*) and a procedure (*Sum*). Further, there are two input variables (*number* and *countingEnable*) and two output variables (*counter* and *status*). All these variables are global and used by other blocks that are not defined within this example. The variables *number* and *counter* are integers with the range 0 to 255.

```
type type_status = (overflow,idle,ok);
var  countingEnable : boolean;     {in}
var  number         : integer;     {in}
var  counter        : integer;     {out}
var  status         : type_status; {out}
```

Declaration of global variables.

```
process Counting;
  var break : boolean;
begin
  cycle {outer loop}
    counter := 0;
    status  := idle;
    break   := false;
    while countingEnable and not break do
    begin {inner loop}
      Sum(number);
      if counter < 256 then
        status := ok
      else
      begin
        status := overflow;
        break  := true
      end
    end {inner loop}
  end {outer loop}
end; {Counting}
```

Listing of process Counting.

```
procedure Sum(var offset: integer);
begin
  counter := counter + offset
end; {Sum}
```

Listing of procedure Sum.

While *countingEnable* is de-asserted, *counter* will be zero. Then, while *countingEnable* is asserted and *counter* is less than 256, *counter* is increased with the value *number* until it is 256 or greater. When *counter* becomes 256 ore greater, it will be reset but first it will contain a wrong value for one period. This is indicated by the status variable, which takes the value “overflow”.

At every moment, the status of the process will be reported. The status variable (*status*) can take the following three values:

- “idle”, i.e. *countingEnable* is de-asserted
- “ok”, i.e. *countingEnable* is asserted and *counter* $\in$ [0, 255]
- “overflow”, i.e. *countingEnable* is asserted and *counter* is greater than 255

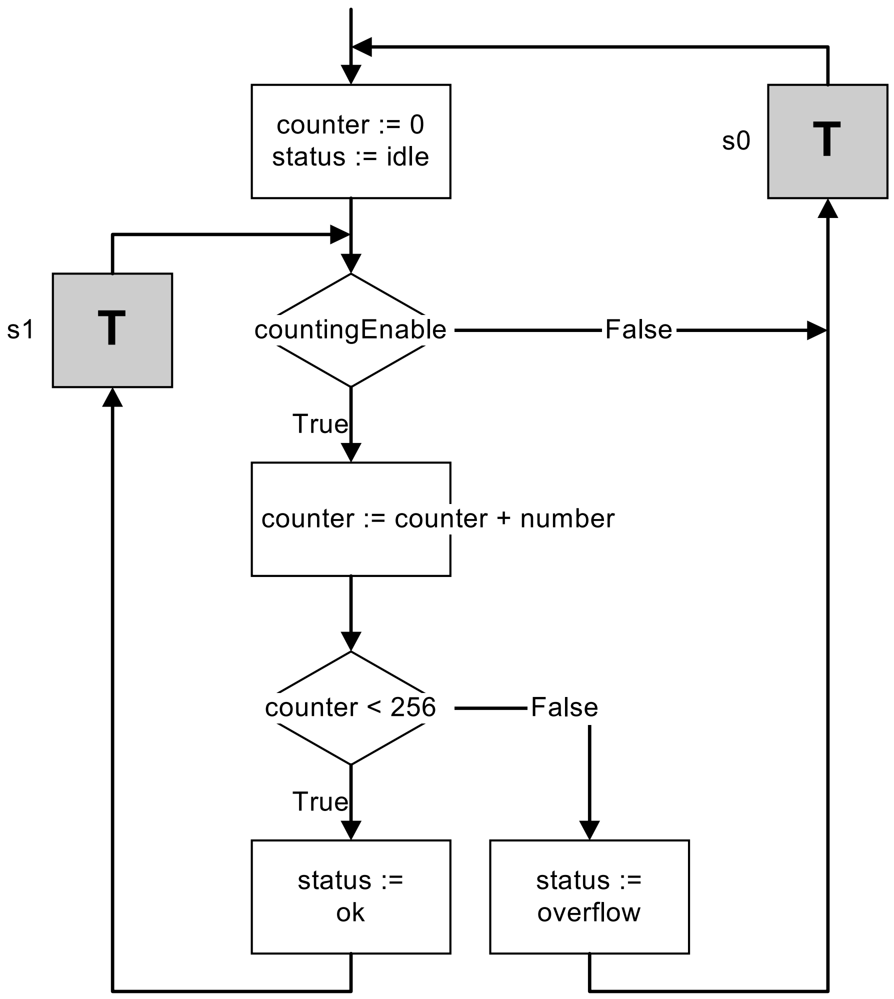


Figure 51: The program flow with grayed out delay elements.

First, which is not illustrated, the procedure(s) is written in line in the code of the process. Then a flowchart (fig. 51) is created, not including the grayed out boxes.

The flowchart now consists of three loops. In each of these loops a delay element must be inserted in order to achieve the correct behavior, otherwise for example the value at the input *number* might be added to *counter* several times when it only was intended to be added once. Each delay element, denoted T in figure 51, is then assigned a state number s0, s1, …

Then, the flowchart is divided into several flows. Each of theses flows begins at a delay element and ends at one or more. When the branches in each of these flows have been studied, some of these might be removed. Such a branch is when starting in state s0. As can be seen in figure 51, the counter is first reset and then if *countingEnable* is asserted it is set to *number*. Since *number* always is less than 256, it is unnecessary to check if *counter* is less than 256 so the false branch can be removed. The resulting flows are shown in figure 52.

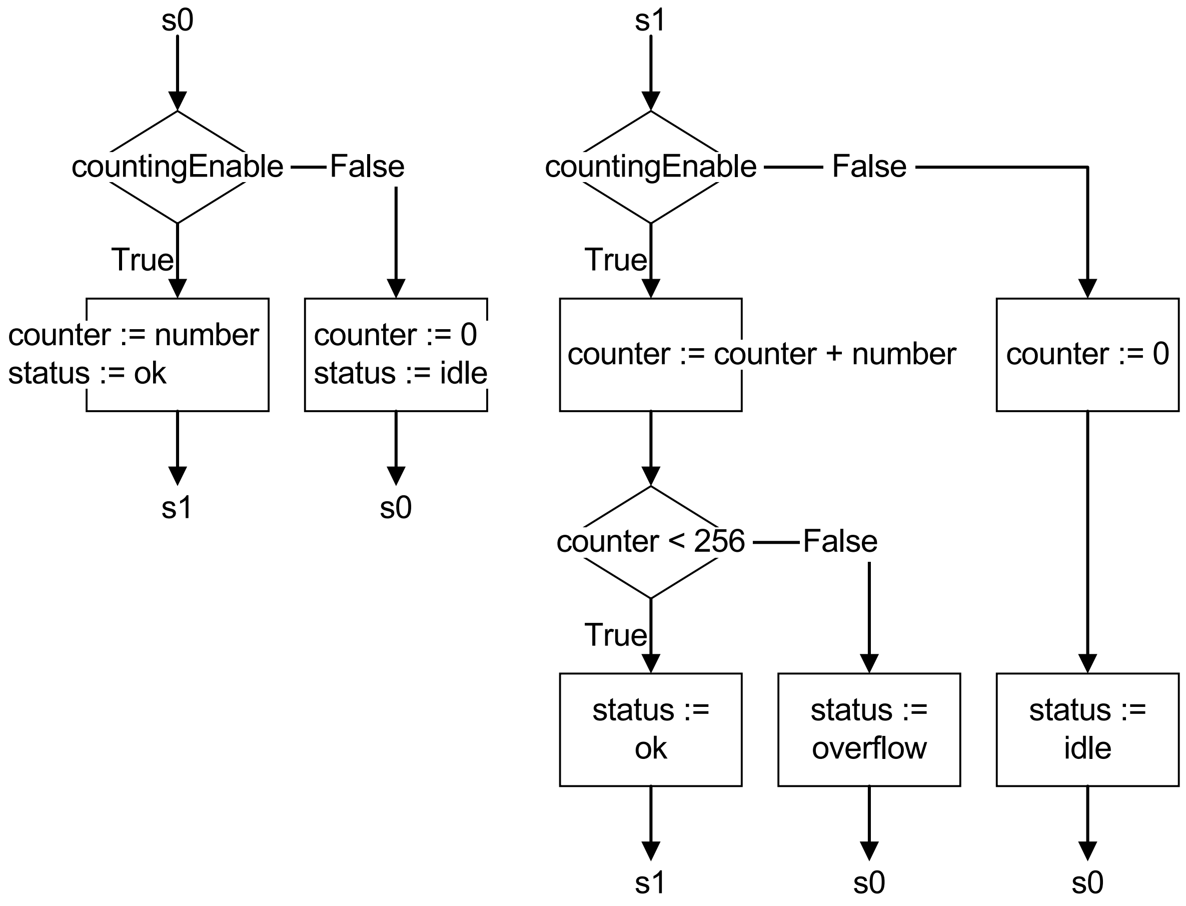


Figure 52: Flow within each state.

