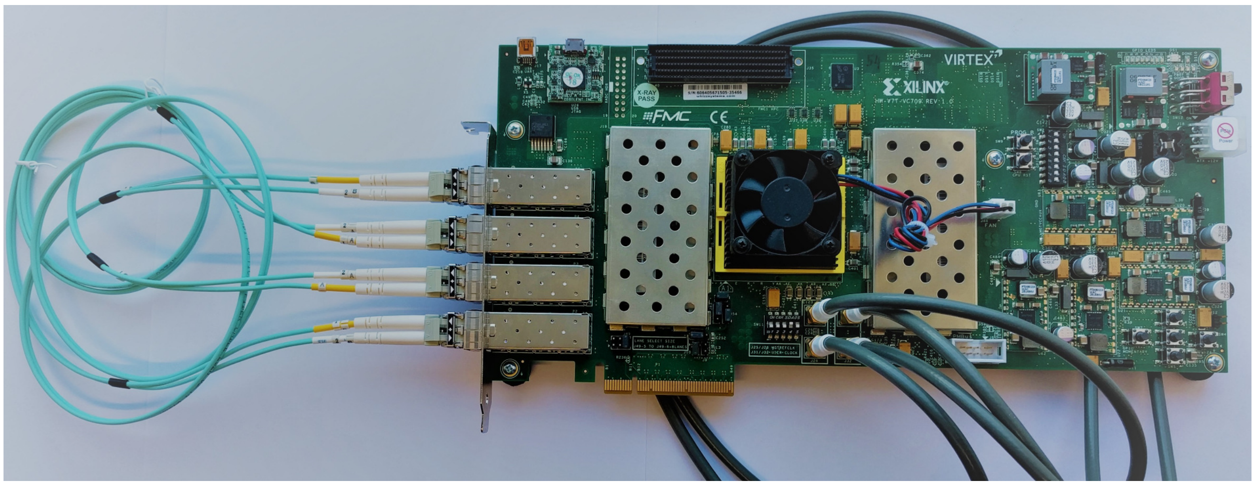


FPGA-Based Demonstrator for Real-Time Evaluation of a Fiber-Optic Communication System

Master of Science Thesis in Embedded Electronic System Design

FREDRIK ÅKERLUND

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
GOTHENBURG UNIVERSITY
Gothenburg, Sweden 2017

Thesis for the Degree of Master of Science

FPGA-Based Demonstrator for Real-Time Evaluation of a Fiber-Optic Communication System

Fredrik Åkerlund

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

FPGA-Based Demonstrator for Real-Time Evaluation of a Fiber-Optic Communication System.

Fredrik Åkerlund

Supervisor: Per Larsson-Edefors, Department of Computer Science and Engineering
Examiner: Lena Petersson, Department of Computer Science and Engineering

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31-772 1000

Cover: The VC709 development board.

Department of Computer Science and Engineering
Göteborg, Sweden 2017

Abstract

When the speed of serial transmission of data increases, it is important that the bit-error rate does not increase correspondingly. One way to maintain a low bit error rate is to use forward-error correcting codes for finding and correcting erroneous bits. This Master's Thesis describes the development of an FPGA system that acts as the physical layer in a fiber-optic communication system with bit-error correcting circuits using Bose–Chaudhuri–Hocquenghem codes. The FPGA transceiver system will allow for further research on, e.g., what level of error correction is suitable for physical coding sublayers.

Nine transceiver systems were developed in this thesis, with different error-finding and correcting Bose–Chaudhuri–Hocquenghem circuits. This report describes how the support logic needed was designed and how the hardware peripherals of the FPGA were enabled. The status of a running system is monitored in real-time from a program running on a PC, e.g., the current measured bit-error rate and the attenuation of the fiber-optic channel. The real-time program was used with the systems in a simple experiment to show how the error correction worked when the fiber-optic signal was attenuated.

Keywords: FPGA, ASIC, forward error correction, BCH, bit error rate.

Acknowledgements

Thanks to my supervisor Prof. Per Larsson-Edefors for suggesting the project, the writing assistance and organizing group meetings every other week. Thanks to Christoffer Fougstedt for being involved with both the BCH part and co-supervising, and my examiner Assoc. Prof. Lena Peterson for helping with writing once again. Thanks to Lars Lundberg, Tamas Lengyel and Magnus Karlsson from MC2 for the interest in helping us with their expertise about fiber optics.

Fredrik Åkerlund, Göteborg, June 2017

Acronyms

ASIC	Application Specific Integrated Circuit
BCH	Bose-Hocquenghem-Chaudhuri
BER	Bit-Error Rate
CLB	Configurable Logic Block
DFE	Decision-Feedback Equalizer
DFF	D-type Flip-Flop
DSP	Digital Signal Processing
FEC	Forward Error Correction
FPGA	Field Programmable Gate Array
FSM	Finite-State Machine
I/O	Input/Output
I2C	Inter-Integrated Circuit
I^2C	Inter-Integrated Circuit
IP	Intellectual Property
LFSR	Linear-Feedback Shift Register
LUT	Look-Up Table
MOSFET	Metal–Oxide–Semiconductor Field-Effect Transistor
OOK	On-Off Keying
PCIe	Peripheral Component Interconnect Express
PCS	Physical Coding Sublayer
PLL	Phase-Locked Loop
PMA	Physical Medium Attachment
QPLL	Quad Phase-Locked Loop
ROM	Read-Only Memory
RTL	Register Transfer Level
RX	Receive
SATA	Serial Advanced Technology Attachment
SFP	Small Form-Factor Pluggable
SMA	Sub Miniature version A
SRAM	Static Random Access Memory
TCL	Tool Command Language
TX	Transmission
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
VCO	Voltage-Controlled Oscillator
VCSEL	Vertical-Cavity Surface-Emitting Laser
VHDL	VHSIC Hardware Description Language

VHSIC	Very High Speed Integrated Circuit

Contents

List of Figures

1
Introduction

The optical fiber is the preferred type of medium between physical layers in the backbone of today's Internet and is continuously expanded to end users by Internet suppliers. As the speed of serial communication links increases, so do the errors due to factors that were not necessary to consider at lower speeds, e.g., noise, signal attenuation or jitter. To keep the transmitted data intact at the receiving end, various components for ensuring the signal integrity are used, e.g., differential signaling and encoding schemes, emphasis of signals, different types of filters to compensate for any signal degradation [1, 2, 3]. Bit-errors can still occur even if all the components mentioned above are used. If forward error correction (FEC) is used in the physical coding sublayer (PCS) the bit-error rate (BER) can be reduced further. FEC works by adding parity bits to both locate incorrect received bits and correct them [4].

Some Xilinx field programmable gate array (FPGA) development kits, e.g., the VC709 [5] that is available at Chalmers, have small form-factor pluggable (SFP) connectors for fiber optical (or Ethernet) transceivers so they can be used for prototyping the physical layer in communications systems. FPGAs like the 7-series mounted on the VC709 board can thus be connected to data streams from and to optical transceivers and be configured with custom systems, e.g., systems with FEC codes or digital signal processing (DSP) components.

As of today, there have been no open literature publications (to the best of my knowledge) about a system that demonstrates how a fiber optical communication system can be attached to an FPGA which implements various configurations of FECs, or in other words, how the PCS with FEC can be implemented in an FPGA. In this thesis project, we aim to make use of the four SFP connectors on a VC709 board which can provide a total transfer rate of 40 Gbit/s, where each of them is connected to a transceiver system implemented in a hardware description language (HDL). Since FPGAs can be reconfigured, any subsystem that is implemented in it can be modified, e.g., the FEC component in the transceiver system. Thus, the final system will allow for different FEC configurations to be evaluated in an FPGA by, e.g., attenuating the signal in the fiber optical channel.

1.1 Aim

The overarching goal of this thesis is to develop and evaluate an HDL implementation of an FPGA system, both logic and peripherals, that acts as physical layer in a fiber-optical communication system. The FPGA system will allow for further research investigations on, e.g., how FEC can be added to the physical layer to improve the resilience of the communication system. The used VHDL code for the FEC circuits are generated by a MATLAB script provided by the assistant supervisor Christoffer Fougstedt.

With an implemented system, it should be possible to connect the FPGA to a vertical cavity surface emitting laser (VCSEL) that has been manufactured at MC2, Department of Microtechnology and Nanoscience at Chalmers. The equipment available at MC2 should be able to attenuate the signal in order to yield the BER to SNR ratio, i.e., the relation between the bit errors and the signal-to-noise ratio. As this required knowledge is outside of our department, this is a long-term aim.

In addition to functional properties demonstrated in the FPGA experiments, a digital ASIC design of the FEC logic portion of the system will also provide accurate power and performance numbers, although not from a manufactured die but from cell netlists.

1.2 Approach

The VC709 [5] board has four SFP modules connected to embedded high-speed circuitry, i.e., SerDes (Serializer/Deserializer), embedded in the FPGA at the very edges of the die. There are ten more SerDes ports with traces routed to a breakout board called an FPGA Mezzanine Card (FMC). Initially we were lacking information on the parallel interfaces and therefore the first milestone aims for enabling the four SerDes transceivers that are connected to the SFPs, e.g., by developing a loop-back over a fiber channel (FC). Xilinx HDL development platform Vivado provides IP cores and example designs for the 7-series transceivers which other projects can customize and be built upon and therefore is the best start.

When we have a functional loop-back system we will add FEC components to the system which will require more circuitry as the input and output word-widths of the FECs will vary. Additional logic for calculating the BER as well as sending the data is needed, too.

To monitor signals in real-time, e.g., the power of the received signal and the BER over some duration of time, either a script for a virtual terminal or a GUI on a host PC will be programmed to receive and present data from a UART component in the FPGA.

All data generated are synthetic so we do not have to consider any aspects pertaining to privacy concerns in this project.

1.2.1 Tools

Vivado [6] is the provided software for HDL development for Xilinx 7-series FPGAs. Vivado synthesizes the HDL, and generates bit-files that are used for the configuration of FPGAs and can also do simulations. Questasim is another software for HDL development which we also use for simulations. The test-benches need reference data to verify the correct functionality of components and therefore writing scripts is the best way to generate such data and this is preferably done with Python. The VHDL code for FEC circuits can be generated with a provided MATLAB script.

1.2.2 Thesis Outline

A reader of this thesis is assumed to have most of the knowledge needed to understand the report, but the Technical Background (Chapter 2) will very briefly explain some of the technologies used, which can connect to the subsequent chapter about the Xilinx VC709 board (Chapter 3). Hopefully these chapters will help readers with no previous experience in this field understand the content better.

The chapter about how the resulting system's VHDL components are designed (Chapter 4) is explaining in detail how a system works, and is intended as the project's documentation for anyone who might do further development, too.

The Results chapter (Chapter 5) shows the output from Vivado after synthesis and implementation, and the results from the experiment which is followed by a discussion (Chapter 6). The thesis then ends with a conclusion (Chapter 7) about the thesis' results.

2
Technical Background

This chapter explains the theory and technologies used in this report. A brief section about FPGAs is included for the unfamiliar reader. One way to describe FPGAs is as “a type of circuit which can implement the logic of almost any other circuit”, which is not entirely correct but a good first explanation. The first section shows the fundamental components of an FPGA which can be configured to implement different digital circuits. In this thesis, an FPGA is used as a physical layer in a network system, a layer which is put into its context in the subsequent section.

Differential signaling and phase-locked loops are also covered, as both are frequently mentioned in relevant data sheets [5, 7, 1, 8] and are used with SerDes circuits [2], the high-speed electronics which is a fundamental component of fiber-optic transmissions. The chapter will end with some theory about encoding data for the purpose of enabling a high-speed connection, and using BCH for finding and correcting erroneous received bits.

2.1 Field-Programmable Gate Array (FPGA)

FPGAs consist of configurable logic blocks (CLBs) that are connected through an array of programmable interconnections (see Fig. 2.3). The logic described in HDL such as VHDL or Verilog can later be synthesized with software that also implements the design and generates a bit-file which describes how the CLBs and interconnections are to be configured (to implement the desired logical functions). For example, whole microprocessors can be described in HDL and implemented in FPGAs, which in turn then can run traditional software written in, e.g., C/C++ or assembler language on a PicoBlaze softcore [9].

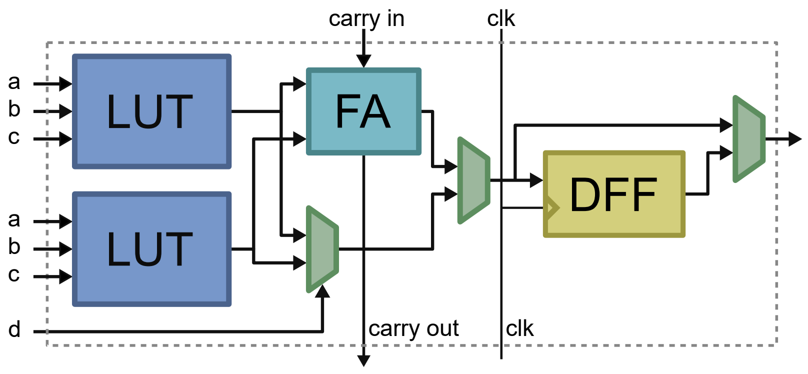


Figure 2.1: A configurable logic block used in FPGAs.

The interconnections and look-up tables (LUTs) are configured by 1-bit memory cells, mostly SRAM based, which make certain connections to CLBs and make the LUTs hold certain values. The look-up tables can implement all combinatorial logical functions, i.e., OR, AND, XOR, NOT and their complements as well as any custom behavior. Fig. 2.1 shows an example of a CLB. CLBs can contain full adders (FAs) for accelerating addition and subtraction and some FPGAs contain special DSP blocks for the more complex mathematical operations, e.g., multiplication and division. There are also MUXes for routing the signals and a d-type flip flop (DFF) for saving one bit.

In Fig. 2.2 a conceptional gate array is illustrated, and shows that there can also be blocks for clock handling through the array as well as a couple of phase-locked loops (PLLs) for frequency synthesis and configurable input and output blocks (IOBs). Some blocks are dedicated for RAM which can be useful for complex functionality, e.g., DSP systems.

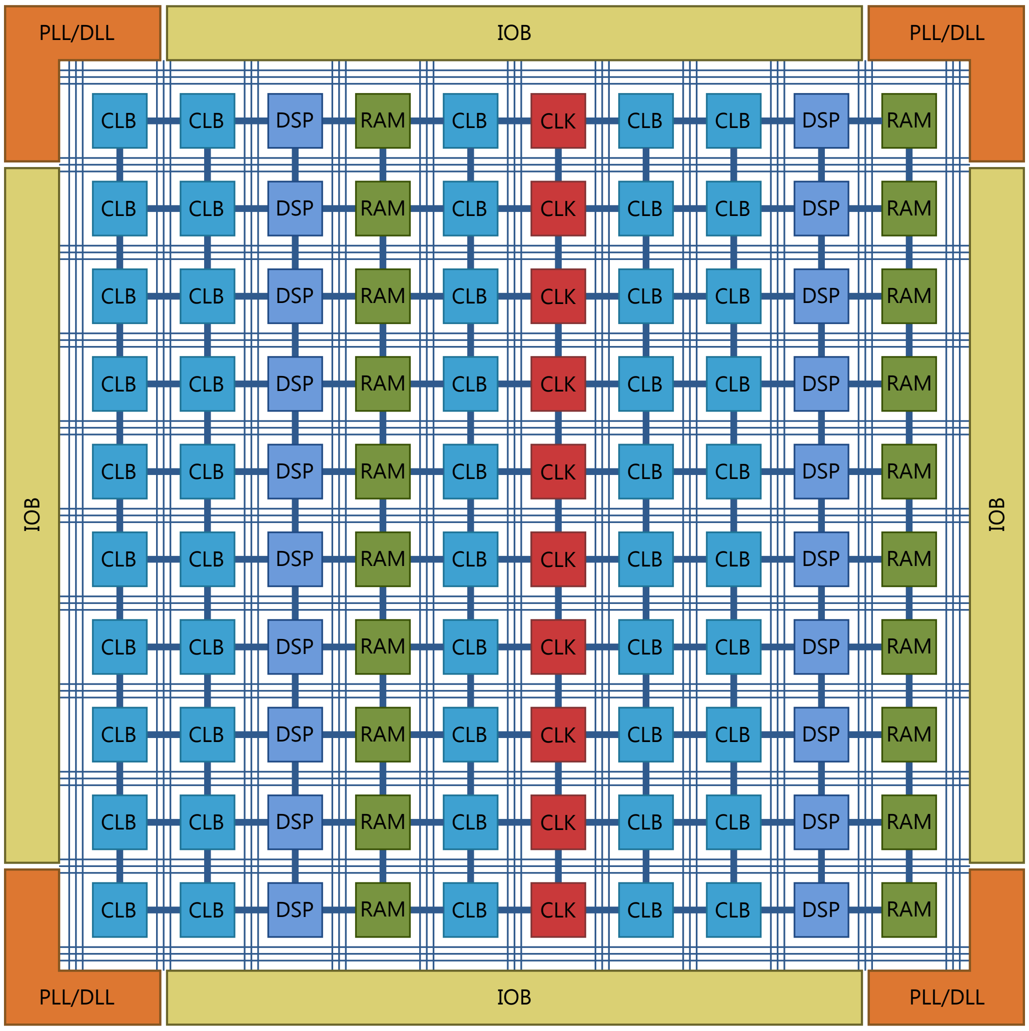


Figure 2.2: Illustration of a conceptional gate array.