The flowchart in figure 52 is then used when creating the VHDL implementation shown on the following two pages.

First, an entity is declared which describes the interface of the block. Then comes the architecture body, which describes the behavior of the block.

As stated in section 4.3.1, all state machines in this study is of the type synchronous Mealy.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ent_Counting is
  port(
    countingEnable : in  std_logic;
    number         : in  std_logic_vector (7 downto 0);
    clk            : in  std_logic;
    arst_n         : in  std_logic;
    counter        : out std_logic_vector (7 downto 0);
    status         : out std_logic_vector (1 downto 0)
  );
end ent_Counting;
```

The entity declaration of the block.

The architecture body is shown as a whole in next page.

```
architecture arch_Counting of ent_Counting is
  type state_type is (s0, s1);
  signal   state    : state_type := s0;
  constant idle     : std_logic_vector(1 downto 0) := "00";
  constant ok       : std_logic_vector(1 downto 0) := "01";
  constant overflow : std_logic_vector(1 downto 0) := "10";

begin
  proc_Counting: process (clk, arst_n)
    variable temp:std_logic_vector(8downto0):=(others=>'0');
  begin
    if arst_n = '0' then      --asynchronous negative reset
      temp    := (others => '0');
      status  <= idle;
      state   <= s0;
      counter <= (others => '0');
    elsif rising_edge(clk) then
      case state is
        when s0 =>
          if countingEnable = '1' then
            temp    := '0' & number;
            status  <= ok;
            state   <= s1;          --next state will be s1
          else
            temp    := (others => '0');
            status  <= idle;
            state   <= s0;          --next state will be s0
          end if;
        when s1 =>
          if countingEnable = '1' then
            temp := temp + ('0' & number);
            if temp(8) = '0' then   --temp < 256
              status <= ok;
              state  <= s1;         --next state will be s1
            else                    --temp >= 256
              status <= overflow;
              state  <= s0;         --next state will be s0
            end if;
          else
            temp   := (others => '0');
            status <= idle;
            state  <= s0;           --next state will be s0
          end if;
      end case;
      counter <= temp(7 downto 0);
    end if;
  end process proc_Counting;
end arch_Counting;
```

The architecture body of the block.

4.4 RTL Simulation

The standard [IEEE 802.3, Clause 4.2.2.1:d.2] declares:

> *"Among processes, no assumptions are made about relative speeds of execution. This means that each interaction between two processes shall be structured to work correctly independent of their respective speeds."*

Since the design consists of several processes, this complicates the verification. However, the processes in the implementation that were defined as functions in the standard are easier to verify since the interaction between a function and its host is clearly specified.

Of the three blocks that were selected in section 4.2 to be implemented, only *BitTransmitter* is specified as a process in the standard. Both *TxCRC32* and *TransmitLinkMgmt* are specified as functions. However, the functions are not always easy to verify either. Such an example is *TransmitLinkMgmt*.

Unlike *TxCRC32*, *TransmitLinkMgmt* interacts with several processes, which makes it hard to tell when the block possesses the correct behavior.

When discussing behavior a distinction can be made between internal and external behavior. The internal behavior is the easier one to verify:

> *"If A and B both are true then C is true, otherwise C is false."*

The external behavior in this implementation can be harder to identify. In the example above, the question arise:

> *"What latency is allowed between that A and B are true until C should be set to true?"*

Since this aspect is not all-over covered by the standard many assumptions have been made, which in the final validation step can turn out to be wrong.

For that reason, the focus in this step was set to verify the internal behavior. It is considered to be possible to change the design in such way that the interaction can be corrected without that the performance and size of a future and correct implementation will differ too much from this implementation.

4.5 Synthesis

This step covers both the selection of device and the synthesis step itself.

4.5.1 Choosing Target Device

There is no need for any hard blocks except memory, which many FPGAs provides. The memory is needed by FIFOs, which are used when crossing clock domains.

Since there is no specific product where this Ethernet controller shall be implemented, it is hard to identify non-functional characteristics that could exclude any FPGA. However, the size of the FPGA has to be sufficient to house the Ethernet controller together with other, future, functions. Another aspect of which FPGA size to select, is its associated routing resources.

An FPGA usually have two different types of routing resources, hierarchical and dedicated. The hierarchical routing resources are different types of global and local resources used for interconnection between blocks. The dedicated resources are used for distribution of e.g. clock signals.

The different types of hierarchical resources are illustrated in figure 53, which refers to a Xilinx Virtex-II device [Xilinx ds031].

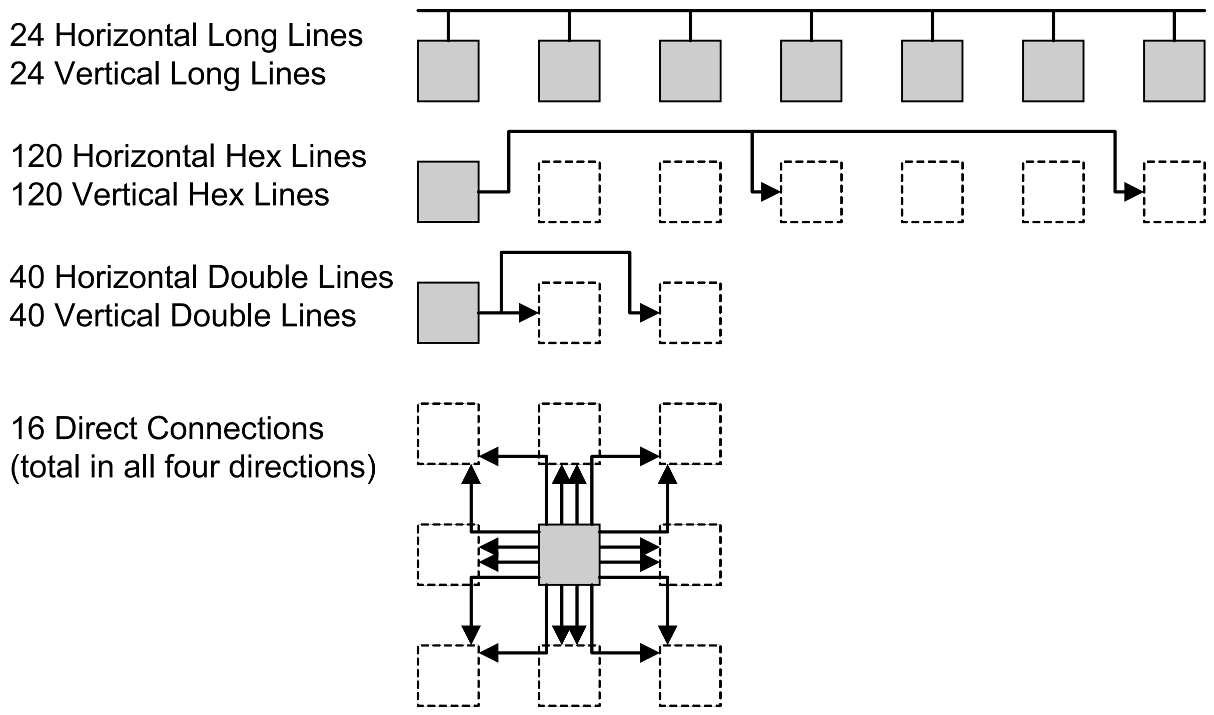


Figure 53: Hierarchical routing resources for each row/column.

If there are many signals in the design that have to be distributed to several blocks, the Long Lines is the ideal routing resource. Nevertheless, this is a strongly limited resource. The Long Lines covers a whole row or column of

CLBs, meaning that in a large FPGA many blocks will be connected to the same Long Line. If there are not sufficient amount of Long Lines available, a shorter type of routing will be used to build long routes. This will lead to a decrease in performance.

The choice is an FPGA from Xilinx's Virtex-II family, which is a rather new family that still grows and that contains both small and large devices with different hard blocks and speed grades. The speed grade used in this project is -4 (which is the slowest, -6 is currently the fastest). By selecting a "slow" device at this stage, the opportunity exists to change to a faster device when implementing a large system where the Ethernet controller is included.

The device XC2V1000-4-ff896 was selected where 1'000 is the number of thousands of equivalent system gates, -4 is the speed grade, ff stands for flip-chip fine-pitch ball grid array (BGA) and 896 is the number of pins of which 432 can be used as I/O-pins. The footprint of this device is identical with the larger devices XC2V1500 and XVC2V2000.

4.5.2 Choosing Synthesis Method

In this study, only parts of the controller will be synthesized. The parts were synthesized one at time. The parts are quite small in the context of synthesis, which results in that the performance will not differ much depending of which optimization target that was used. There are two targets to optimize for as mentioned in section 3.6.2, area or delay. The results are presented in section 5.1 and 5.2. The other choice whether the design should be flattened or not does not have any affect since there is only one block at a time that will be synthesized.

The option to assign different timing criteria to different paths has not been used. Such an example where this option can be used is the signals controlled by the block Initialize (e.g. the signal *extend*), which may be multi cycle paths since they do not change during transmission/reception. This potential of performance improvement is saved for the future.

4.6 Place & Route

The place & route tool (ISE Alliance) allows the user to have a big influence on the result. However, in this work the tool has been used with its default settings. The reason for this is simply that, in the future, when one wants to actually construct a fully functional Ethernet controller with the use of this work there should be a good potential of improvement on this stage to compensate the problems related to a much larger design.

The sizes of the implementations are presented in section 5.1.

4.7 Static Timing Analysis

The synthesis tool (Leonardo Spectrum) and the place & route tool (ISE Alliance) together analyze the static behavior of the implementation. The interest for this study is of course to assure that the timing constraints are fulfilled.

The results are presented in section 5.2.

CHAPTER

5

Results

The results presented in section 5.1 and 5.2 are reported values from the place & route tool ISE Alliance. The results in section 5.3 are estimated values from Xilinx's Virtex-II Power Estimate Worksheet.

Each table in the following sections has two columns of data named O.f.A. and O.f.D. The values in column O.f.A. corresponds from values obtained when the design is optimized for minimum area and O.f.D. when the design is optimized for minimum delay. The choice of optimization target is done in the synthesis step (i.e. the synthesis tool Leonardo Spectrum).

The target device is XC2V1000-4-ff896, which belongs to Xilinx's Virtex-II family. It is a mid-size device with about one million gate equivalents. The speed grade is –4, which makes it the slowest version available in that family.

5.1 Size

The target device (XC2V1000-4-ff896) has 10’240 LUTs.

Table 10: Size of the selected implementations

Block	O.f.A. [LUTs]	O.f.D. [LUTs]
TransmitLinkMgmt	227	239
TxCRC32	135	137
BitTransmitter	110	116

Table 11: Size of additional implementations

Block	O.f.A. [LUTs]	O.f.D. [LUTs]
TransmitFrame	122	131
BurstTimer	37	43
Deference	26	29
Random	31	31
RealTimeDelay	9	11
TxStateReg	39	39

5.2 Performance

Table 12: Performance of the selected implementations

Block	**O.f.A. [MHz]**	**O.f.D. [MHz]**
TransmitLinkMgmt	145.5	141.1
TxCRC32	140.1	146.8
BitTransmitter	162.0	164.0

Table 13: Performance of additional implementations

Block	**O.f.A. [MHz]**	**O.f.D. [MHz]**
TransmitFrame	126.6	131.9
BurstTimer	155.2	143.8
Deference	212.8	211.7
Random	339.3	301.8
RealTimeDelay	305.8	290.1
TxStateReg	265.2	252.3

5.3 Power Dissipation

The values entered into the “Xilinx Virtex-II Power Estimate Worksheet” can be found in appendix D.

Table 14: Power dissipation of the selected implementations

Block	O.f.A. [mW]	O.f.D. [mW]
TransmitLinkMgmt	14	15
TxCRC32	35	36
BitTransmitter	10	11

Table 15: Power dissipation of additional implementations

Block	O.f.A. [mW]	O.f.D. [mW]
TransmitFrame	9	9
BurstTimer	3	3
Deference	1	2
Random	11	11
RealTimeDelay	1	1
TxStateReg	4	4

CHAPTER

6

Discussion

In section 6.1, the results presented in chapter 5 will be discussed. In section 6.2 follows a comparison with an IP core released by Xilinx in section 6.2.

6.1 Reliability and Availability of Obtained Results

After the rough estimation of the hardest blocks to implement *TxCRC32*, *BitTransmitter* and *TransmitLinkMgmt* were selected. As it turned out, the *TransmitFrame* block should has been selected instead of the block *BitTransmitter*. However, since much more design effort has been placed on the latter block the selection might not been wrong.

6.1.1 Size

As table 10 and 11 shows, the difference in area is not big depending of which optimizations target that has been selected.

By adding the area for all blocks in column O.f.A., the sum is 736 LUTs. If the whole *TxMAC* would be implemented the size could be both bigger and smaller. Bigger since the place and route tool not tends to share slices between different blocks and some LUTs may be needed for routing. Smaller since some shared logic may be removed during optimization.

An estimation of the size of the whole controller would be in the range of 3’000 LUTs. The *RxMAC* is considered slightly smaller than the *TxMAC*. The *MIB* together with RS would probably be in parity with the size of the *TxMAC*. The size of the *TxBufMgr* and *RxBufMgr* is hard to estimate, much depending of the size of the buffers and functionality.

A good piece of advice is that never fill an FPGA to more than 80%. Using the same device as before (XC2V1000) and following the advice gives 80% * 10’240 = 8’192 available LUTs. If the Ethernet controller takes 3’000 LUTs, there is room for a design of about 5’200 LUTs, which allows the implementation of a rather big design.

6.1.2 Performance

As can be seen in table 12 and 13, the most critical block is *TransmitFrame*. As pointed out in the beginning of this chapter, this block was not among the selected ones and the design effort of this block is minimal meaning that there surely is room for improvements in the design.