The interconnection between blocks can be done as illustrated in Fig. 2.3 where several transistors can either short connections together or keep them open. The gates of the transistors in Fig. 2.3 are connected to squares that are representing SRAM blocks which configures the routeing.

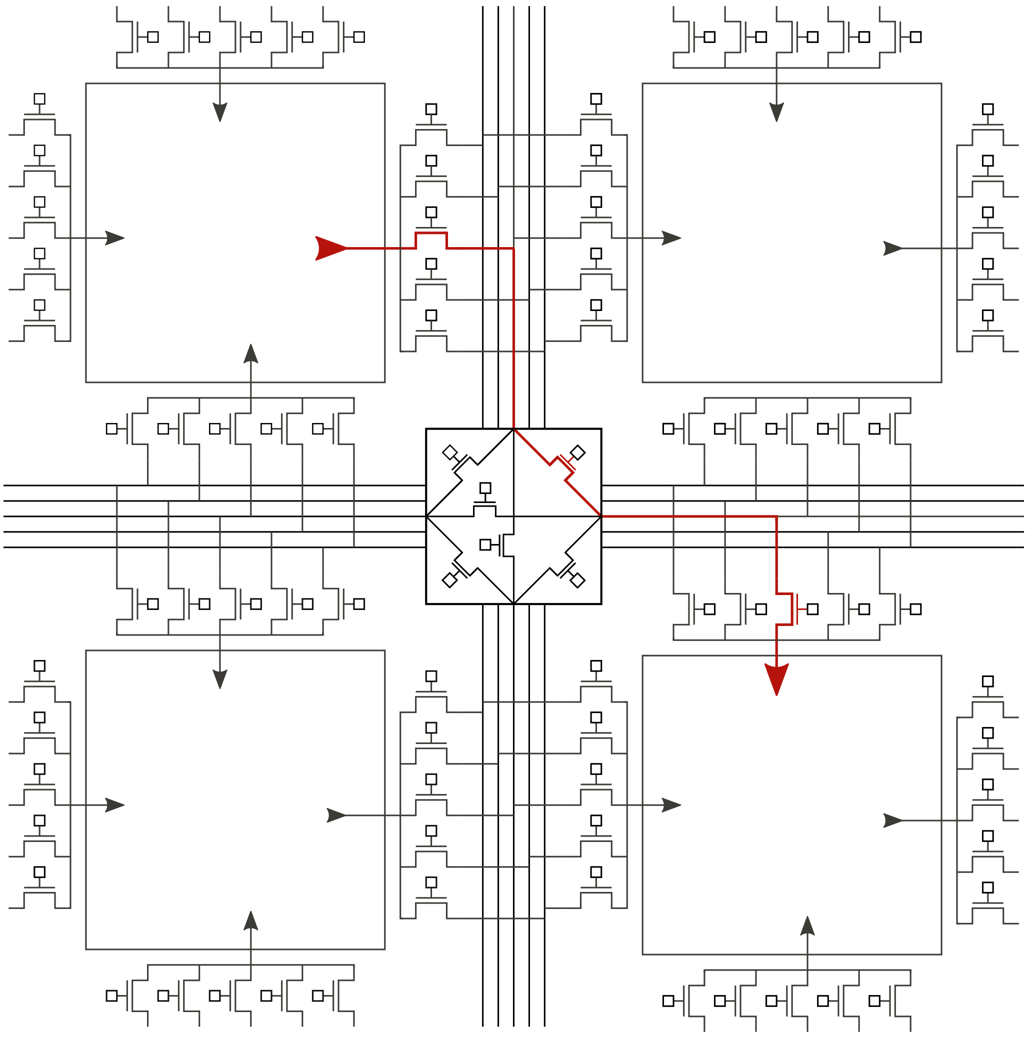


Figure 2.3: An interconnection matrix with CLBs.

Because of the enormous amount of wires, i.e., interconnections, FPGAs often work at low frequencies because of the large capacitance. The most power consuming part of an FPGA are all the interconnections. Table 2.1 shows that the power consumption is dominated by the interconnections [10] in a Xilinx XC4003A FPGA.

Table 2.1: Power breakdown for a XC4003A FPGA from Xilinx [10].

	CLB Power	IO Power	Clock Power	Interconnect Power
Percentage	5	9	21	65

2.1.1 Look-Up Tables (LUTs)

An LUT essentially has the same functionality as a ROM. A four-input, one output LUT, can generate any four-input Boolean function (AND/OR/XOR/NOT). Assume the logic expression

$$(I0 \wedge I1) \vee (I2 \wedge I3)$$

is wanted in a FPGA. A LUT can then configured to the truth table of the expression as in Table 2.2. For implementing the logical expression, the address input of the ROM are instead used as the equivalent to a logic gate's input ports, and the resulting logic value for the expression is saved at the corresponding address.

Either I0 **and** I1 needs to be **true** ("1" and "1") **or**, I2 **and** I3 for the expression to evaluate to **true**, i.e., a high output or logically one. Thus, this LUT is just as a ROM with sixteen 1-bit registers.

Table 2.2: A truth table of a LUT with four inputs.

I3	I2	I1	I0	Output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Tools like Xilinx Vivado takes care of all configurations of LUTs when running the implementation which translates the HDL into the physical design, i.e., the bit-file which contains values for configuring a LUT's SRAM transistors.

2.2 Physical Layer

The open systems interconnection (OSI) reference model shown in Fig. 2.4, was developed by the International Standards Organization (ISO) and defines seven layers in networks communications systems. Fig. 2.4 puts the Physical layer in context, it is the lowest in the hierarchy and is concerned with the physical transmission of data [11] over various transmission mediums such as cabling, fiber optics or wireless. The Application layer is presented to users and the layers in between takes care of, e.g., routing data packages over the Internet. Examples of protocols for each layer are shown above the arrows in Fig. 2.4 which connects two nodes in a network.

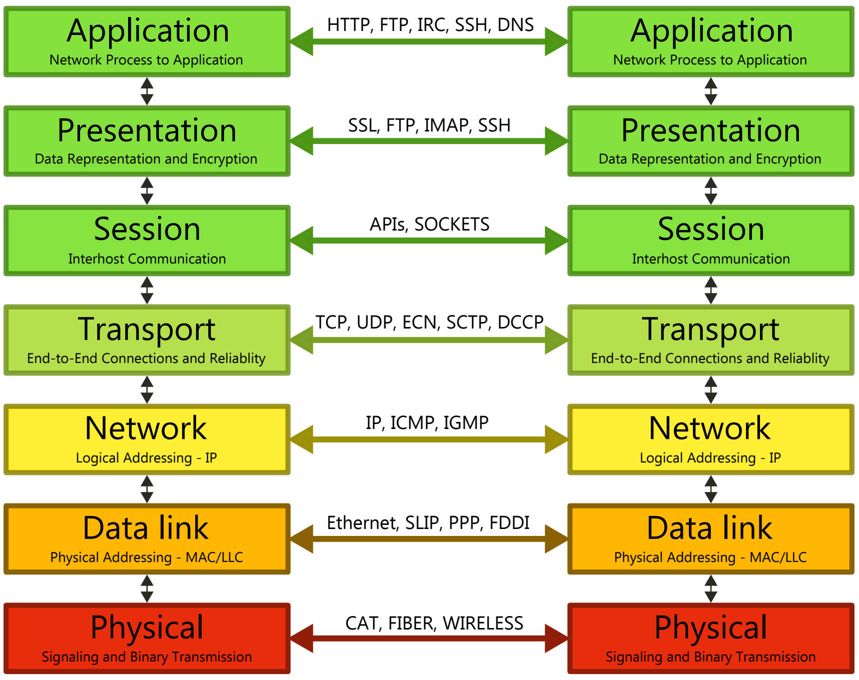

Application
Network Process to Application
HTTP, FTP, IRC, SSH, DNS
Application
Network Process to Application
Presentation
Data Representation and Encryption
SSL, FTP, IMAP, SSH
Presentation
Data Representation and Encryption
Session
Interhost Communication
APIs, SOCKETS
Session
Interhost Communication
Transport
End-to-End Connections and Reliablity
TCP, UDP, ECN, SCTP, DCCP
Transport
End-to-End Connections and Reliablity
Network
Logical Addressing - IP
IP, ICMP, IGMP
Network
Logical Addressing - IP
Data link
Physical Addressing - MAC/LLC
Ethernet, SLIP, PPP, FDDI
Data link
Physical Addressing - MAC/LLC
Physical
Signaling and Binary Transmission
CAT, FIBER, WIRELESS
Physical
Signaling and Binary Transmission

Figure 2.4: The OSI reference model.

The physical layer includes mechanical, electrical and timing interfaces in its design to make the transferring of bits as accurate as possible. The physical layer is complex and can be described by other sub-layers as shown in Fig. 2.5. The Xilinx VC709 board (Chapter 3) can implement the PCS and PMA sublayer.

2.2.1 Physical Coding Sublayer (PCS)

The physical coding sublayer lies below the data link layer (layer two) in the OSI reference model and above the physical medium attachment (PMA) physical sublayer [12]. The PCS performs several functions such as the encoding/decoding (Section 2.6) and scrambling/descrambling of data blocks (Section 2.6.2). The encoding of data is performed together with scramblers to ensure enough transitions between logical ones and zeroes so that the receiver can perform clock recovery (Section 2.5) from a serial link which provides no clock reference. The PCS can insert and remove markers for alignment of data and identify blocks of received data streams (Section 2.6.1) for encoding schemes such as 8b10b. The PCS can also have forward

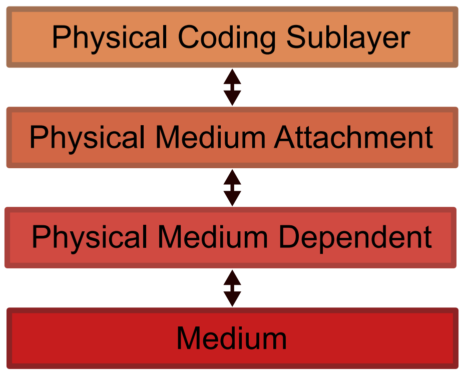


Figure 2.5: Sublayers of the physical layer [12].

error correction codes (FECs) which is an aim of this thesis, using BCH circuits (Section 2.7).

2.2.2 Physical Medium Attachment Sublayer (PMA)

The physical medium attachment sublayer connects the PCS to an electrical interface such as an optical transceiver (Section 3.6). Electrical transceivers for fiber optical channels can contain laser drivers for VSCEL lasers. Receivers are essentially photo-detectors connected to circuits for amplification and quantization, often sharing any controlling circuit with the transmitter. For high-speed links, the PMA serializes and de-serializes data and performs clock recovery from received data streams. The PMA is in other words responsible for taking inputs of parallel data and sending it out at high speed (and vice versa), and the data conversions can be implemented in a SerDes circuit which is further described in Section 2.5.

2.3 Differential Signaling

Long wires or traces on a printed circuit board (PCB) for high-speed digital signals can be problematic as, e.g., the capacitive load increases and affects the speed, interference from other signals can be picked up and signals are reflected back due to impedance mismatching [3]. Serial transmission of data allows for higher speeds than parallel as these mentioned factors get isolated between only two points, which makes impedance matching easier, any time skew between received (parallel) signals are removed and noise get canceled out better. Today this signaling is used in many common technologies, e.g., PCIe, SATA and USB.

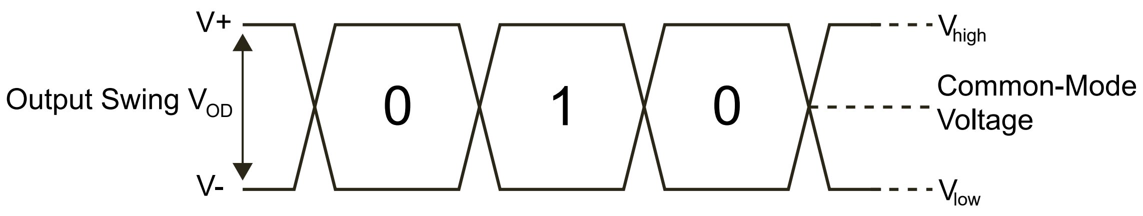


Figure 2.6: Differential signaling.

Differential signaling uses two signals where one signal is the complement of the other, as shown in Fig. 2.6. The signals drive the receiving logic, i.e., a differential amplifier, which determines logic values as the difference between the signals. TIA/EIA-644 or Low-Voltage Differential Signaling (LVDS) is a standard capable of data rates up to 3 Gb/s [13] and is centered at 1.25 V with an output swing of 0.3 V [3]. For even faster circuits Current-Mode Logic (CML) can be used.

2.4 Phase-Locked Loop (PLL)

A phased locked loop (see Fig. 2.7) can generate an output signal (f_{out}), e.g., a clock signal ("clock recovery"), that has a frequency which is close to the incoming (f_{ref}) signal's frequency. PLLs can also synthesize other frequencies, i.e., generating multiples or divisions of the input signal's frequency. A PLL used as a frequency multiplier is one of the most common applications [3].

The phase detector (which can be digital) drive transistors with a voltage generated by the difference between the edges of the input signals. A filter removes any unwanted frequencies of the voltage signal before it is passed to the voltage-controlled oscillator (VCO). The VCO acts as an integrator and will increase its output frequency if the phase lags and decrease it as the lag decreases.

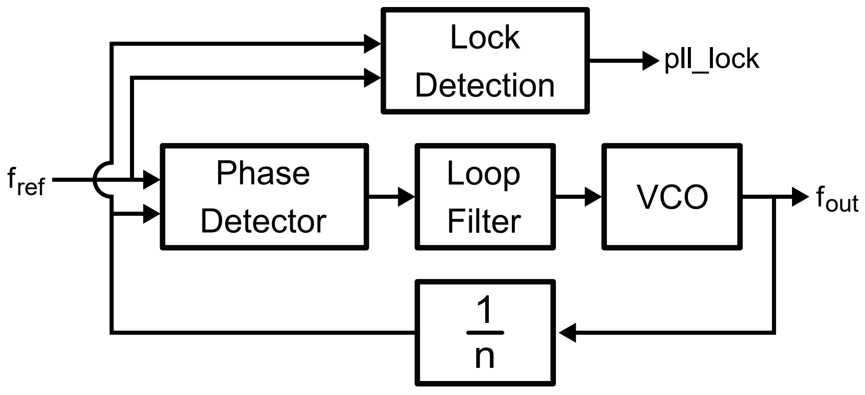


Figure 2.7: Phase-locked loop.

A signal indicating that the PLL has locked onto the incoming signal's frequency is needed for some applications, e.g., systems which recovers the reference signal's frequency (which might be subject to noise) and use it as a clock source and, thus, must be certain the PLL is outputting a stable signal.

2.5 SerDes

SerDes (Serializer/Deserializer) circuits are used for parallel-to-serial and serial-to-parallel conversion. In Fig. 2.8 a generic block diagram of a SerDes transceiver circuit is shown. Both the TX and RX electrical interfaces [3] are amplifiers for differential signals (Section 2.3). The serializing and de-serializing circuits are often referred to as "parallel in serial out" (PISO) and "serial in parallel out" (SIPO) [1] respectively, and are using the technique of having different parts of the circuit clocked by different phases (of one clock) to implement their shift-register functionality at a high speed.

To achieve high speeds, the clock signals come from PLLs (Section 2.4) that have multiplied a reference clock (up to several gigahertz), illustrated by the "Clock Manager" and "Oscillator" (OSC) in Fig. 2.8. Because a clock signal is so fast, the requirements on its frequency precision is measured in parts per million (PPM), as a small deviation will lead to several Hertz in difference.

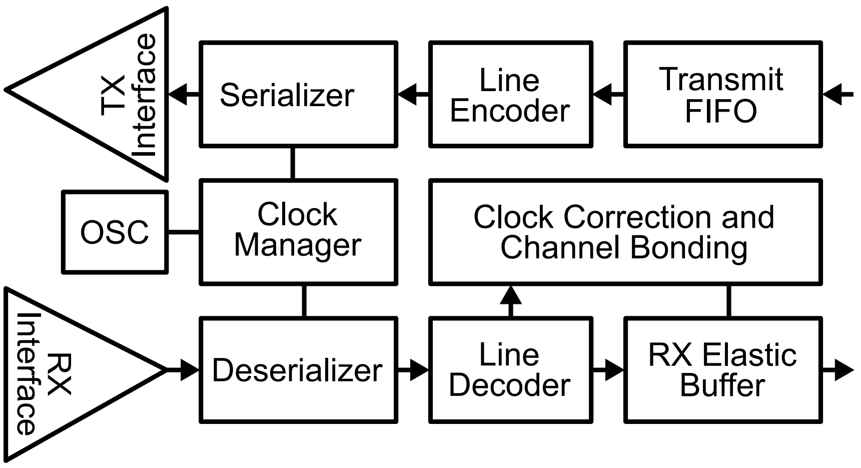


Figure 2.8: Generic block diagram of a SerDes transceiver.