All blocks are implemented using registered outputs, which makes the performance results usable even in the future. The only factor that can decrease the performance is how the routing resources are divided among the blocks. This is not a negligible factor but also, to a certain degree, possible to affect by writing constraint files for the place & route tool.

As suggested in section 8.2, a lot of performance can be gained by dividing the current design into three different versions. The controller can be in three different modes (half duplex, full duplex and full duplex with bursting enabled). The controller is not allowed, by standard, to change mode in runtime meaning that it should be possible to have three different versions of the controller in the ROM, each version corresponding to one mode, and then load a certain version depending of which mode the controller should operate in. This was a miss in the current design, which allows changing of mode within one clock cycle. Still, the controller must be able to change transfer speed (10, 100 or 1’000 Mbps) in runtime.

In ISE Alliance 5.1i, it is possible to add an optional module, Modular Design, which supports Partial Reconfiguration. With this technique is it possible to reconfigure a part of the FPGA, while the device continues to run. This seems to be a very suitable solution in this case.

6.1.3 Power Dissipation

By adding the estimated values for power dissipation in table 14 and 15 for all blocks in column O.f.A., a sum of 92 mW is obtained. Following a similar line of reasoning as in section 6.1 where the total area can be estimated to be four times the area of the transmitter, a power dissipation would be around 370 mW. Add to this a device quiescent power of 183 mW for the device. Each used pin on the device can dissipate around 1 mW.

A rough estimation of the total power dissipation for the device would be around 600 mW.

6.2 Comparison with IP core from Xilinx

In April 2002, Xilinx had the initial release of their first IP core for gigabit Ethernet, DO-DI-1GEMAC.

The key features presented at Xilinx's web as is of January 29, 2003 (www.xilinx.com/systemio/gmac/gmac_desc.htm):

- Single-speed half and/or full-duplex 1 Gbps MAC controller
- Compliant with IEEE 802.3-2000
- Choice of GMII or serial PHY interface options
    - 8b GMII interface running 125MHz for 1Gbps bandwidth
        - 8b wide interface in both Tx and Rx directions
        - Allows direct interfacing between Xilinx FPGAs and industry standard ASSP PHY devices
    - Serial PHY interface integrates PCS/PMA functions for 1.25 Gbps bandwidth
        - Provides a single-chip solutions for 1000BASE-X applications
- 8-bit internal data path and back-end interface
    - 125MHz operation
- Cut-through operation with minimum buffering for maximum flexibility in 64-bit client bus interfacing
- Configured and monitored through an independent microprocessor-neutral interface
- Powerful EtherStats-based statistics gathering
- Optional flow control through MAC Control pause frames; symmetrically or asymmetrically enabled
- MDIO interface to managed objects in PHY layer
- Optional support of VLAN frames to specification IEEE 802.3ac-1998
- Programmable Interframe Gap
- Optional support of "jumbo frames" of any length
- Available under terms of the SignOnce IP License

There are some differences between the design presented in this study (from now on referred to as STU in this section) and above IP-core. First of all, with a few number of modifications the STU is capable of transmitting in both 10, 100 and 1'000 Mbps in comparison with this IP's single-speed capability.

The STU incorporates only the GMII option. The serial PHY interface is only needed for communication over optical medium. The two remarks regarding the

IP-core's GMII ("bit width" and "direct interfacing"), are already covered by the fact that the core is IEEE 802.3 compliant.

Like the IP-core, the STU also operates at 125 MHz and uses an 8-bit internal data path.

The "Cut-through operation…"- and "Configured and monitored…"-features is not, yet, implemented in the STU since this should be done in higher layers.

The statistics gathering feature for the IP-core is also, but yet not fully, implemented in the STU.

The flow control should be implemented in the MAC Control sublayer, which is not within the scoop of this study.

The STU is also capable of handling VLAN-tags as well as the IP-core.

Programmable Interframe Gap is a feature that not has been implemented in the STU and, at this state, hard to see the need of since the design already capable of handling pause frames.

Jumbo frames are not supported by the STU.

According to the documentation of the IP-core, the size of it varies between 625 and 1'777 slices depending of device and optional features. Translated to LUTs, this means a size between 1'250 and 3'554 LUTs, which can be compared with the estimated size of 3'000 LUTs for the STU.

To summarize above comparison, the STU is in parity with Xilinx IP core regarding functionality, size and choice of platform (Virtex-II).

CHAPTER

7

Conclusion

It would be fully possible to implement a gigabit Ethernet controller using an FPGA. Even though many parts of the controller have not been implemented and nothing else but the controller has been implemented in the FPGA, it is beyond reasonable doubt that it should not be possible.

– This page was intentionally left blank –

CHAPTER

8

Recommendations

8.1 Status of Work

At this moment, three out of seven main parts of the system are implemented. These are the transmission part of the MAC sublayer, RS and STA.

Guidelines for verification do only exist for the highest level, i.e. how a complete implementation shall react in a number of different situations. This means that the blocks that have been implemented during this project have not been possible to verify fully complete. The verification process in this project has instead focused on the functional behavior of the blocks.

8.2 Future Work

As stated in previous section, four main parts of the complete Ethernet controller remains to be implemented. These are the receive part of the MAC sublayer, the transmit and receive buffers and the host interface.

During the development of the blocks that were implemented many discoveries were made about how the system is intended to work, which made the author to realize that the partition of the system could have been much better in order to achieve a simpler design. Some of these observations follow:

- Make a clear division between the data path and the control path when the frame is assembled in the MAC sublayer. Now, a lot of data is unnecessarily moved back and forward. By dividing into data and control paths, the timing is much easier to manage.
- There are three different clock signals present in the MAC sublayer. These could be generated by the RS. Now, RS only generates the transmit clock for GMII. By letting RS generates all clock signals used between the MII/GMII and the MAC client, the design should be easier.

- There are signals present in the transmit part of the MAC sublayer that are never used. Some of these signals are identified in section 4.3.

The use of the signal `wasTransmitting` was not consequent in the standard IEEE 802.3 year 2000 edition. It was mentioned in the summary of the services provided by the Physical layer [IEEE 802.3, Clause 4.2.7.4], but then not declared in the definition of the services [IEEE 802.3, Clause 4.3.3], which the summary refers. The signal was assumed similar with the signal `transmitting` and implemented in that way, which turned out to be wrong. Later, the year 2002 edition of the standard was released in which this had been corrected and the variable is now declared as a shared variable within the transmitter.

CHAPTER

9

Acknowledgements

First, I would like to thank my supervisor at Enea Epact, Åke Andersson for great tutoring and support during the project.

I would also like to thank the staff at the Embedded System Division at Enea Epact for a nice stay and especially Lars Asplund and Mathias Bergvall for sharing their great knowledge.

Finally, I would like to thank my examiner at the Computer Engineering Division within the Department of Electrical Engineering, professor Dake Liu.

– This page was intentionally left blank –

CHAPTER

10

References

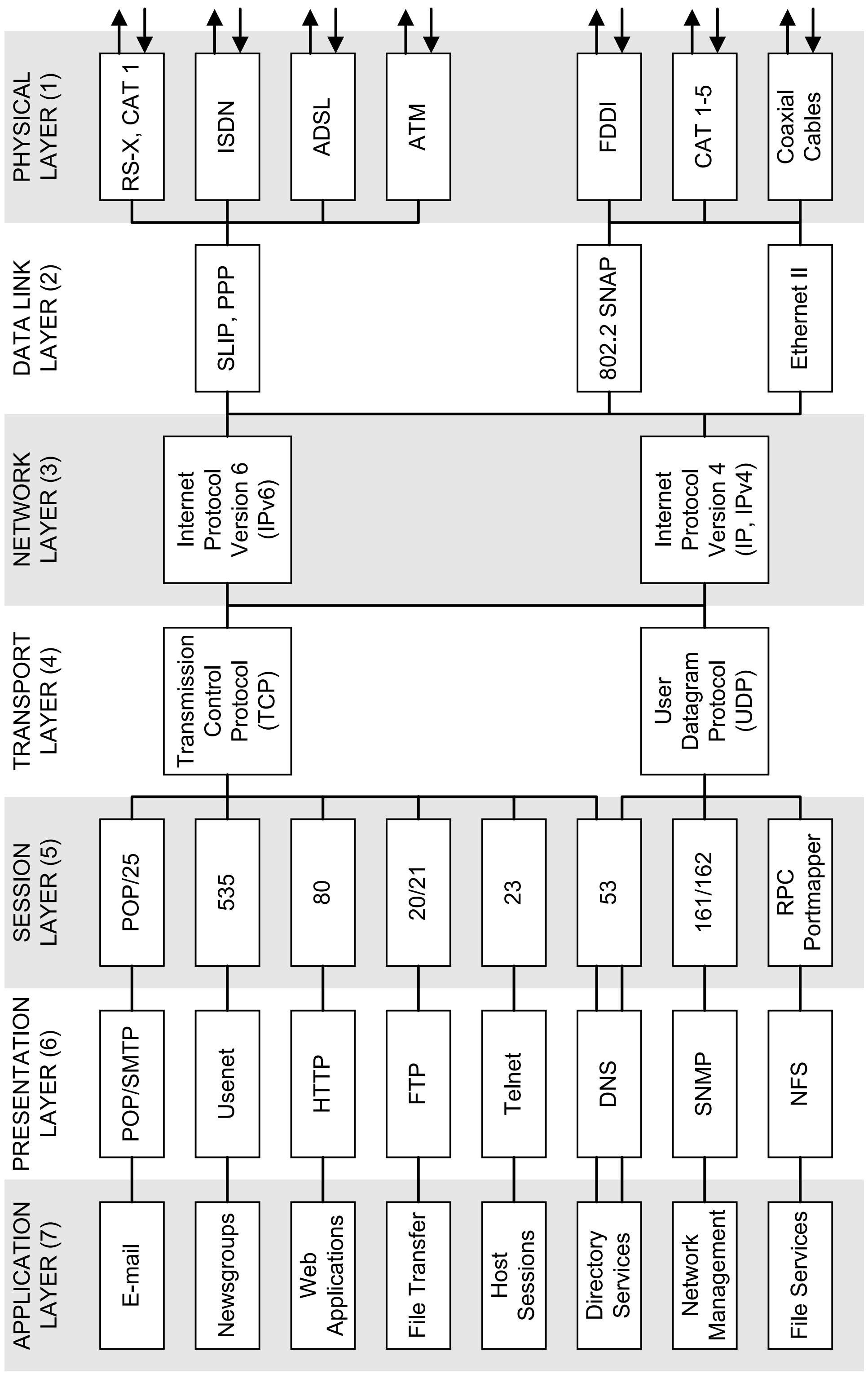
[Boehm 1980]	Boehm, Barry W. and Wolverton, R. W. (1980). "Software cost modeling: some lessons learned", The Journal of Systems and Software, vol.1, num.3, pp.195-201
[IEEE 802.3]	Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specification, IEEE Std 802.3-2002, IEEE, 2002
[XAPP209]	IEEE 802.3 Cyclic Redundancy Check, Xilinx, 2001
[XAPP258]	FIFOs Using Virtex-II Block RAM, Xilinx, 2001
[Xilinx ds031]	Virtex-II Platform FPGAs: Detailed Description, Xilinx, 2002

– This page was intentionally left blank –

APPENDIX A

Protocols

SELECTION OF PROTOCOLS MAPPED TO THE OSI/BR MODEL



– This page was intentionally left blank –

APPENDIX

B

EDA Software

The following EDA software have been used throughout the project:

- HDL PD Transition – HDL Designer Series 2001.3 (Mentor Graphics)
  Design entry tool
- Leonardo Spectrum LS2001.1b.12 (Exemplar Logic)
  Synthesis tool
- ISE Alliance 4.1i (Xilinx)
  Implementation and configuration tool
- ModelSim PE 5.5a (Model Technology)
  Simulation and verification tool

When estimating the power dissipation the following tool was used:

- Xilinx Virtex-II Power Estimate Worksheet, version 1.05 (Xilinx)

– This page was intentionally left blank –

APPENDIX

C

Signal Table for MAC Sublayer

Table begins on next page.

Tx

MAC :: Common Constants		Value	[unit]	Defined	Initialize	CRC32	nothing	TransmitFrame	TransmitDataEncap	ComputePad	TransmitLinkMgmt	StartTransmit	WatchForCollision	Random	BackOff	BurstTimer	Deference	StartRealTimeDelay	RealTimeDelay	BitTransmitter	PhysicalSignalEncap	InterFrameSignal	NextBit	StartJam	LayerMgmtInitialize	IncLargeCounter	SumLarge	LayerMgmtTransmitCounters	DeferTest	CarrierSenseTest	MAC :: Common Constants
addressSize		48	[b]	4.2.7.1																					- -	- -	- -	- -	- -	- -	addressSize
clientDataSize		...	[b]	4.2.7.1																					- -	- -	- -	- -	- -	- -	clientDataSize
crcSize		32	[b]	4.2.7.1																					- -	- -	- -	- -	- -	- -	crcSize
dataSize		clientDataSize + padSize	[b]	4.2.7.1																					- -	- -	- - R	- -	- -	- -	dataSize
extend	(rem 3)	Boolean	[-]	4.2.7.1	R						R														- -	- -	- -	- -	- -	- -	extend
extensionBit		EXTEND	[-]	4.2.7.1																R		R			- -	- -	- -	- -	- -	- -	extensionBit
extensionErrorBit		EXTEND_ERROR	[-]	4.2.7.1																R		R			- -	- -	- -	- -	- -	- -	extensionErrorBit
frameSize		144 + dataSize	[b]	4.2.7.1								R													- -	- -	- -	- -	- -	- -	frameSize
headerSize		64	[b]	4.2.7.1								R	R												- -	- -	- -	- -	- -	- -	headerSize
lengthOrTypeSize		16	[b]	4.2.7.1																					- -	- -	- -	- -	- -	- -	lengthOrTypeSize
maxUntaggedFrameSize		1518	[B]	4.2.7.1																					- -	- -	- -	- -	- -	- -	maxUntaggedFrameSize
maxValidFrame		1500	[B]	4.2.7.1																					- -	- -	- -	- -	- -	- -	maxValidFrame
minFrameSize		512	[b]	4.2.7.1																					- -	- -	- -	- -	- -	- -	minFrameSize
minTypeValue		1536	[-]	4.2.7.1																					- -	- -	- -	- -	- -	- -	minTypeValue
padSize		max{0, 368 - clientDataSize}	[b]	4.2.7.1						R															- -	- -	- -	- -	- -	- -	padSize
preambleSize		56	[b]	4.2.7.1																					- -	- -	- -	- -	- -	- -	preambleSize
qTagPrefixSize		4	[B]	4.2.7.1																					- -	- -	- -	- -	- -	- -	qTagPrefixSize
sfdSize		8	[b]	4.2.7.1																					- -	- -	- -	- -	- -	- -	sfdSize
slotTime	(rem 1)	512 / 512 / 4096	[bT]	4.2.7.1									R		R								R		- -	- -	- -	- -	- -	- -	slotTime
MAC :: Common Variables		**Type**		**Defined**																											**MAC :: Common Variables**
halfDuplex		Boolean		4.2.7.1	C S R						R						R						R		- -	- -	- -	- - R	- -	- -	halfDuplex
MAC :: Common Procedures				**Defined**																											**MAC :: Common Procedures**
Initialize	(rem 4)			4.2.7.5	- - -	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	Initialize
nothing				4.2.10	X		- - -				X					X	X											X		X	nothing
MAC :: Common Functions		**Type of value returned**		**Defined**																											**MAC :: Common Functions**
CRC32		CRCValue		4.2.10	- - -	- - -			X																★	★	★	★	★	★	CRC32
MAC :: Transmit Constants		**Value**	**[unit]**	**Defined**																											**MAC :: Transmit Constants**
attemptLimit		16	[-]	4.2.7.2							R														- -	- -	- -	- - R	- -	- -	attemptLimit
backOffLimit		10	[-]	4.2.7.2											R										- -	- -	- -	- -	- -	- -	backOffLimit
burstLimit	(rem 1)	- / - / 65 536	[b]	4.2.7.2												R									- -	- -	- -	- -	- -	- -	burstLimit