High-speed SerDes use several phases (of one clock) to clock different parts of the circuitry to achieve a Gbit/s transmission speed. For example, imagine a 1 GHz clock signal split into four phases which drives four different flip-flops connected to the same output port. Since all four flip-flops connects to the output, but use it at different time, the effective bit-rate will be 4 Gbit/s. The opposite can be done, too, for de-serializing data and have flip-flops (clocked by different phases of one clock) to read bits, e.g., a 10 Gbit/s data stream can then be turned into 64-bit words outputted with a frequency of 156.25 MHz, as GTH components do (Section 3.2).

Because there is no reference clock provided to a receiver for synchronizing data correctly, the clock period must be determined in another way. Clock correction is one technology which can be implemented by using unique symbols in the data-stream that is found nowhere else like with the 8b/10b encoding scheme (Section 2.6.1). Often the idle streams are just clock correction sequences.

A PLL (Section 2.4) can be used to recover a clock signal which matches the highest frequency of the incoming stream of serial data. In some high-speed systems clock recovery is not needed, e.g., on chip-to-chip systems, as the components are close enough to each other to use the same oscillator source.

The PMA (Section 2.2.2) circuitry, i.e., the TX and RX interfaces attached to the SerDes can also include pre and de-emphasis logic to maintain signal integrity, in some cases also combined with an equalizer on the receiving side, either active or passive [2]. Encoding schemes such as 8b/10b or 64b/66b (Section 2.6.2) or others are being used too, for keeping a good DC balance together with scrambling of the data, and is sometimes hardware implemented as Fig. 2.8 illustrates (Encoder and Decoder). The PLLs recovered frequency might not always match the incoming data stream's which is compensated for in a clock correcting component which also can bond several channels together and correct any skew between them [2].

2.6 Encoding Schemes

Encoding schemes are used for achieving a DC-balance on a transmission line by using symbols with equal amounts of zeros and ones, sending headers or a Scrambler which rearranges data in a certain pattern. A Descrambler can in turn arrange the data back to the way it was, even if it would appear to be random.

2.6.1 8b/10b

The 8b/10b encoding scheme was developed by IBM [2]. With a look-up table, every byte is translated to a 10-bit word or symbol. The symbols have either four ones and six zeroes (or vice versa) to maintain a good DC balance on the line and ease the clock recovery of a PLL. Some of the 10-bit words are control words which are used, e.g., as commas for alignment of data or to perform clock recovery. By adding two bits the overhead becomes 25 % which means that for a 8 Gbit/s (efficient bit-rate) serial link a 10 Gbit/s line is needed.

2.6.2 64b/66b

High-speed applications use encoding schemes such as 64b/66b for the less overhead (compared to 8b/10b) which were developed for 10 Gb/s Ethernet [2]. The two bits of overhead are used for control and are always sent first and only two pairs counts as valid headers. They are used for synchronization and specifying what type of words that follows them, either data ("01") follows or control ("10") words follows.

Synchronization can be done by observing either "01" or "10" in the data stream several times in a row and always 66 bits away from each other. When enough valid headers have been observed, a system can decide that it has a valid synchronization. If an invalid header is observed, the received data should be shifted one bit at the time until it becomes possible to observe valid headers continuously, 66 bits apart.

Thus, 64b/66b does not use any look-up values (like 8b/10b) so, to guarantee enough bit-transitions for DC-balance the systems have a Scrambler and Descrambler integrated instead. Scrambling works with a polynomial on the data blocks which reorders data (to appear random). The headers are not subject to scrambling. Therefore, a transition is guaranteed every 66 bits, but the probability to receive only zeros or ones in a scrambled word is extremely low.

Scrambling the data for 64b/66b encoding is done using the polynomial

$$X_{58} + X_{39} + 1.$$

The scrambling of bits is done with a linear-feedback shift register (LFSR). So, the polynomial tells us that the 39th and 58th output of the register generates the feedback (by an XOR operation) into the first bit of the word.

2.7 Forward Error Correction

Forward error correction (FEC) is a technique for a transmitter to encode data in way so that a receiver can detect and correct any erroneous bits. For example, Bose-Hocquenghem-Chaudhuri (BCH) uses an encoder which adds more bits in a way so that the decoder can use them to find the erroneous bits. BCH were developed two times, first in 1959 and then in 1960 by first Hocquenghem followed by Bose and Chaudhuri. A BCH(n,k,t) code describes n as the length of a message, k as the length of the codeword and t as the number of bits which can be found erroneous and corrected.

Let d denote the number of positions that two code-words belonging to one code differ, then the following must hold for a valid BCH code.

$$n = 2^m - 1$$

$$n - k \leq mt$$

$$d_{min} \geq 2t + 1$$

From a hardware design perspective all the math [4] is not as interesting as what valid combinations of (n,k,t) exists. A few example of combinations for (n,k,t) are shown in Table 2.3 along with the overhead.

Table 2.3: FEC overhead for different valid BCH codes.

n	k	t	Overhead
63	57	1	9.52%
127	120	1	5.51%
255	247	1	3.13%
511	502	1	1.76%
1023	1013	1	0.0098%
63	51	2	19.05%
127	113	2	11.02%
255	239	2	6.27%
511	493	2	3.52%
1023	1003	2	0.0196%
63	45	3	28.57%
127	106	3	16.54%
255	231	3	9.41%
511	484	3	5.28%
1023	993	3	0.039%

The smaller messages gets a large overhead, and the efficient bit-rate (bit/s) will be lower than for the larger messages. To correct more bits, more parity bits are needed which will further increase the overhead, too.

3
The Xilinx VC709 Development Board

This chapter is about the used development board which photo is shown in Fig. 3.1, unpacked from the Xilinx Connectivity Kit box [14]. The kit also includes a pair of low-smoke zero-halogen (LSZH) cables which in the photo are plugged into the four included optical 10.3125 Gbit/s transceivers. The transceivers are in turn plugged into the SFP connectors on the left. The lower grey pair of SMA cables connects the 200 MHz system reference clock to the FPGAs 10.3125 Gbit/s GTH transceivers for reference. On the backside of the SMA connectors is an IC with a programmable PLL called the Si5324 from Silicon Labs that can be used as an alternative to the SMA input. To program the Si5324 by I^{2}C the FPGA ports for SCL and SDA must be routed through an I^{2}C MUX which is also programmed by I^{2}C. The MUX has one of its pairs of I/Os connected yet another I^{2}C MUX which can be programmed to connect to either of the four SFPs. Naturally, the 7-series Virtex XC7VX690T is under the cooling fan, and has two RAM DIMMs closely placed next to it. The silkscreen above the leftmost DIMM says FMC which is the acronym for FPGA Mezzanine Card, which refers to the connector above with a high pin count (HPC). With an FMC HPC connector, it is possible to attach ten more 10.3125 Gbit/s transceivers. On the top left corner there are two USB connectors connected to an USB-Serial bridge IC and a JTAG port for programming.

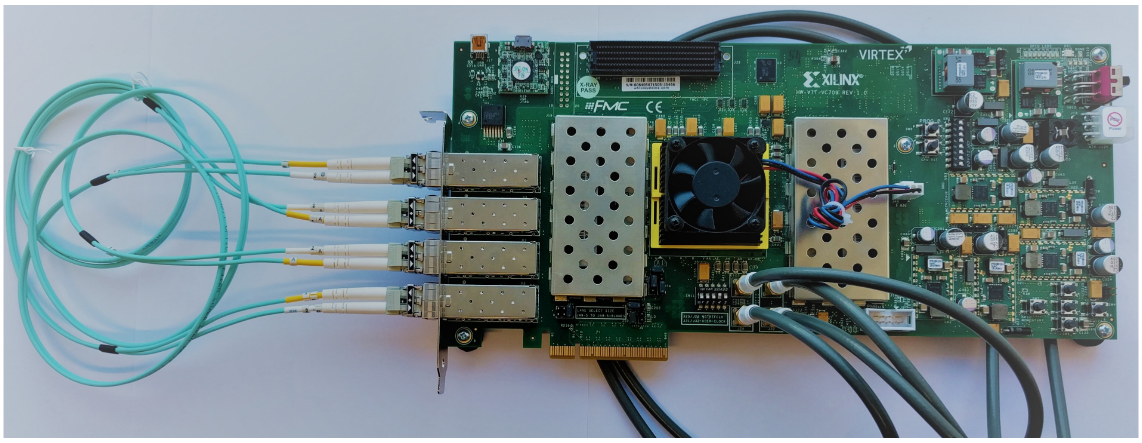


Figure 3.1: Xilinx VC709 board.

The rest of the chapter now focuses on some of the mentioned components in the above text. There will also be a brief text about Xilinx's example design that can be generated after an IP core has been configured with the wizard in Vivado.

3.1 The Virtex XC7VX690T-2FFG1761C

Mounted onto the VC709 is a XC7VX690T FPGA. From the data sheet there are some specifications of interest as follows [7].

- 42.5 mm x 42.5 mm in size.
- Contains 108300 Slices, where each contains four LUTs and eight flip-flops. Some of the slices can use their LUTs as distributed RAM or SRLs (Shift Logical Left).

  10 888 Kb as max distributed RAM. Each block RAM is 36 Kb.
- Contains 3600 DSP slices, where each contains one pre-adder, one 25 x 18 multiplier, one adder and one accumulator.
- Clock management tiles (CMT), 20 components where each one mixed mode clode manager (MMCM) and one PLL.
- Gigabit transceivers (GT), 80 GTHs each capable of 10.315 Gbit/s.
- RAM Blocks, maximum 52 920 Kb.
- 850 I/O, supports voltages from 1.2 V to 1.8 V.

3.2 GTH Transceivers

The various 7-series FPGA models have transceivers of different speed classes. The C7VX690T have 80 gigabit transceivers (GT) of class H, hence the acronym GTH, where the H means a speed-class of 10.325 Gbit/s. Ten of the GTH transceivers are wired to the FMC HPC connector and four are wired to the four SFP/SFP+ connectors. The ASIC part of the FPGA that constitutes a GTH transceiver are referred to as a GTHE2_CHANNEL by Xilinx and is placed on different banks, i.e., segments, of the FPGA die but always at the edge. Apparently it is only the transceivers of the SFP modules that can utilize the external PLL from the Silicon Labs, the Si5324 jitter attenuator [5]. The SFP modules can also use another external reference that can be provided with the inputs from the on board SMA connectors. The ten transceivers connected to the FMC header have to share the same clock reference.

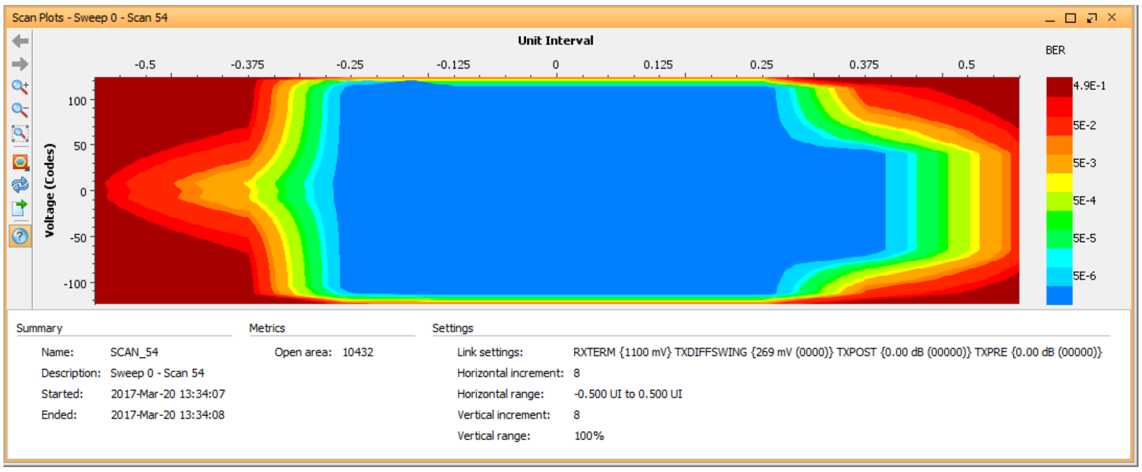
3.3 Xilinx IP IBERT Design

Xilinx provides an integrated bit-error rate test (IBERT) as an IP-core which can be generated and used through Vivado. The dynamic reconfiguration ports (DRP) allows a designer to access the configuration and status registers of, e.g., a transceiver's GTHE2_CHANNEL or clock signal components as a PLL or a mixed-mode clock manager. The IBERT design uses the DRPs to change different settings while running a test. With dynamic settings in tests the hardware can be confirmed working without problems. Different settings are changed with the DRP ports, for example,

- Bit-counter and Error-counter for BER.
- TX/RX PRBS Pattern - (7, 15, 23, 31)-bit.
- TX Pre-Cursor - 0.0 dB to 6.02 dB.
- TX Pre-Cursor - 0.0 dB to 12.96 dB.

- TX Diff Swing - 269 mV to 1119 mV.
- DFE Enable/Disable.
- Error Injection.
- TX/RX Reset.
- TX/RX PLL Status.
- Loopback Modes: None, Near-End PCS, Near-End PMA, Far-End PCS, Far-End PMA.

Vivado can also establish a serial link to the FPGA when it is running the IBERT and provide plots with data from the receiver. Fig. 3.2 shows an example of a resulting plot from one iteration during a sweep of different parameters.



Summary

```
Name:         SCAN_54
Description:  Sweep 0 - Scan 54
Started:      2017-Mar-20 13:34:07
Ended:        2017-Mar-20 13:34:08
```

Metrics

```
Open area:   10432
```

Settings

```
Link settings:          RXTERM {1100 mV} TXDIFFSWING {269 mV (0000)} TXPOST {0.00 dB (00000)} TXPRE {0.00 dB (00000)}
Horizontal increment:   8
Horizontal range:       -0.500 UI to 0.500 UI
Vertical increment:     8
Vertical range:         100%
```

Figure 3.2: Resulting IBERT eye scan plot in Vivado.

After the IP component has been generated in Vivado, a pair of SMA cables must be bridged as showed in Fig. 3.3 where the 200 MHz clock signal from the FPGA is routed right back to the SMA input. Then it is only to upload the bitfile and use the the interface in Vivado for testing.

3.4 Silicon Labs Si5324

The Si5324 jitter attenuator contains a PLL which has three inputs for reference clock signals. One reference clock input is from a closely placed crystal with a frequency of 114.285 MHz. The other two differential pairs are inputs either from the FPGA or one SMA pair. The registers of the Si5324 can be read and written to with an I^{2}C interface, i.e., using an HDL implementation. The register values needed for a specific output frequency can be generated with help of the software called DSPLLsim from Silicon Labs. The GTH transceivers (connected to the SFP/SFP+ modules) are using the reference clock provided from the Si5324 to drive their reference clock inputs in order to perform clock recovery from the serial data input stream.

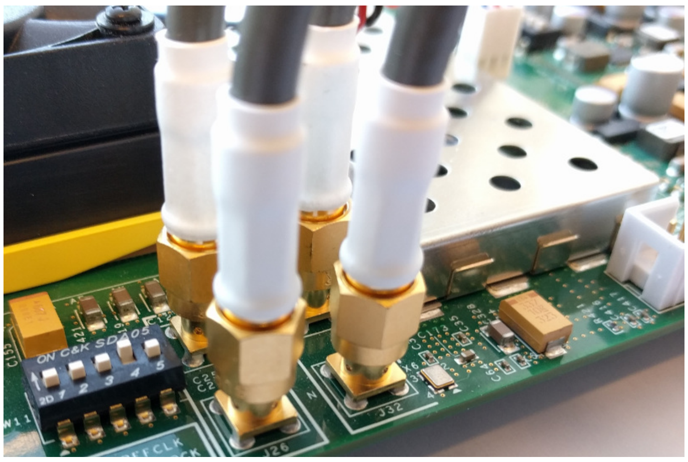


Figure 3.3: The SMA connectors of the VC709 board.

3.5 Xilinx IP Transceiver Example

Xilinx provides an example design that can be customized and used as a start to build own designs upon. After the IP component has been generated, the example design can be generated from it which provides wrappers as seen in Fig. 3.4.

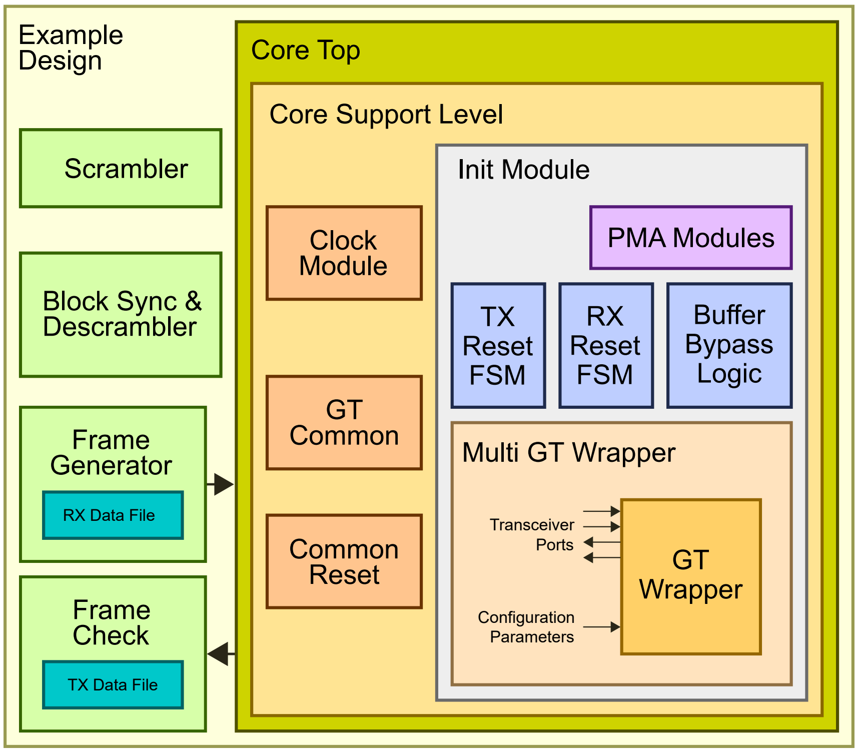


Figure 3.4: Structure of the transceiver wrapper and example design.

- For 64b/66b encoding scheme, the scrambler and descrambler are provided. The block sync will assure the received data are aligned correctly.
- The frame generator generates a data stream for transmission that is definable by a user, e.g., by changing the input file's content.

- The frame checker examines the data stream transmission and accumulates found errors and asserts the sum to an output. The data file is an exact copy of the one that the frame generator is using.
- The clock module generates clocking signals, i.e., it takes a reference input clock, e.g., from the Si5324 and passes it to the special IBUFDS for the GTE2 channel.
- The GT common module instantiates the GTHE2_COMMON primitive which is shared to multiple transceiver cores. The quad PLL (QPLL) used for all for channels has its settings defined here, e.g., divider, selection of reference clock and QPLL lock detection.
- The common reset module provides circuitry for resets that is used for a core reset. It signals the FSMs and can be provided a user reset, e.g., from a push button.
- FSMs modules for resetting the transmitter and receiver. It takes care of the sequential reset of hardware as described in [1]. The reset includes verification of a stable clock from the PLL and MMCM, i.e., phase alignment, PLL lock and synchronization for enabling a stable clock.
- PMA Modules: Not used in this project.
- GT Wrapper contains the actual IP core for the transceiver hardware. It holds the defined settings for cores and enables access to the DRP ports.

The Frame Generator Iterates over TX data. The TX data has 16 different words to send, each read from a file with its respective encoding header for 64b/66b hard coded into the same line. Therefore it also signals the GTH.

The External Gearbox Sequence Counter A gearbox is a component that helps with encoding of data, e.g., 64b/66b. There is a counter implemented in FPGA logic which pauses both the frame generator and scrambler logic. This counter is connected to its respective GTH2_CHANNEL ports to make the Gearbox pause as according to the transceiver data sheet [1]. Because every sent data block of 64 bits has 2 bits overhead after it has passed through the Gearbox, every 32th clock period will have delivered 64 bits overhead to the transmitter or serializer. Since 64 bits is a whole word the system must stop the input of more data for one clock period to compensate for the extra bits sent.

The Scrambler and Descrambler The scrambler and descrambler in Fig. 3.4 are implemented as linear-feedback shift registers and must be synchronized with each other for them to work. This means that if any data input is unintentionally wrong on the RX side and discarded, the two components will not be synchronized anymore and both have to be reset.

The RX Block Synchronizer The RX block sync in Fig. 3.4 will hold a signal asserted for some time until enough headers have been counted, e.g., 64 as in one case, to make sure that the blocks are arriving in a correct order. In the case when misaligned blocks are arriving to RX, the input buffer will be shifted by one bit each time an incorrect header has been found, i.e., “00” or “11”. When enough headers have been observed in a row this will be signaled to the other components to indicate the received data are valid.

The Frame Checker The Frame Checker compares received data with its reference file.

3.6 Fiber Optical Transceivers

With the Connectivity Kit comes four transceivers from Avago for 10.3125 Gbit/s data transferring. If 66b/64b encoding is used a total overhead of

$$\frac{2}{64} = 0.03125\%$$

is added. When 10 Gbit of data has been transmitted, a total of

$$0.03125 \cdot 10^9 = 312.5^6$$

bits of overhead has been sent. The data and overhead sums up to $10.3125 \cdot 10^9$ bits which means the Avago devices are capable of 10 Gbit/s transmitting and receiving.

Figure 3.5: Avago AFBR-709SMZ SFP+ transceiver.

Two EEPROM registers with address A20 and A2h on the I^{2}C bus contains values for settings that are already programmed correctly for 10.3125 Gbit/s data transferring. There are registers that contain the real-time data for the transceiver's V$_{CC}$, temperature, TX current and TX and RX power.

4 VHDL Implementation of the System

This chapter describes the VHDL components that were developed for this thesis. The system was designed to continuously output data saved in a ROM and not implement any logic for any other state of the system, i.e., pausing the transmission or any other kind of command that can be received from a higher layer. With the Xilinx IP IBERT Design (Section 3.3) the hardware could first be verified to be working. Attention was then given to the Transceiver Wizard which is integrated into Vivado. After the Wizard had generated an IP component containing the GTH transceivers it was possible (but not straightforward) to get an example design generated, too. The design does, however, not start up by itself as it is missing a reference clock input. The development of the system therefore started with the I^{2}C component so that the Si5324's registers could be written to so it would generate a clock signal with a frequency of 156.25 MHz. Also, the MUX that routes the signals to the Si5324 is configured with an I^{2}C data stream so the first code was a finite-state machine (FSM) that initialized the reference clock (Si5324) after configuring the MUX.

This chapter describes the circuits that together form a transceiver system, shown in Fig. 4.12 and Fig. 4.13. The circuits are described in the order from the TX side to the RX side. The project's top module comes last, which integrates the Transceiver Module with the example design, i.e., the GTH wrapper, and the FSM which controls the setup of the Si5324, an UART for communicating with a PC and the I^{2}C circuit which connects to the SFP modules.

4.1 Data Generator

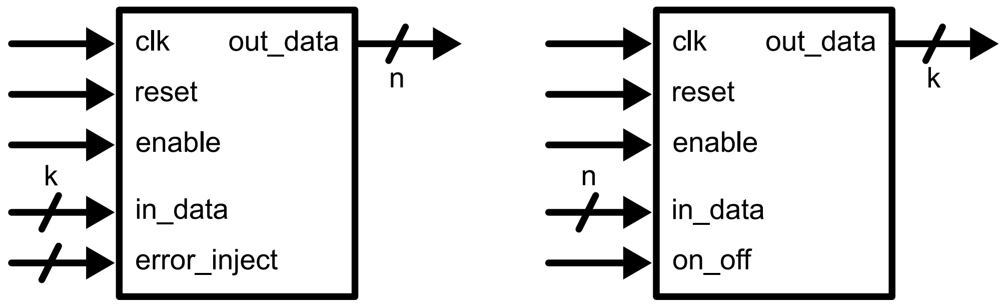
The Data Generator has a ROM which contains the data which are sent over the fiber optic channel. The standard package contains 16 different words copied from the Xilinx Example Design as default, but can easily be changed to something else and more words. The first word in the ROM is a start-word followed by just random data. The start-word is used by the bit-error rate tester (BERT) for synchronization (Section 4.5), before it starts analyzing received data for erroneous bits.

4.2 Integration of BCH Circuits

The GTH of the FPGA expects either 32 or 64 bits of data depending on its configuration which can be determined in the Xilinx Transceiver Wizard. FEC components use other widths than the GTH, e.g., an encoder can have an input of $k = 113$ bits and an output of $n = 127$ bits. To connect the outputs of an encoder and the input of a decoder to the GTH, some logic for "expanding" words to larger widths and "compressing" them to shorter is needed.

4.2.1 The Encoder and Decoder

The BCH circuits were generated with a MATLAB script provided by the assistant supervisor Christoffer Fougstedt. Changes were made on the circuits' top modules so errors can be injected in the Encoder and the Decoder's correction can be turned off, for acquiring bit-error rates without correction.



(a) Encoder. **(b)** Decoder.

Figure 4.1: Block representation of the modified BCH top components.

4.2.2 Word Expander

The Word Expander saves its input data in a register. Where in the register the data are placed depends on how many bits are already saved. When there are enough bits for the output, there are most often more bits than the output width. The remaining bits are then shifted logical right in the register and the next input word is placed after the shifted bits, so the bits will be outputted in the correct order. The challenge is then to write, read and shift, all in one clock cycle.

The resulting circuit must be synthesizable so any dynamic addressing of an VHDL *std_logic_vector* is not allowed, so instead all different cases must be covered, i.e., hard coded with a *case* statement. There is one case per every possible number of remaining bits, e.g., if the input is of the width $k = 113$, there are 113 possible remaining bits (0 to 112) and thus 113 cases to cover. To write the VHDL files, a Python script was written to generate any Word Expander of any input and output bits. Another Python script generated the correct outputs (all possible) for some provided input data which can be used with the behavioral simulations of the test-bench.

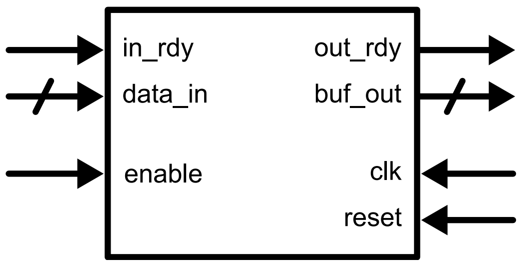


Figure 4.2: Block representation of the Word Expander component.

4.2.3 Word Compressor

When a word is “compressed” there will always be remaining bits. The remainder can maximum be the output size minus one. Since there is also the case when there are zero bits left, the total amount of possible remaining bits becomes equal to the number of output bits. The Word Compressor implements a state machine (called the input FSM) which alters between two states called *FILL* and *MOVE*.

When data is available, the *FILL* state uses VHDL case statements for placing input data after any remaining bits in the input register (like the Word Expander) and then changes to the *MOVE* state. The *MOVE* state copies as many whole words possible over to the output register and right-shifts the input register accordingly so, the remaining bits are then placed first in the register. Because the two needed operations, i.e., writing to a register and shifting it, are separated into two states, the Word Compressor will not produce a continuous output if the input size is less than twice the output size. This means for a 64-bit output the input must be ≥ 128 bits. The benefit of using two states is the area efficiency (compared to previous implementations), and thus that the large input versions are synthesizable.

Another FSM (called the output FSM) copies words from the output register to the output ports and can lock the *MOVE* state with a flag, i.e., so it cannot change to *FILL*. Thus, no data can be received until the flag is low again, i.e., when all words are outputted. The input FSM starts the output FSM with another flag. When the *MOVE* state is locked, it is however allowed to overwrite the lower word of the output register to have the output FSM to provide a continuous output (assuming there has been another input and the input register is filled). The output FSM uses case statements for moving words from the output register to the output ports, which only makes a few possible cases.

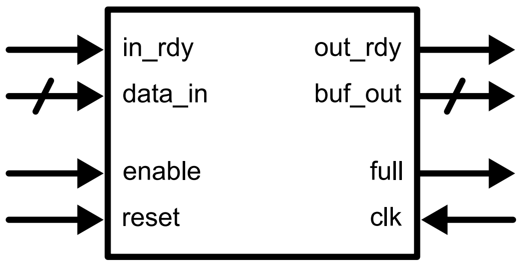


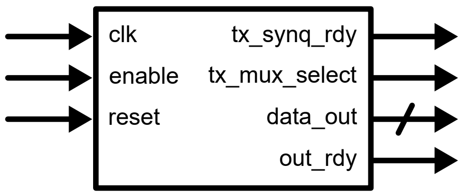
Figure 4.3: Block representation of the Word Compressor.

When the *full* signal is high the Word Compressor will not read *data_in* even if *in_rdy* is high because the FSM is in the *MOVE* state. The *full* signal will be set low again when the current state is changed to *FILL*. The Word Compressor is designed to be in a pipeline with other preceding components, and the *full* signal can be used to disable them until the output buffer is emptied so the input buffer can be copied over.

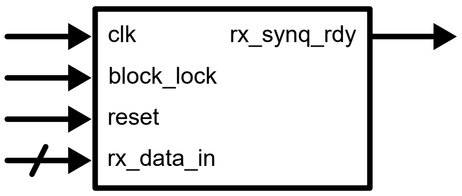
The Word Compressors are generated by a Python script which takes the sizes of the input and output vectors as arguments and writes the corresponding VHDL to a file or returns it as a string. The test-bench for the Compressors can use the same data as the Expanders (by switching output data to input data).

4.3 TX and RX Synchronization

The receiver, i.e., the FPGA itself in a loop-back system, uses a PLL as a reference clock for its RX region. The PLL takes some amount of time to lock to the incoming frequency. The TX Synchronizer (see Fig.4.4a) has a ROM of 128 random words which on start-up will be sent over the fiber-optic channel. The ideal signal for the PLL would be *"101010..."* but this sequence contains only valid 64b/66b headers (Section 2.6.2). If a Block Synchronizer (Section 3.5) receives data with only valid 64b/66b headers the odds are high it will synchronize and align words incorrectly. Therefore, the data is random but generated with a condition that allowed only three equal bits in a row for many transitions (to achieve a "high frequency").



(a) TX Synchronizer.



(b) RX Synchronizer.

Figure 4.4: Block representation of the TX and RX Synchronization.

The TX Synchronizer sends out its ROM content 128 times, i.e., (128 · 128 words), and then sends the sync-word *x"aaaaaaaa"* eight times before it stops and sets the *tx_mux_select* high. The *tx_mux_select* sets the source of *TXDATA* to the Scrambler's output instead of the Synchronizer's *data_out*. At the preceding clock cycle *tx_synq_rdy* was set high, which enables the pipeline. The pipeline has a delay of one clock cycle and thus needs an earlier release.

Fig. 4.5 shows how the RX Synchronizer (see Fig.4.4b) waits for the eight sync-words and then sets *rx_synq_rdy* high which enables the RX Descrambler.

The RX Synchronizer is not operational until the *block_lock* signal from the Block Synchronizer (Section 3.5) is high.

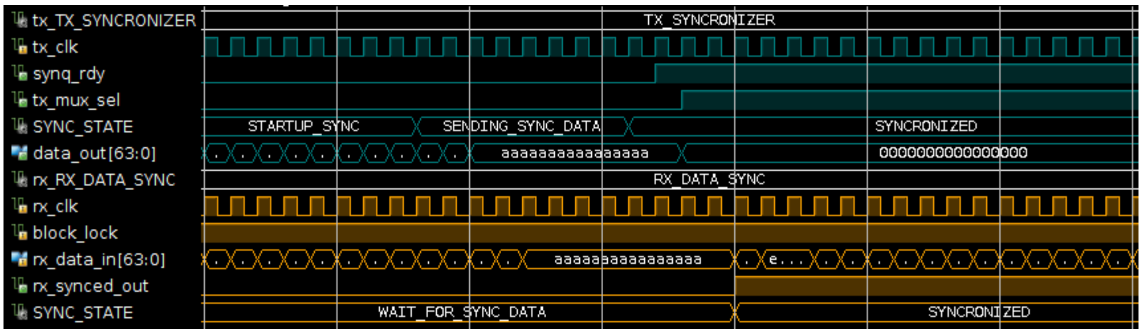


Figure 4.5: Waveform simulation of the synchronization.

4.4 TX Buffer

The Word Compressor is not having a constant output (*out_rdy* high) from its start-up which causes one clock period with *out_rdy* set to low. A simulation is shown in Fig. 4.6, with the waveforms of the Word Compressor and its preceding components.

If the Word Compressor would connect directly to the GTH, the first outputted word would be read twice by the GTH twice since it does not care about the *out_rdy* signal. Therefore, the TX Buffer saves up a number of words and then starts outputting them, allowing for the pipelined TX circuits to fill up first.