interFrameSize		96	[b]	4.2.7.2																		R			- -	- -	- -	- -	- -	- -	interFrameSize
interFrameSpacing		96	[bT]	4.2.7.2													R								- -	- -	- -	- -	- -	- -	interFrameSpacing
interFrameSpacingPart1		[0, 64]	[bT]	4.2.7.2													R								- -	- -	- -	- -	- -	- -	interFrameSpacingPart1
interFrameSpacingPart2		[64, 96]	[bT]	4.2.7.2													R								- -	- -	- -	- -	- -	- -	interFrameSpacingPart2
jamSize		32	[b]	4.2.7.2																		R		R	- -	- -	- -	- -	- -	- -	jamSize
MAC :: Transmit Variables		**Type**		**Defined**																											**MAC :: Transmit Variables**
attempts		Integer[0, attemptLimit]		4.2.7.2							C S R				R										- -	- -	- -	- - R	- -	- -	attempts
burstCounter		Integer[0, burstLimit]		4.2.8	★	★	★	★	★	★	★	★	★	★	★	C S R	★	★	★	★	★	★	★	★	★	★	★	★	★	★	burstCounter
bursting		Boolean		4.2.7.2	C						S R		C			C R				C S R		C			- -	- -	- -	- -	- -	- -	bursting
burstMode		Boolean		4.2.7.2	C S						R		R												- -	- -	- -	- -	- -	- -	burstMode
burstStart		Boolean		4.2.7.2							C S		R										R		- -	- -	- -	- -	- -	- -	burstStart
currentTransmitBit		Integer[1, frameSize]		4.2.7.2								C	R							R	C S R		S R	C R	- -	- -	- -	- -	- -	- -	currentTransmitBit
deferring		Boolean		4.2.7.2	C						R						C S								- -	- -	- -	- -	- -	- -	deferring
extendError		Boolean		4.2.7.2																C R		S R		C S	- -	- -	- -	- -	- -	- -	extendError
frameWaiting		Boolean		4.2.7.2	C						C S R						R			R					- -	- -	- -	- -	- - R	- -	frameWaiting
interFrameCount		Integer[0, interFrameTotal]		4.2.8	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C S R	★	★	★	★	★	★	★	★	interFrameCount
interFrameTotal		Integer{interFrameSize, jamSize}		4.2.8	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	S R	★	★	★	★	★	★	★	★	interFrameTotal
lastHeaderBit		Integer[1, headerSize]		4.2.7.2								S									R				- -	- -	- -	- -	- -	- -	lastHeaderBit
lastTransmitBit		Integer[1, frameSize]		4.2.7.2								S								R			R	S R	- -	- -	- -	- -	- -	- -	lastTransmitBit
maxBackOff		Integer[2, 1024]		4.2.8	★	★	★	★	★	★	★	★	★	★	C S R	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	maxBackOff
newCollison		Boolean		4.2.7.2	C								S							R	R			C	- -	- -	- -	- -	- -	- -	newCollison
outgoingFrame		Frame		4.2.7.2					S											R					- -	- -	- -	- - R	- -	- -	outgoingFrame
outgoingHeader		Header		4.2.7.2					S												R				- -	- -	- -	- -	- -	- -	outgoingHeader
transmitSucceeding		Boolean		4.2.7.2							C R	S	C R										R		- -	- -	- -	- - R	- -	- -	transmitSucceeding
wasTransmitting		Boolean		4.2.7.2													C S R								- -	- -	- -	- -	- -	- -	wasTransmitting
MAC :: Transmit Procedures				**Defined**																											**MAC :: Transmit Procedures**
BackOff				4.2.8	- - -						X				- - -										★	★	★	★	★	★	BackOff
InterFrameSignal				4.2.8	- - -															X		- - -			★	★	★	★	★	★	InterFrameSignal
NextBit				4.2.8	- - -															X			- - -		★	★	★	★	★	★	NextBit
PhysicalSignalEncap				4.2.8	- - -															X	- - -				★	★	★	★	★	★	PhysicalSignalEncap
StartJam				4.2.8	- - -															X	X			- - -	★	★	★	★	★	★	StartJam
StartRealTimeDelay				4.2.8	- - -												X	- - -							★	★	★	★	★	★	StartRealTimeDelay
StartTransmit				4.2.8	- - -						X	- - -													★	★	★	★	★	★	StartTransmit
TransmitDataEncap				4.2.8	- - -			X	- - -																★	★	★	★	★	★	TransmitDataEncap
WatchForCollision				4.2.8	- - -						X		- - -												★	★	★	★	★	★	WatchForCollision

				Initialize	CRC32	nothing	TransmitFrame	TransmitDataEncap	ComputePad	TransmitLinkMgmt	StartTransmit	WatchForCollision	Random	BackOff	BurstTimer	Deference	StartRealTimeDelay	RealTimeDelay	BitTransmitter	PhysicalSignalEncap	InterFrameSignal	NextBit	StartJam	LayerMgmtInitialize	IncLargeCounter	SumLarge	LayerMgmtTransmitCounters	DeferTest	CarrierSenseTest	
MAC :: Transmit Functions	**Type of value returned**		**Defined**																											**MAC :: Transmit Functions**
ComputePad	DataValue		4.2.8	- - -				X	- - -															★	★	★	★	★	★	ComputePad
Random	Integer[0, maxBackOff - 1]		4.2.8	- - -									- - -	X										★	★	★	★	★	★	Random
RealTimeDelay	Boolean		4.2.8	- - -												X		- - -						★	★	★	★	★	★	RealTimeDelay
TransmitFrame *(rem 4)*	TransmitStatus		4.2.8	★	★	★	- - -	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	TransmitFrame
TransmitLinkMgmt	TransmitStatus		4.2.8	- - -			X			- - -														★	★	★	★	★	★	TransmitLinkMgmt
MAC :: Transmit Processes			**Defined**																											**MAC :: Transmit Processes**
BurstTimer			4.2.8	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	★	★	★	★	★	★	BurstTimer
Deference			4.2.8	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	★	★	★	★	★	★	Deference
BitTransmitter			4.2.8	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	★	★	★	★	★	★	BitTransmitter
PHY Layer Interface :: Transmit Variables	**Type**		**Defined**																											**PHY Layer Interface :: Transmit Variables**
carrierSense	Boolean		4.3.3	- - R	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -	- - R	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -	- - R	carrierSense
collisionDetect	Boolean		4.3.3	- -	- -	- -	- -	- -	- -	- -	- -	- - R	- -	- -	- -	- -	- -	- -	- -	- -	- - R	- -	- -	- -	- -	- -	- -	- -	- - R	collisionDetect
transmitting	Boolean		4.3.3	C						R	S					R			C R			C S		- -	- -	- -	- -	- - R	- - R	transmitting
PHY Layer Interface :: Transmit Procedures			**Defined**																											**PHY Layer Interface :: Transmit Procedures**
TransmitBit			4.3.3	- - -															X	X	X			★	★	★	★	★	★	TransmitBit
Wait			4.3.3	- - -										X	X									- - -				X		Wait
DTE Layer Mgmt :: Common Constants	**Value**	**[unit]**	**Defined**																											**DTE Layer Mgmt :: Common Constants**
max64	2^64 - 1	[-]	5.2.4.1																											max64
maxDeferTime *(rem 1)*	24 288 / 24 288 / 155 488	[bT]	5.2.4.1																									R		maxDeferTime
maxLarge	2^32 - 1	[-]	5.2.4.1																											maxLarge
oneBitTime	1	[bT]	5.2.4.1																											oneBitTime
DTE Layer Mgmt :: Common Procedures			**Defined**																											**DTE Layer Mgmt :: Common Procedures**
IncLargeCounter			5.2.4.4	- - -															X					- - -	- - -		X			IncLargeCounter
LayerMgmtInitialize *(rem 4)*			5.2.4.4	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	- - -	★	★	★	★	★	LayerMgmtInitialize
SumLarge			5.2.4.4	- - -																				- - -		- - -	X			SumLarge