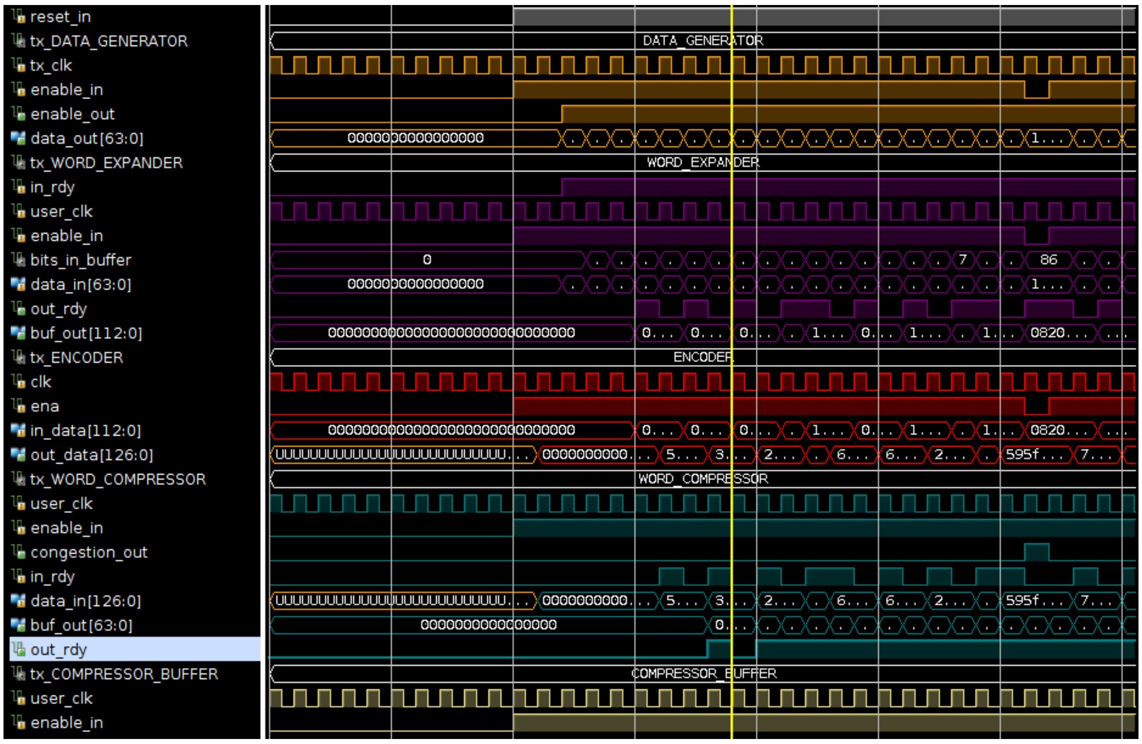


Figure 4.6: Waveform simulation with focus on the Word Compressor.

The TX Buffer will also keep encoded data (sliced to 64 bits) ready for when the *TX_synq_rdy* signal (Section 4.3) changes to high so that the GTH's *TXDATA* port instantly get new (encoded) data (instead of synchronization data).

The TX Buffer's *congestion* signal can be used to stop preceding components in a pipeline.

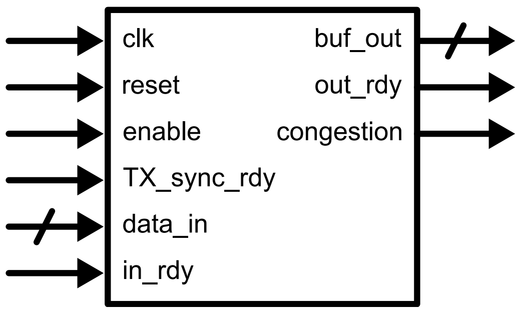


Figure 4.7: Block representation of the TX Buffer.

4.5 Bit Error Rate Tester (BERT)

The bit-error rate tester (BERT) saves and analyses all errors found by the BER Calculator. The BERT counts the errors of a predefined number of words, the default (interval) is set to 100 M words. After an interval, the *BER_out* port is updated with the number of erroneous bits found and *run_rdy* is high for one clock period. The BERT waits for the decoded 64-bit start-word *x"fb"* before it starts summing up the found erroneous bits, and keeps the *halt_out* signal high (see Fig. 4.8). The *halt_out* signal is connected to the BER Calculator to have it align the RX data correctly with the reference ROM, and is set low after the start-word.

After the BERT has finished one interval it will set the *halt_out* signal high and start again once the decoded start-word appears.

4.6 BER Calculator

The BER Calculator (shown in Fig. 4.10a) is essentially the control logic for the BER circuit (shown in Fig. 4.10b) and is idle until the *halt_in* goes low. When it is not idle, the *rx_data_in* is forwarded to the BER circuit (Section 4.6.1) with the ROM reference data.

4.6.1 The BER Circuit

The identification of how many bits that differ between two registers can be done with a rather simple algorithm. A logical XOR operation on the two registers yields a new register where every bit which is set to one indicates a bit difference. The difference in this context is a bit-error, as XOR is done on the received data and the reference data.

After the XOR operation of e.g., two 64-bit words, follows a 1-bit addition of all pair of bits, i.e., 32 additions resulting in 32 2-bit values. Next follows sixteen

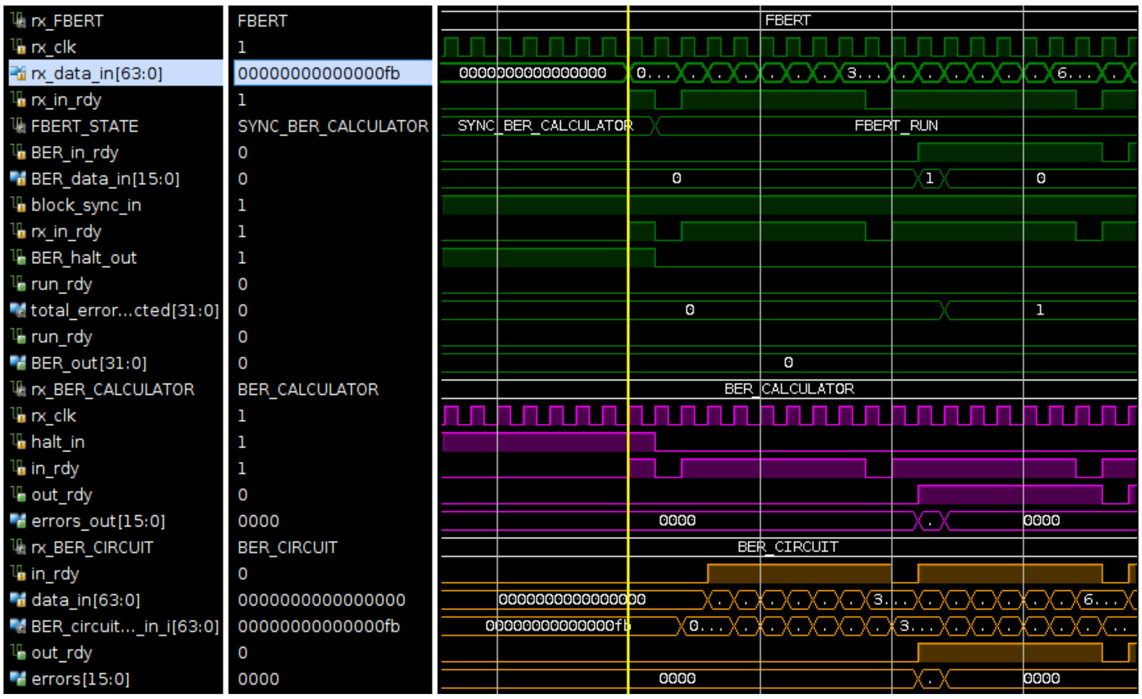


Figure 4.8: Waveform simulation of the complete BERT process.

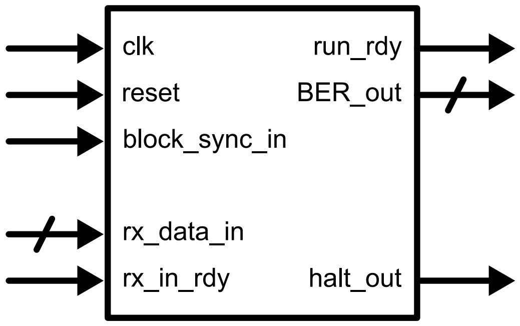
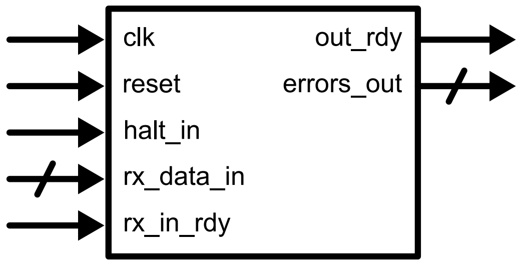
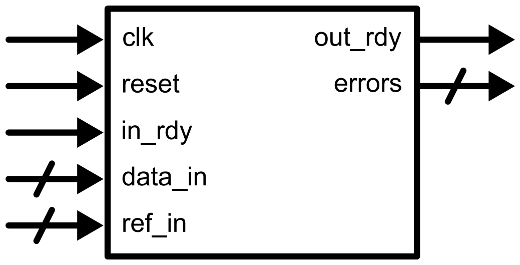


Figure 4.9: Block representation of the BERT component.



(a) The BER calculator.



(b) The BER circuit.

Figure 4.10: Block representation of the BER Calculator and the BER circuit.

additions of the 2-bit values, resulting in sixteen 3-bit values, and so on until there are two 6-bit registers left to be added together.

The total number of additions depends on the size of the vectors. VHDL is very type-safe and will require a lot of code for the algorithm to be synthesizable. A Python script was, thus, written to do the repetitive and long lines of code, e.g., the 512-bit version is around 650 lines of code. The 512-bit version can be used to, e.g., extract the BER before a 511-bit decoder corrects any errors.

The resulting circuit has the XOR and additions pipelined which causes a delayed result. The 64-bit version used in this thesis has a delay of eight clock cycles as it is using one XOR, six additions and one last operation which re-sizes the word-length to 16 bits.

4.7 Gearbox Module

The Gearbox Module connects to a GTH's *HEADER* and *TXSEQUENCE* ports which are used for 64b/66b encoding (Section 2.6.2). The output of *tx_sequence* repeats a cycle; first it outputs the header *"10"* for one clock period and the following 15 periods it outputs *"01"*, but which valid header is sent is not important as the receiving side is not concerned about them.

Because of the 2-bit overhead per word due to the 64b/66b encoding, 32 sent words will cause a whole extra 64 bits to have been sent, i.e., a whole word. To compensate for the extra word, all transmission needs to be stopped for one clock cycle [1] which is signaled with *tx_valid_out*. The logical value of *valid_out* is simply the result of comparing a counter with the integer 32, and is used to pause the TX system (Section 4.9.1). The counter's current value is copied over to the *tx_sequence* port as well, as the GTH needs an external counter [1] to count the number of words which have been passed to it.

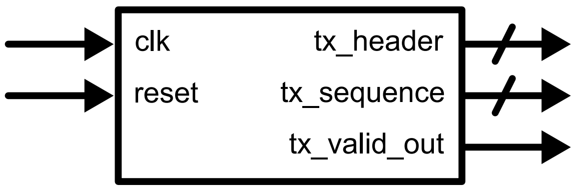


Figure 4.11: Block representation of the Gearbox Module.

4.8 Reset Circuitry

It is important that all flip-flops and state-machines are reset and that all circuits start up at the same time after a reset signal (active low) is set high. Because the complete system pipelines data through the different circuits, a small deviation on one circuit's reset signal can cause the whole system to malfunction.

In most cases a global reset works fine [15], but as systems (of circuits) expand and take up more logic of an FPGA the reset signal can become timing-critical. The "Reset Circuitry" is a solution to have all circuits provided with their own reset signal, but one which originates from the same source.

The source reset is split and carried through a pipeline of flip-flops which will cut the fan-out of a global reset and have the source reset to arrive at all circuits synchronously. The flip-flops were implemented in a VHDL process which was integrated into the Transceiver Module (Section 4.9), but could just as well have been moved into its own component, i.e., in another VHDL file.

4.9 Transceiver Module

The Transceiver Module is a top module with one TX and one RX system. All previously described components are included in either or both two systems.

4.9.1 TX System

Fig. 4.12 shows how the circuits are connected to make up the transmitting part of the system. The Gearbox Module (Section 4.7) disables the system after 32 words have been sent because 64 bits of overhead has then been passed to the GTH.

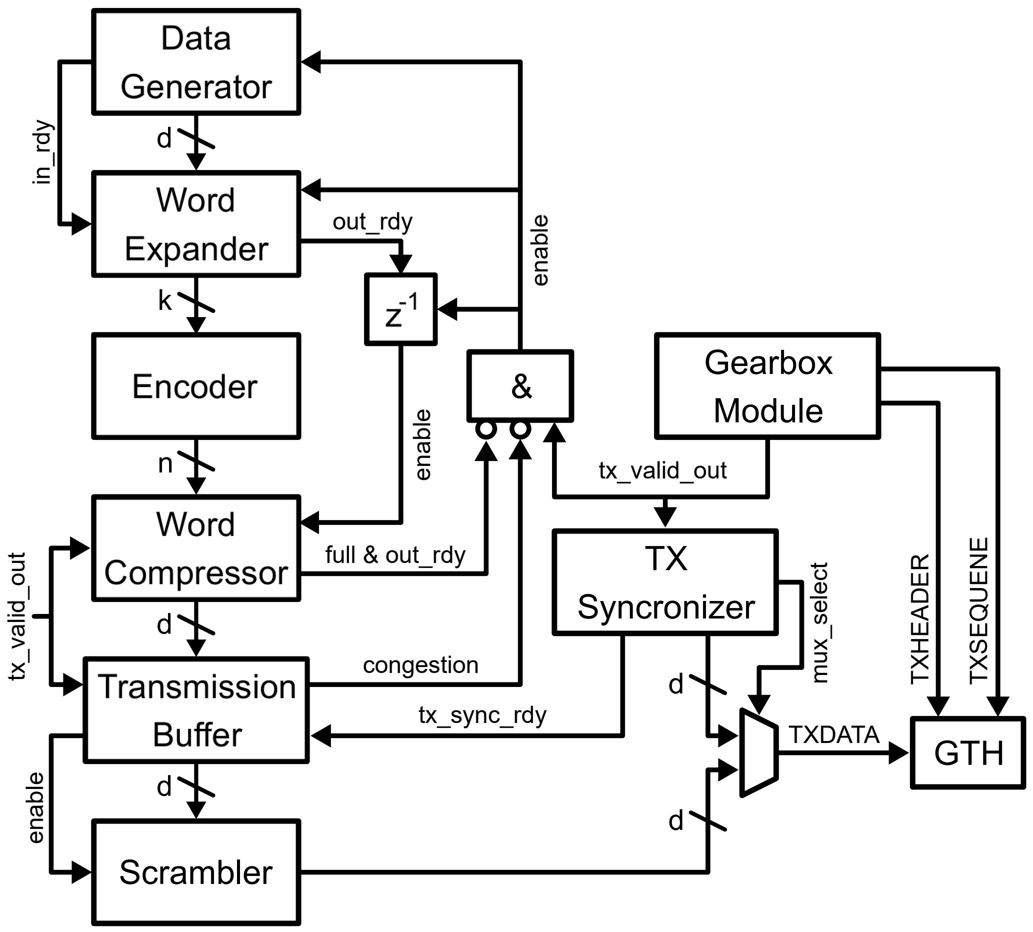


Figure 4.12: Block diagram of the transmission chain.

The TX Synchronizer (Section 4.3) will output its ROM data with the purpose of locking the RX PLL (Section 2.4), and then adjusts the MUX so that the data from the Scrambler (Section 3.5) is passed to the GTH (Section 3.2) instead. The TX Synchronizer's *tx_synq_rdy* signal is set high one clock cycle before *mux_select* because of the Scrambler's delay of one clock cycle, and the TX Buffer (Section 4.4) will be full at that time.

As the TX Buffer starts outputting data, the *congestion* signal will be set low so the preceding circuits will be enabled again and will not be halted again until the Gearbox Module disabled them.

4.9.2 RX System

The Block Synchronizer (Section 3.5) holds the system in reset until the received data are aligned to correct words by checking for valid 64/66b headers (Section 2.6.2). When 64 valid headers have been received in a row, the RX Synchronizer (Section 4.3) starts comparing the input data to the known sync-words. If all sync-words have been identified the *synced* signal is set high which will route the *RXVALID* signal to the Descrambler (Section 3.5), effectively enabling the RX system.