DTE Layer Mgmt :: Transmit Variables	**Type**		**Defined**																											**DTE Layer Mgmt :: Transmit Variables**
carrierSenseFailure	Boolean		5.2.4.2																					C			R		C S	carrierSenseFailure
carrierSenseTestDone	Boolean		5.2.4.2																					C			R		C S	carrierSenseTestDone
carrierSeen	Boolean		5.2.4.2	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C S R	carrierSeen
collisionSeen	Boolean		5.2.4.2	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C S R	collisionSeen
deferred	Boolean		5.2.4.2							C S														C			R			deferred
deferBitTimer	Integer[0, maxDeferTime]		5.2.4.2	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C S R	★	deferBitTimer
excessDefer	Boolean		5.2.4.2							C														C			R	S R		excessDefer
lateCollisionCount	Integer[0, attemptLimit - 1]		5.2.4.2							C S R									S R					C		R	R			lateCollisionCount
lateCollisionError	Boolean		5.2.4.2							C R		S												C						lateCollisionError
transmitEnabled *(rem 2)*	Boolean		5.2.4.2				R																	S						transmitEnabled
//MAC transmit counters//																														//MAC transmit counters//
broadcastFramesTransmittedOK	CounterLarge		5.2.4.2	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C	S R					broadcastFramesTransmittedOK
collisionFrames	*array* [1, attemptLimit - 1] *of* CounterLarge		5.2.4.2	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C	S R					collisionFrames
deferredTransmissions	CounterLarge		5.2.4.2	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C	S R					deferredTransmissions
framesTransmittedOK	CounterLarge		5.2.4.2	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C	S R					framesTransmittedOK
multicastFramesTransmittedOK	CounterLarge		5.2.4.2	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C	S R					multicastFramesTransmittedOK
multipleCollisionFrames	CounterLarge		5.2.4.2	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C	S R					multipleCollisionFrames
octetsTransmittedOK	CounterLarge		5.2.4.2	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C		S R				octetsTransmittedOK
singleCollisionFrames	CounterLarge		5.2.4.2	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C	S R					singleCollisionFrames
//MAC transmit error counters//																														//MAC transmit error counters//
carrierSenseErrors	CounterLarge		5.2.4.2	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C	S R					carrierSenseErrors
excessiveCollision	CounterLarge		5.2.4.2	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C	S R					excessiveCollision
excessiveDeferral	CounterLarge		5.2.4.2	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C	S R					excessiveDeferral
lateCollision	CounterLarge		5.2.4.2	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	C		S R				lateCollision
DTE Layer Mgmt :: Transmit Procedures			**Defined**																											**DTE Layer Mgmt :: Transmit Procedures**
LayerMgmtTransmitCounters			5.2.4.2	- - -						X														- - -			- - -			LayerMgmtTransmitCounters
DTE Layer Mgmt :: Transmit Processes			**Defined**																											**DTE Layer Mgmt :: Transmit Processes**
CarrierSenseTest			5.2.4.2	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	CarrierSenseTest
DeferTest			5.2.4.2	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	DeferTest

C	= Clear
S	= Set
R	= Read
X	= Execute
-	= Not Applicable
★	= Not Accessible

rem 1	= The values refers to the speeds 10 Mbps / 100 Mbps / 1000 Mbps respectively.
rem 2	= This signal is cleared by higher layers.
rem 3	= This signal can be used to decide whether the system is in gigabit mode or not.
rem 4	= This procedure/function is executed by higher layers.

The sectionnumbers in column **Defined** referes to IEEE 802.3:2002.

Rx

					Initialize	CRC32	nothing	ReceiveFrame	ReceiveDataDecap	RecognizeAddress	RemovePad	ReceiveLinkMgmt	StartReceive	BitReceiver	PhysicalSignalDecap	SetExtending	LayerMgmtInitialize	IncLargeCounter	SumLarge	LayerMgmtReceiveCounters	LayerMgmtRecognizeAddresses	
MAC :: Common Constants		**Value**	**[unit]**	**Defined**																		**MAC :: Common Constants**
addressSize		48	[b]	4.2.7.1													- -	- -	- -	- -	- -	addressSize
clientDataSize		...	[b]	4.2.7.1													- -	- -	- -	- - R	- -	clientDataSize
crcSize		32	[b]	4.2.7.1													- -	- -	- -	- -	- -	crcSize
dataSize		clientDataSize + padSize	[b]	4.2.7.1													- -	- -	- - R	- -	- -	dataSize
extend	*(rem 3)*	Boolean	[-]	4.2.7.1	R											R	- -	- -	- -	- -	- -	extend
extensionBit		EXTEND	[-]	4.2.7.1													- -	- -	- -	- -	- -	extensionBit
extensionErrorBit		EXTEND_ERROR	[-]	4.2.7.1													- -	- -	- -	- -	- -	extensionErrorBit
frameSize		144 + dataSize	[b]	4.2.7.1					R			S R		S			- -	- -	- -	- -	- -	frameSize
headerSize		64	[b]	4.2.7.1													- -	- -	- -	- -	- -	headerSize
lengthOrTypeSize		16	[b]	4.2.7.1													- -	- -	- -	- -	- -	lengthOrTypeSize
maxUntaggedFrameSize		1518	[B]	4.2.7.1					R								- -	- -	- -	- -	- -	maxUntaggedFrameSize
maxValidFrame		1500	[B]	4.2.7.1							R						- -	- -	- -	- - R	- -	maxValidFrame
minFrameSize		512	[b]	4.2.7.1								R					- -	- -	- -	- - R	- -	minFrameSize
minTypeValue		1536	[-]	4.2.7.1							R						- -	- -	- -	- -	- -	minTypeValue
padSize		max{0, 368 - clientDataSize}	[b]	4.2.7.1													- -	- -	- -	- -	- -	padSize
preambleSize		56	[b]	4.2.7.1													- -	- -	- -	- -	- -	preambleSize
qTagPrefixSize		4	[B]	4.2.7.1					R								- -	- -	- -	- -	- -	qTagPrefixSize
sfdSize		8	[b]	4.2.7.1													- -	- -	- -	- -	- -	sfdSize
slotTime	*(rem 1)*	512 / 512 / 4096	[bT]	4.2.7.1										R			- -	- -	- -	- -	- -	slotTime
MAC :: Common Variables		**Type**		**Defined**																		**MAC :: Common Variables**
halfDuplex		Boolean		4.2.7.1	C S R											R	- -	- -	- -	- -	- -	halfDuplex
MAC :: Common Procedures				**Defined**																		**MAC :: Common Procedures**
Intialize	*(rem 4)*			4.2.7.5	- - -	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	★	Initialize
nothing				4.2.10	X		- - -					X								X		nothing
MAC :: Common Functions		**Type of value returned**		**Defined**																		**MAC :: Common Functions**
CRC32		CRCValue		4.2.10	- - -	- - -			X								★	★	★	★	★	CRC32
MAC :: Receive Variables		**Type**		**Defined**																		**MAC :: Receive Variables**
b		PhysicalBit		4.2.9	★	★	★	★	★	★	★	★	★	C S R	★	★	★	★	★	★	★	b
currentReceiveBit		Integer[1, frameSize]		4.2.9	★	★	★	★	★	★	★	★	★	C S R	★	★	★	★	★	★	★	currentReceiveBit