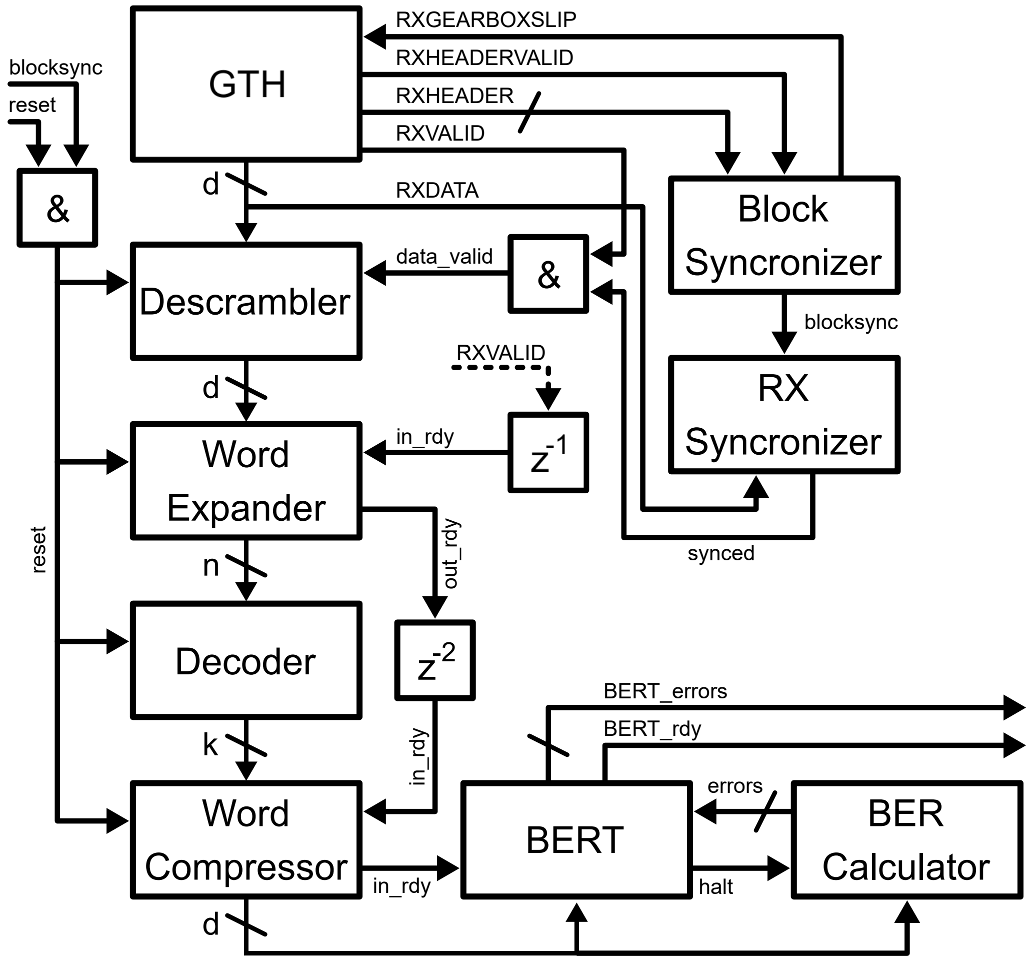


Figure 4.13: Block diagram of the receiving chain.

A Decoder has a delay of two clock periods. In order to signal the Word Compressor that decoded data is ready, the *out_rdy* signal from the Word Expander is delayed two clock cycles and routed to the Word Compressor's *in_rdy* port.

The BERT circuit (Section 4.5) starts by comparing the data to the start-word *x"fb"*. When the start-word is identified the *halt* signal will be set low and the BER Calculator (Section 4.6) will start comparing the received data with the reference data. The BERT counts how many words that have been checked for errors, and will restart when the default of 100 M words have been counted which takes ≈ 0.5 s.

4.9.3 Transceiver Top Module

In Fig. 4.14 we see the block diagram of the Transceiver Top Module which contains a TX and a RX system.

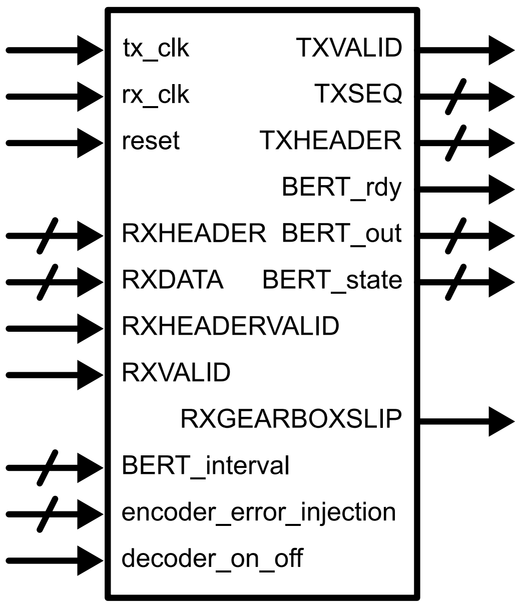


Figure 4.14: Block diagram of the Transceiver Module.

4.10 PC Communication

The PC communication is used for transferring the status of a system, e.g., the current bit-error rate or the power of a signal measured by one of the Avago transceivers (Section 3.6). The VC709 board has a USB connector which is used to connect the FPGA to a PC. Additional logic is needed for serializing the data, i.e., a UART and a program which presents the data on the PC.

4.10.1 UART

The FPGA is connected to a CP2103 USB-to-UART bridge, with the FPGA on the UART side. The CP2103 can use Baud rates of 300 Baud to 1 MBaud [16]. The generic variable for the UART's Baud rate is set to 921 600 Baud. The UART circuit is shown in Fig. 4.15.

The circuit will start serializing the data passed to its *TX_byte* port when *TX_start* is set high. While the circuit is serializing data the *TX_busy* port is

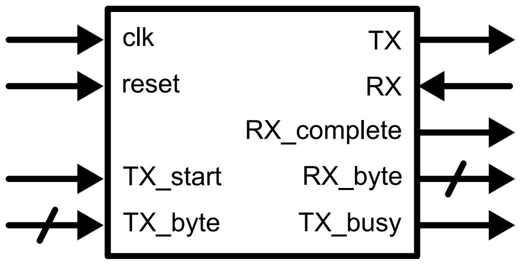
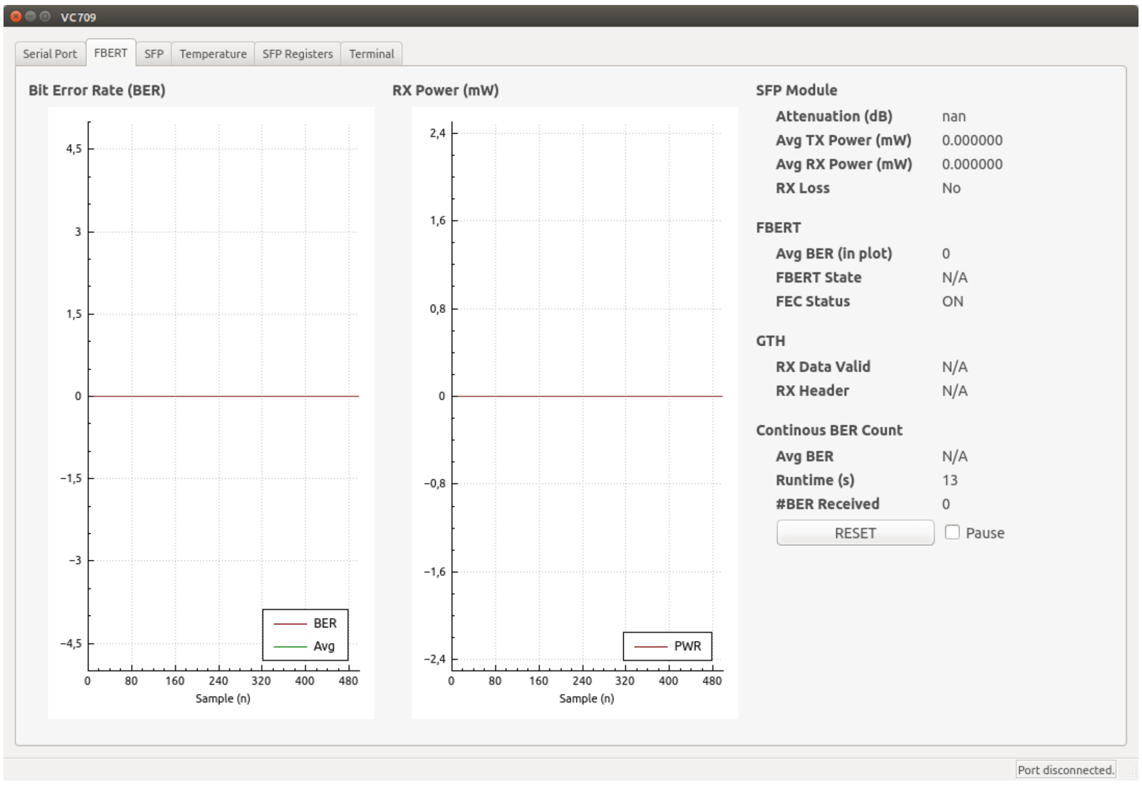


Figure 4.15: Block representation of the UART module.

high. When a byte has been received it will be copied to the *RX_byte* port and *RX_complete* will be high for one clock period.

4.10.2 The Real-Time Monitoring Software Using Qt

The Qt program is made in C++ and is using the *qcustomplot* library for plotting the real-time graphs. Two graphs are shown in Fig. 4.16. All other real-time data from the SFPs are plotted under the "SFP" and "Temperature" tabs. The register values of the SFPs are printed under the "SFP Registers" tab. The "Serial Port" tab contains the settings for connecting to the VC709 over USB. In the bottom, the status of the serial port is shown. The "Terminal" tab lets a developer print anything that could be useful, e.g., debug messages.



SFP Module

Attenuation (dB)	nan
Avg TX Power (mW)	0.000000
Avg RX Power (mW)	0.000000
RX Loss	No

FBERT

Avg BER (in plot)	0
FBERT State	N/A
FEC Status	ON

GTH

RX Data Valid	N/A
RX Header	N/A

Continous BER Count

Avg BER	N/A
Runtime (s)	13
#BER Received	0

Port disconnected.

Figure 4.16: The real-time program for the VC709.

The leftmost graph in Fig. 4.16 plots the BER and its average value. The received

BER is an integer so the floating-point conversion (to an average value) is done by the PC, and the number of corrected words is known.

The middle graph plots the RX signal power. The rightmost column shows data separated under four titles. The SFP's power data is used to calculate the channel's attenuation (in dB). The state of the BERT is shown to indicate that the system is running, as well as if the error correction is on or off.

Furthest down ("Continuous BER Count") are some tools for measuring the BER in real-time. When the attenuation of the channel is changed, the reset button can be used to start a new measurement. The average BER is shown and has no limit for the number of samples it can save. A timer shows how long time that has passed since a reset. A check-box (pause) will stop updating the presented values but any received data will still be saved, which makes it easier to see the less significant decimals.

4.11 I^{2}C Communication

The I^{2}C circuit can initialize the Si5324's registers with values that are saved in a ROM. It can also read the registers of all four SFP modules, or just some registers by using the *index_start* and *index_stop* ports. Because some of the registers are constant they only need to be read once. Selecting exactly which registers should be read is done when the real-time data is extracted.

The circuit's FSM can be started to read if *idle_state* is high by setting the *start_read* port high. The FSM will signal a complete read by setting the port *read_done* high with the resulting (one byte) data outputted on the *read_data* port. With the *dev_sel* (device select) port a user decides to do either a read of the SFPs ('0'), or a write to the Si5324 ('1'). With the *sfp_sel* (SFP select) port, either one (of two) EEPROM addresses is used in a read operation.

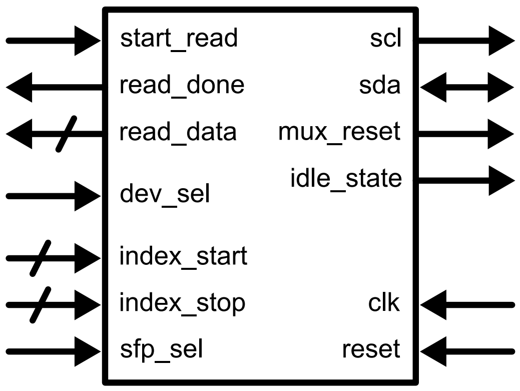


Figure 4.17: Block representation of the I2C circuit.

The FPGA has two dedicated ports, i.e., SCL and SDA, for the I^{2}C clock signal and data signal. Fig. 4.18 shows how an SFP module is accessed through two different MUXes. The SFPs all have the same address so the MUX is necessary. The MUX closest to the FPGA can connect to the Si5324 or the other MUX.

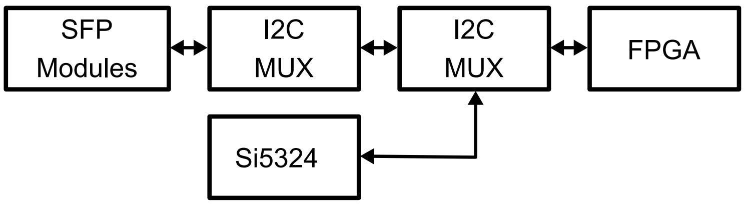


Figure 4.18: Description of I^{2}C connections to the FPGA.

The reset pin of each I^{2}C-MUX is connected to one FPGA port which is routed to the I^{2}C-circuit's *mux_reset*. Which port a MUX connects to is decided with one of its register values, which can be accessed through I^{2}C.

4.12 Top Module

The Transceiver Wizard in Vivado can generate a wrapper for the GTH (Section 3.5) containing the IP declaration and the FSMs for resetting the TX and RX hardware, e.g., the PLL. The wrapper also contains declaration of the clock components and the GTH2_COMMON circuit which initializes the QPLL shared among the four GTHs that are connected to the SFP modules on the VC709 board.

The Top Module (see Fig. 4.19) is using the GTH wrapper from the example design (Section 3.5). The Transceiver Modules are first held in reset by the FSM in *Main Process* while the Si5324 is being configured. After the configuration is complete, the reset signal is instead routed to the TX FSM's (from the wrapper) reset output. The TX FSM will assert its reset signal when the reference clock is stable, i.e., the QPLL inside the FPGA GTH, not the Si5324 (which is the reference for the QPLL).

The RX FSM waits for the RX PLL to lock to the incoming data stream's frequency before its reset is asserted. The TX Synchronizer should send out its training bit-pattern long enough so the RX FSM is ready before actual data arrive since the RX side is provided the same reset as the TX side, and not the RX FSM's reset signal.

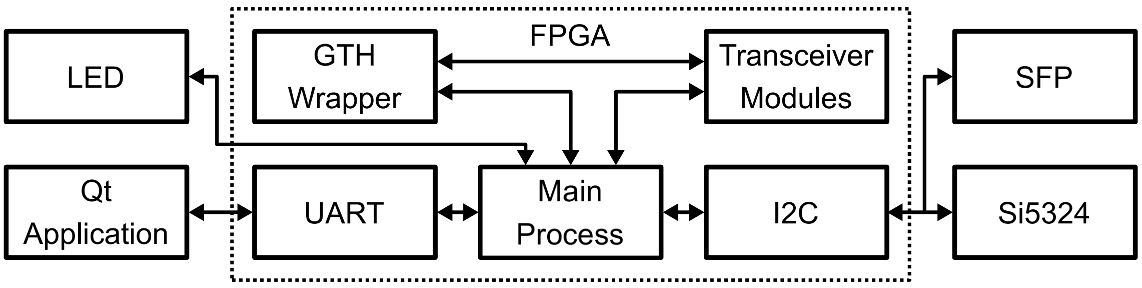


Figure 4.19: Block diagram of the Top Module.

The Top Module's process *Main Process* also forwards bytes from the I^{2}C to the UART and sends headers before the data so the receiving PC can decide what data it is. Therefore, the UART is working at full speed (921 600 Baud) and the I^{2}C clock (SCL) is 100 kHz, guaranteeing that the UART is not busy when I^{2}C has fetched one byte of data. The *Main Process* has one state which is a delay for 20 ms (a host computer would receive new data approximately 50 times per second) so, the exact time depends on how many clock cycles it takes to fetch and send the data in the other states. The registers of the SFPs are only sent *once*, at start-up, because they are constants. Only the real-time data on the SFPs are later read and sent together with the BER and the GTH's status. A subsection now follows which is explains the *Main Process's* FSM.

4.12.1 Main Process

The Main Process in the Top Module controls the serial communications, i.e., controls the I^{2}C circuit and the UART. It starts with resetting the Si5324 per its datasheet and then writes values saved in the I^{2}C component's ROM to the EEPROM of the Si5324. The process is clocked by *drpclk_in_i* which is the FPGA's main clock of 200 MHz and starts in the SI5324_RESET_1 state.

Two registers, CURR_SERIAL_STATE and NEXT_SERIAL_STATE, are used for changing states where the latter is used for deciding on which state to return to after leaving a state that can be reached from more than one state, i.e., the ones in the middle of Fig. 4.20. Therefore, the colors of the arrows in Fig. 4.20 aims to show how the current state changes, i.e., a red arrow which leads out from a state, was reached from a previous state with a red arrow, too. All states will be explained further below.