enableBitReceiver		Boolean		4.2.9	★	★	★	★	★	★	★	★	★	C S R	★	★	★	★	★	★	★	enableBitReceiver
exceedsMaxLength		Boolean		4.2.7.3					C S R								- -	- -	- -	- -	- -	exceedsMaxLength
excessBits		Integer[0, 7]		4.2.7.3					R			S					- -	- -	- -	- -	- -	excessBits
extending		Boolean		4.2.7.3	C S									C R		C S	- -	- -	- -	- -	- -	extending
extensionOk		Boolean		4.2.7.3					R					C S			- -	- -	- -	- -	- -	extensionOk
frameFinished		Boolean		4.2.9	★	★	★	★	★	★	★	★	★	C S R	★	★	★	★	★	★	★	frameFinished
incomingFrame		Frame		4.2.7.3					R					S			- -	- -	- -	- - R	- -	incomingFrame
incomingFrameSize		Integer[0, ...]		4.2.9	★	★	★	★	★	★	★	★	★	C S R	★	★	★	★	★	★	★	incomingFrameSize
receiveSucceeding		Boolean		4.2.7.3				R	C S R			C S R	S	C S			- -	- -	- -	- -	- -	receiveSucceeding
receiving		Boolean		4.2.7.3	C							R	S	C R			- -	- -	- -	- -	- -	receiving
status		ReceiveStatus		4.2.9	★	★	★	★	S R	★	★	★	★	★	★	★	★	★	★	★	★	status
validLength		Boolean		4.2.7.3					R		C S R						- -	- -	- -	- -	- -	validLength
MAC :: Receive Procedures				**Defined**																		**MAC :: Receive Procedures**
ReceiveLinkMgmt				4.2.9	- - -			X				- - -					★	★	★	★	★	ReceiveLinkMgmt
StartReceive				4.2.9	- - -							X	- - -				★	★	★	★	★	StartReceive
PhysicalSignalDecap				4.2.9	- - -									X	- - -		★	★	★	★	★	PhysicalSignalDecap
MAC :: Receive Functions		**Type of value returned**		**Defined**																		**MAC :: Receive Functions**
ReceiveFrame	*(rem 4)*	ReceiveStatus		4.2.9	★	★	★	- - -	★	★	★	★	★	★	★	★	★	★	★	★	★	ReceiveFrame
ReceiveDataDecap		ReceiveStatus		4.2.9	- - -			X	- - -								★	★	★	★	★	ReceiveDataDecap
RecognizeAddress		Boolean		4.2.9	- - -				X	- - -							★	★	★	★	★	RecognizeAddress
RemovePad		DataValue		4.2.9	- - -				X		- - -						★	★	★	★	★	RemovePad
MAC :: Receive Processes				**Defined**																		**MAC :: Receive Processes**
BitReceiver				4.2.9	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	★	★	★	★	★	BitReceiver
SetExtending				4.2.9	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	★	★	★	★	★	SetExtending
PHY Layer Interface :: Receive Variables		**Type**		**Defined**																		**PHY Layer Interface :: Receive Variables**
receiveDataValid		Boolean		4.3.3	- - R	- -		- -	- -	- -	- -	- -	- -	- - R	- -	- - R	- -	- -	- -	- -	- -	receiveDataValid
PHY Layer Interface :: Receive Functions		**Type of value returned**		**Defined**																		**PHY Layer Interface :: Receive Functions**
ReceiveBit		PhysicalBit		4.3.3	- - -												★	★	★	★	★	ReceiveBit
DTE Layer Mgmt :: Common Constants		**Value**	**[unit]**	**Defined**																		**DTE Layer Mgmt :: Common Constants**
max64		2^64 - 1	[-]	5.2.4.1																		max64
maxDeferTime	*(rem 1)*	24 288 / 24 288 / 155 488	[bT]	5.2.4.1																		maxDeferTime
maxLarge		2^32 - 1	[-]	5.2.4.1																		maxLarge
oneBitTime		1	[bT]	5.2.4.1																		oneBitTime

			Initialize	CRC32	nothing	ReceiveFrame	ReceiveDataDecap	RecognizeAddress	RemovePad	ReceiveLinkMgmt	StartReceive	BitReceiver	PhysicalSignalDecap	SetExtending	LayerMgmtInitialize	IncLargeCounter	SumLarge	LayerMgmtReceiveCounters	LayerMgmtRecognizeAddresses	
DTE Layer Mgmt :: Common Procedures		**Defined**																		**DTE Layer Mgmt :: Common Procedures**
IncLargeCounter		5.2.4.4	- - -												- - -	- - -		X		IncLargeCounter
LayerMgmtInitialize *(rem 4)*		5.2.4.4	★	★	★	★	★	★	★	★	★	★	★	★	- - -	★	★	★	★	LayerMgmtInitialize
SumLarge		5.2.4.4	- - -												- - -		- - -	X		SumLarge
DTE Layer Mgmt :: Receive Variables	**Type**	**Defined**																		**DTE Layer Mgmt :: Receive Variables**
receiveEnabled *(rem 2)*	Boolean	5.2.4.3				R						R			S					receiveEnabled
//MAC receive counters//																				//MAC receive counters//
framesReceivedOK	CounterLarge	5.2.4.3	★	★	★	★	★	★	★	★	★	★	★	★	C	S R				framesReceivedOK
octetsReceivedOK	CounterLarge	5.2.4.3	★	★	★	★	★	★	★	★	★	★	★	★	C		S R			octetsReceivedOK
//MAC receive error counters//																				//MAC receive error counters//
alignmentErrors	CounterLarge	5.2.4.3	★	★	★	★	★	★	★	★	★	★	★	★	C	S R				alignmentErrors
frameCheckSequenceErrors	CounterLarge	5.2.4.3	★	★	★	★	★	★	★	★	★	★	★	★	C	S R				frameCheckSequenceErrors
frameTooLongErrors	CounterLarge	5.2.4.3	★	★	★	★	★	★	★	★	★	★	★	★	C	S R				frameTooLongErrors
inRangeLengthErrors	CounterLarge	5.2.4.3	★	★	★	★	★	★	★	★	★	★	★	★	C	S R				inRangeLengthErrors
outOfRangeLengthField	CounterLarge	5.2.4.3	★	★	★	★	★	★	★	★	★	★	★	★	C	S R				outOfRangeLengthField
//MAC receive address counters//																				//MAC receive address counters//
broadcastFramesReceivedOK	CounterLarge	5.2.4.3	★	★	★	★	★	★	★	★	★	★	★	★	C	S R				broadcastFramesReceivedOK
multicastFramesReceivedOK	CounterLarge	5.2.4.3	★	★	★	★	★	★	★	★	★	★	★	★	C	S R				multicastFramesReceivedOK
DTE Layer Mgmt :: Receive Procedures		**Defined**																		**DTE Layer Mgmt :: Receive Procedures**
LayerMgmtReceiveCounters		5.2.4.3	- - -				X								★	★	★	- - -	★	LayerMgmtReceiveCounters
DTE Layer Mgmt :: Receive Functions	**Type of value returned**	**Defined**																		**DTE Layer Mgmt :: Receive Functions**
LayerMgmtRecognizeAddresses	Boolean	5.2.4.3	- - -				X								★	★	★	★	- - -	LayerMgmtRecognizeAddresses

C	= Clear	***rem 1***	= The values refers to the speeds 10 Mbps / 100 Mbps / 1000 Mbps respectively.
S	= Set	***rem 2***	= This signal is cleared by higher layers.
R	= Read	***rem 3***	= This signal can be used to decide whether the system is in gigabit mode or not.
X	= Execute	***rem 4***	= This procedure/function is executed by higher layers.
-	= Not Applicable		
★	= Not Accessible		The sectionnumbers in column **Defined** referes to IEEE 802.3:2002.

– This page was intentionally left blank –

APPENDIX

D

Power Dissipation

CLB Logic Power

Name	Frequency (MHz)	Total Number of CLB Slices	Total Number of Flip/Flop or Latches	Total Number of Shift Register LUTs	Total Number of Select RAM LUTs	Average Toggle Rate %	Amount of Routing Used	VCCint Subtotal (mW)
TransmitLinkMgmt	125	125	37	0	0	12%	Medium	**14**
TxCRC32	125	69	33	0	0	55%	Medium	**35**
BitTransmitter	125	58	12	0	0	12%	High	**10**
TransmitFrame	125	70	38	0	0	12%	Medium	**9**
BurstTimer	125	20	23	0	0	12%	Medium	**3**
Deference	125	15	9	0	0	12%	Medium	**1**
Random	125	21	27	0	0	55%	Medium	**11**
RealTimeDelay	125	8	7	0	0	12%	Medium	**1**
TxStateReg	125	31	36	0	0	12%	Medium	**4**
							Total	**92**

– This page was intentionally left blank –