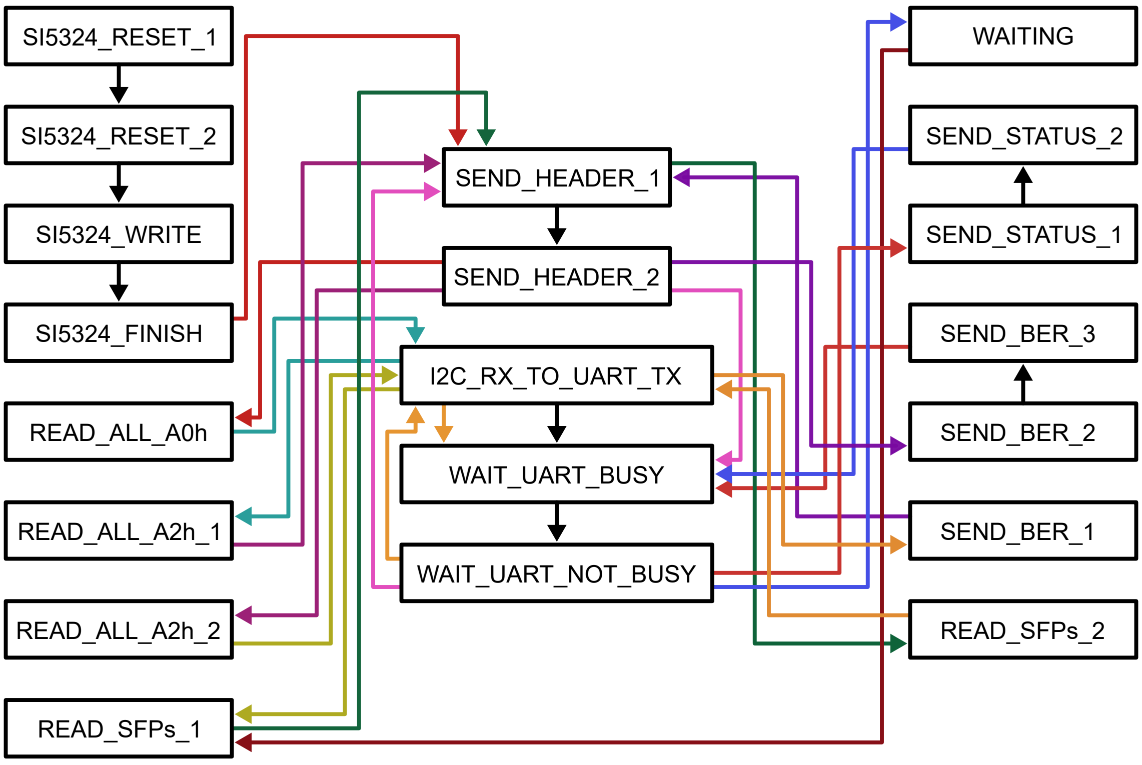


Figure 4.20: The FSM process for UART and I2C.

SI5324_RESET_1 It is specified in the Si5324 datasheet [17] that the minimum time the reset-pin should be set low is 1 μs, which is done by this state.

SI5324_RESET_2 A counter is used to wait 10 ms before changing state, as the Si5324 datasheet [17] specifies that a waiting of 10 ms is necessary for the Si5324 to be "access ready" after a reset.

SI5324_WRITE In this state the I^{2}C component's *I2C_device_sel* port is set high which means the I^{2}C FSM will write its ROM with register values to the Si5324. The I^{2}C component will do so if *I2C_start* is set high.

SI5324_FINISH This state sets the *I2C_start* port low again and waits for the I^{2}C circuit to finish the configuration of the Si5324. Because of the start-up delay of the I^{2}C circuit, both the *I2C_idle_state* output must be high and a counter must be high enough before the current state will change (to avoid malfunction). The *I2C_idle_state* signal will not be high when this state is first reached, so the counter makes sure the I^{2}C circuit gets enough clock cycles to set the *I2C_idle_state* port low. Before a read from the registers of the SFPs, a header is sent over the UART which indicates that the following serial data are from the A0h registers of the SFPs. This state sets the register *serial_header* to 0xA1 which is used by the SEND_HEADER_1 state.

SEND_HEADER_1 and SEND_HEADER_2 Sending headers before any data helps the application running on the receiving side to decide on what the data are. Any preceding state to SEND_HEADER_1 sets the *serial_header* register which is passed on to the *TX_byte* register, the register which is used as the input to the UART circuit. The SEND_HEADER_1 state starts the UART by setting the *TX_start* port high, and the SEND_HEADER_2 sets it low again, and changes the current state to the WAIT_FOR_UART_BUSY state. The register NEXT_SERIAL_STATE is used in the WAIT_FOR_UART_NOT_BUSY for changing to the correct next state.

READ_ALL_A0h This state sets up the I^{2}C circuit's ports for reading the registers (of interest) at address A0h of the SFP modules. Because some of the registers does not contain any valuable information, the I^{2}C component's index ports are set to zero and 95, which makes up a total of 96 bytes to read per SFP. The register *RX_byte_counter* is set to $(96 \cdot 4)$ which is used by the *I2C_RX_TO_UART_TX* state.

READ_ALL_A2h_1 This state has a delay so it is certain that the I^{2}C and UART are ready with their operations started by the preceding state. After the delay, the process of sending out a header is repeated which signals that the subsequent serial data are from the SFPs (address 0xA2) registers.

READ_ALL_A2h_2 This state has the same functionality as the READ_ALL_A0h state, but here the index is set from 0 to 117.
The *RX_byte_counter* register is set to $(118 \cdot 4)$.

I2C_RX_TO_UART_TX This state decreases the value of register *RX_byte_counter* every time the I^{2}C circuit signals out that a read is done. Every time a read is done the read byte is forwarded to the UART circuit which is signaled to start at the same time.

When the byte counter reaches zero the current state changes to the WAIT_FOR_UART_BUSY state which will eventually change to the state saved in the NEXT_STATE register.

WAIT_FOR_UART_BUSY This state waits for the UART to signal that it is busy, and then changes the current state to the WAIT_FOR_UART_NOT_BUSY state. This state is needed because it takes one clock cycle for the UART to signal it is busy so the UART states are used to synchronize the state transitions with the UART's operation.

WAIT_FOR_UART_NOT_BUSY When the UART is not busy anymore, the current state changes to the state saved in the NEXT_SERIAL_STATE register.

READ_SFPs_1 This state starts the repetition of sending a header for signaling that the following data are real-time values from the SFPs, i.e., voltage, power and current.

READ_SFPs_2 This state is reached after the header has been sent. The I^{2}C circuit's index is set from 96 to 105, i.e., the registers that contain the real-time data. Ten bytes will be read from every SFP, and therefore the *RX_byte_counter* register is set to 40.

SEND_BER_1 This state repeats the sending of a header. It will send out the current BER value from the BERT circuit. If more than one Transceiver Module is used in the system, here it is possible to choose which output should be sent with the DIP switches.

SEND_BER_2 The output of the BERT is a 32-bit value so four bytes will now be sent. This state uses the register *BER_counter* to decide which byte of the BERT output to send. When the last byte has been forwarded to the UART the NEXT_SERIAL_STATE register will be set to SEND_STATUS_1, else it will return to itself after the current state has shifted to SEND_BER_3 and then the two UART states.

SEND_BER_3 This state sets *TX_start* to low and changes the current state to the WAIT_FOR_UART_BUSY state.

SEND_STATUS_1 and SEND_STATUS_2 In this state the *TX_byte* register (passed to the UART) is filled with different information about the system, e.g., GTH information and whether the Decoder circuit is on or off. The succeeding state will just set the *TX_start* signal back to low and the next state will be WAITING.

WAITING This state will change the current state when its counter has reached 4000000 which takes 20 ms since the system clock is 200 MHz. Not including the time it takes to pass through the other states, this will yield a period time of the state machine of approximately 50 Hz. While in this state, another timer is used to toggle an LED for indicating to a user that the process is running.

4.13 Automatic Generation of Systems

Vivado projects with different BCH circuits can be automatically generated with a Python script. First, the different VHDL files for the BCH circuits should be placed in a folder, which will be searched through by the Python script. A folder which contains the VHDL files must be named, e.g.,

```
t1_k247_n255
t2_k493_n511
t3_k993_n1023
```

so the script can extract the values (n,k,t) as integers (Section 2.7). The integers are passed to functions which generates circuits and writes them to the files described below. The scripts that are used in the automatic process are further described in Appendix A. There is a Vivado reference project which is copied to the new location. The project is lacking all the files that are generated which are copied over to their respective location.

transceiver_module.vhd The top module for the transceiver system.

word_expander_64IN_to_kOUT.vhd The k is an integer. This is the TX Expander which expands a 64-bit word from the Data Generator to a k-bit word for the Encoder circuit.

word_expander_64IN_to_nOUT.vhd The n is an integer. This is the RX Expander which expands a 64-bit word from the GTH to an n-bit word for the Decoder circuit.

word_compressor_nIN_to_64OUT.vhd The n is an integer. This is the TX Compressor which takes the Encoder's output and compresses it down to 64 bits which is forwarded to the GTH.

word_compressor_kIN_to_64OUT.vhd The k is an integer. This is the RX Compressor which takes the Decoder's output and compresses it down to 64 bits which is forwarded to the BER and BERT circuit.

enc_reg.vhd The Encoder's top module has no error injection as default when it is generated with MATLAB. The project generator rewrites the *.vhd* file and adds a 2-bit port for (0 to 3) possible errors.

bch_peterson.vhd The MATLAB script does not generate a top module. The project generator will call another script for this and also adds a port and logic in it for turning the error correction on and off. This makes it possible to acquire a BER both with and without error correction.

gtwizard_0_exdes.vhd This file is based on the Xilinx example project and is the top module of the Vivado project. The project-generating script calls another script for this file which changes all the component declarations in a consistent manner.

transceiver_module_tb1.vhd The test bench for the system. Another Vivado project is copied over to another folder along with all the new files to form a new configuration of the copied project.

4.14 Testbenches

All circuits were simulated individually to some extent for verifying their functionality. This section shortly describes the testbenches for the three larger and more complex circuits.

4.14.1 Transceiver Module

This testbench connects a Transceiver Module's RX system to its TX system. A component called GTH_SIM connects the two through registers, causing a delay of a few clock cycles. The intention of the GTH_SIM component was to inject invalid headers to observe the behavior on the RX side but was never utilized.

The testbench has a prepared waveform file which names the waveforms and colors them differently. All the waveforms are presented in chronological order from the TX Data generator to the RX BERT. Per default, a successful simulation should have the BERT's errors output set to zero always. Making changes to the Data Generator's ROM or the reference data can be used as a method to verify the BERT's functionality.

4.14.2 Word Expander

To verify the Expanders, all the possible outputs should be checked. All possible outputs are many and are therefore generated with a Python script. The input data for the simulations was set to be a copy of the Data Generator's ROM which is 16 rows of 64-bit words, which will give several different outputs of an Expander. If an erroneous output is found this will be seen in the waveforms but the simulation will continue the running loop, thus it is possible to see if all subsequent outputs are wrong after a single error. The simulation ends with a message about whether the simulation was correct or not.

4.14.3 Word Compressor

The input data for the Compressor is the data that the Expanders should output, which means that the correct output should then be the 64-bit words (the input to an Expander). This test bench is therefore like the Expanders.

5

Results

This chapter will first cover the synthesis and implementations of the different systems which were done in Vivado. The systems are using different BCH circuits and therefore, different Word Expanders and Word Compressors. Before presenting the results of the experiment that was conducted with the systems, the material used in the experiment will be presented.

5.1 Synthesis

A total of nine systems were developed and each uses a different BCH(n,k,t) circuit. The systems BCH versions are presented in Table 5.1. All the systems were synthesized and implemented, but with only one Transceiver Module during the development because of the long time it took to complete them, e.g., 1 h for System 9. This means that the experiment (Section 5.4) was performed with one Transceiver Module (Section 4.9.3) implemented in the FPGA, but only one was needed anyway as one fiber-optic channel was connected in the experiment. The other three modules which can be connected to the other GTH channels were therefore commented out before starting the implementation to reduce the development time.

Table 5.1: The different implemented systems BCH variables.

Name	n	k	t	Overhead (%)	Efficient data rate
System 1	255	247	1	3.137	9.686 Gbit/s
System 2	511	502	1	1.761	9.823 Gbit/s
System 3	1023	1013	1	0.977	9.902 Gbit/s
System 4	255	239	2	6.274	9.372 Gbit/s
System 5	511	493	2	3.522	9.648 Gbit/s
System 6	1023	1003	2	1.955	9.804 Gbit/s
System 7	255	231	3	9.411	9.059 Gbit/s
System 8	511	484	3	5.284	9.471 Gbit/s
System 9	1023	993	3	2.933	9.707 Gbit/s

All projects were synthesized, implemented and tested on the FPGA successfully. Instead of presenting the synthesis results of all nine different systems in this section, we will only look at the largest project which uses four modules with a BCH(1023,993,3) circuit. Table 5.2 shows the synthesis summary. Because the largest system, i.e., the one that uses the most area, passes its synthesis so can all the others, too. The synthesis time for this system was approximately 20 minutes.

Table 5.2: Synthesis summary of four modules using BCH(1023,993,3).

Resource	Utilization	Available	Utilization %
LUT	192511	433200	44.44
LUTRAM	4	174200	0.00
FF	73416	866400	8.47

Since the whole system only utilizes 44.44 % of the lookup tables (LUTs) there is a possibility even larger BCH circuits, i.e., for t $\geq$ 4, can be implemented, too. Table 5.3 shows some of the hierarchy breakdown of the components and their respective utilization. Recall that the *support* component is the wrapper from the example design which contains the FSMs for resetting the GTH hardware and PLLs.

Table 5.3: Synthesis hierarchy of four modules using BCH(1023,993,3).

Name	Slice LUTs	Slice Registers	F7 Muxes	F8 Muxes
Available	433200	866400	21660	10830
tm0	48198	17981	530	192
tm1	47535	17978	530	192
tm2	47494	17962	530	192
tm3	47494	17962	530	192
I2C	278	164	3	0
UART	24	15	0	0
support	1072	1037	0	0

5.2 Implementation

The implementation of all nine systems was successful, including the bit-file generation. Fig. 5.2 shows the four implemented Transceiver Modules (Section 4.9.3) using BCH(1023,993,3) circuits in different colors (tmX), the I^2C and UART circuit. Fig. 5.1 shows one of them (tm0) zoomed in, with the circuits in different colors. Visible on bottom the right, colored in orange, are the GTHs. The implementation time for this system was approximately 40 minutes, using an Intel i7-4710HQ and 16 GB of RAM.

5.3 Experiment Equipment

One way to demonstrate the complete real-time system is to use an OLA-54 (Optical Level Attenuator [18]), shown in Fig. 5.3a, which was borrowed from the MC2 department (thanks to Lars Lundberg) and inserted in the fiber loop (see Fig. 5.3a).

By adjusting the attenuation wheel of the OLA-54 unit, it becomes possible to directly control the optical signal and regulate the RX Power presented in Fig. 5.6. The unit is passive, i.e., operates without drawing any power.

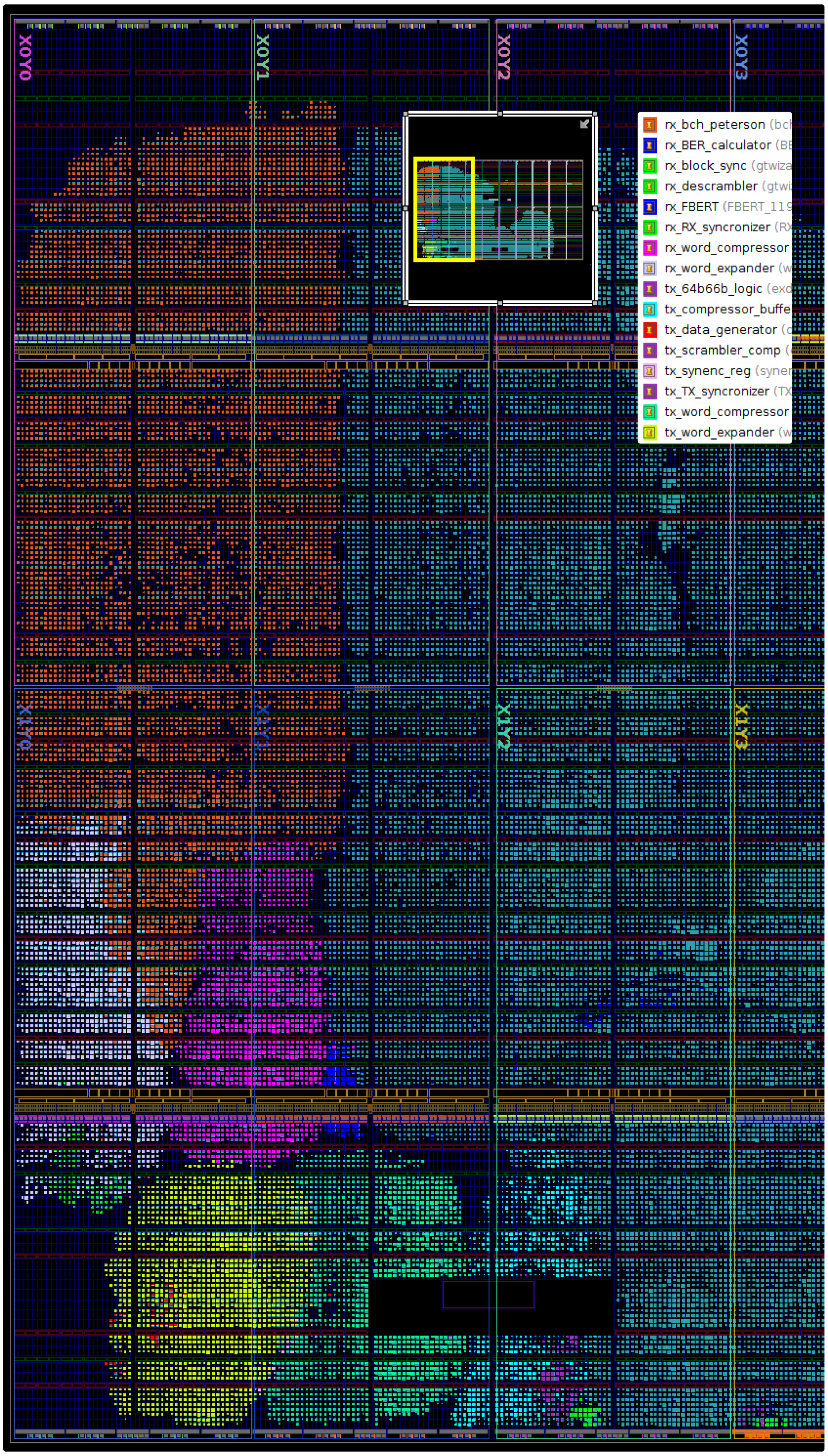


Figure 5.1: One Transceiver Module zoomed in and colored.

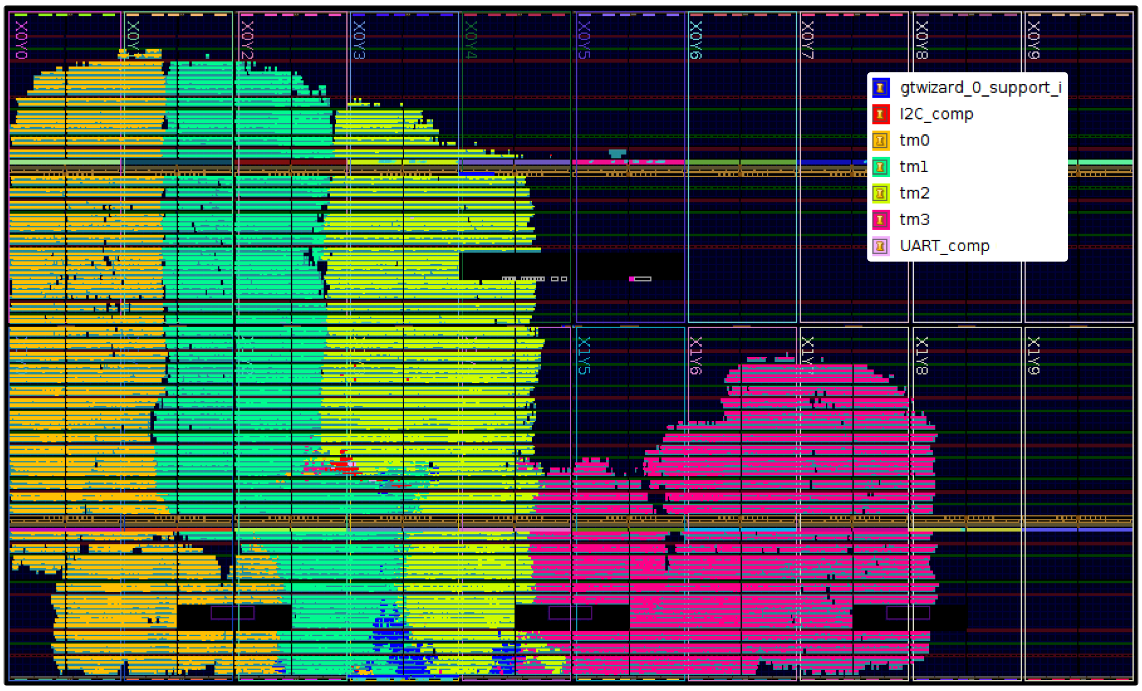
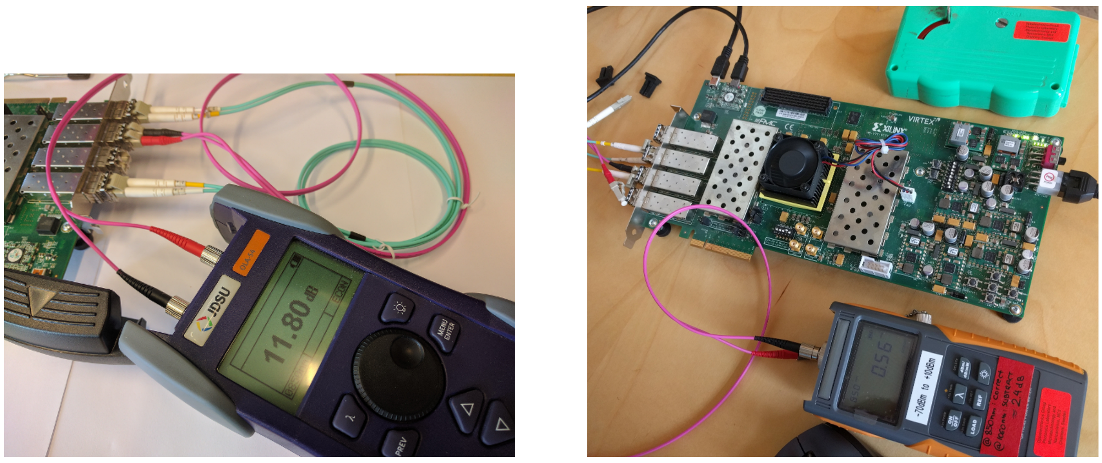


Figure 5.2: FPGA partitioning of the system with four BCH(1023,993,3) circuits.



(a) OLA-54. **(b)** Power meter.

Figure 5.3: Attenuator and power-meter units from MC2.

The optical-power meter (an OPM-G-A, also from MC2) shown Fig. 5.3b measured 0.56 mW from one of the SFP transceivers. The Qt program which is using the data extracted from the SFP showed a power of 0.5815 mW. Thus, there is a difference, but it is close enough to say that the data are extracted and interpreted correctly from the SFPs registers.

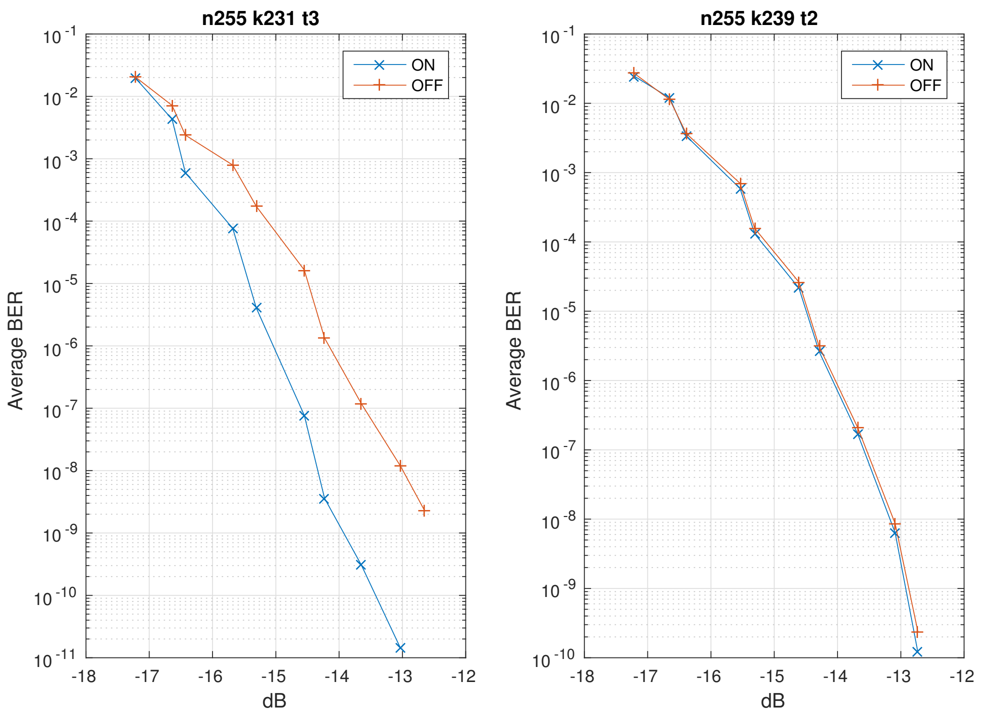


Figure 5.4: Plotted results for two BCH circuits with block-size n = 255.

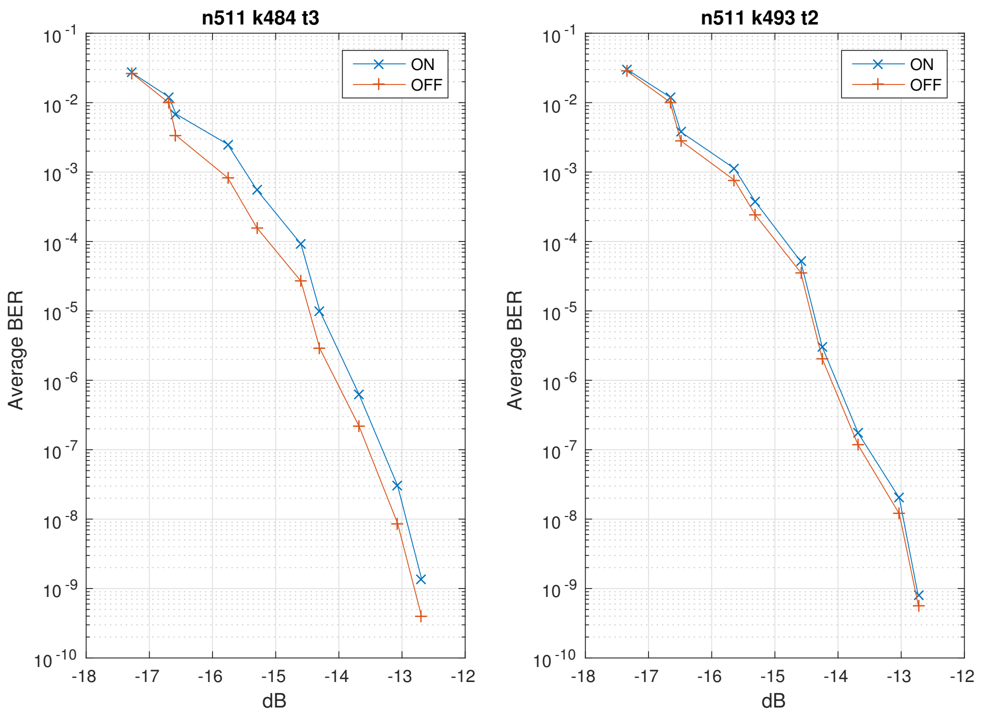


Figure 5.5: Plotted results for two BCH circuits with block-size n = 511.

5.4 Results of the Experiment

During the development of the systems it was noticed that the system with a BCH(255,231,3) circuit appeared to lower the BER the most and therefore the experiment started with it first. The BER was averaged for ten different attenuation levels for each system. The average measuring time was around 1 minute, but with a high attenuation the average BER reported by the Qt program (see Fig. 5.6) stabilized much faster and could be determined in less than 1 minute. The BER results for different attenuation of the signal are shown in Fig. 5.4 and Fig. 5.5, with the BER on the y-axis and the attenuation on the x-axis. The plots are colored blue and red for the error-correction turned on and off respectively.

The results for two systems with a BCH block-size n = 255 are shown in Fig. 5.4. The t = 3 system corrects errors much better than the t = 2 system whose improvement is very low, still the former's performance is lower than expected. Because the t = 2 circuit was almost not improving the BER at all, the t = 1 system was not used because we expected it to be even worse.

The resulting measurements for the systems which used a BCH block-size n = 511 are shown in Fig. 5.5. The results are interesting, here the bit-error correction increases the BER. Because this is the opposite of what is wanted, the other systems with larger block-sizes were not considered as we expected even worse results with increasing block sizes.

The attenuation might cause bursts of errors and it would be interesting to see the words which contain errors by sending them to the Qt program for analysis. Perhaps there is something to learn about how the errors appear in the vectors which can explain why the BCH(255,231,3) circuits work best.

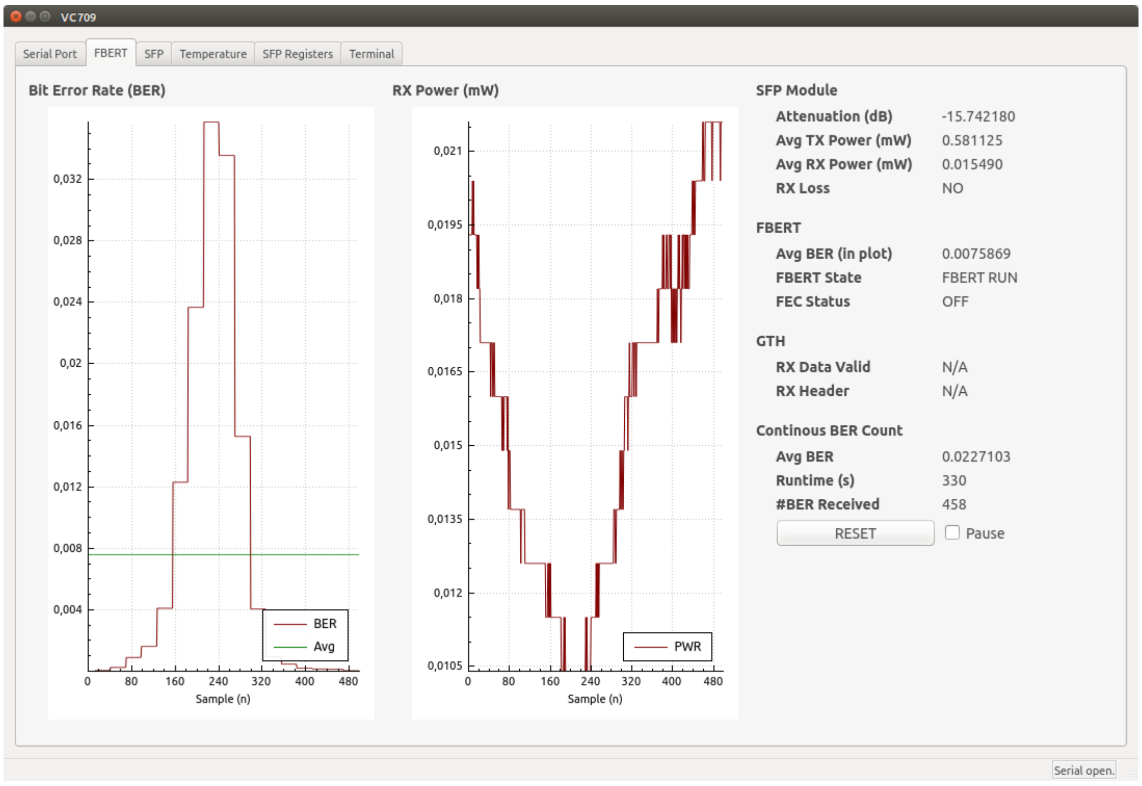
5.5 Real-Time Monitoring

One tab of the program is shown in Fig. 5.6, a screen dump from when the real-time monitoring system was connected to an attenuator, showing how the BER plot grows (left blue) as the RX Power plot decreases (red middle), for one TX/RX system. The *qcustomplot* libraries work good.

The first version of the GUI was using pyQt (Python Qt) and was developed on a Linux PC. Getting that version to start on any computer can be time consuming because of pyQt's dependencies, which is why the final version is made in C++ so it can be compiled and run easily on all computers. Compiling and running the C++ program on a Windows machine was successful.

The right column's "Continuous BER Count" section was helpful when conducting the experiment and was added just before the experiment. The code is structured and commented, in order to make it easy for other users to modify the GUI.

Because the FPGA sends out the BER continuously, several received values will be copies. The C++ code was adjusted to save the last registered BER value so that all subsequent received values can be compared to it, and if the two values differ the received will be added to the vector which saves the BERs. There is a risk that there will arrive to equal values from two runs of the BERT circuit so there is also a



SFP Module

Attenuation (dB)	-15.742180
Avg TX Power (mW)	0.581125
Avg RX Power (mW)	0.015490
RX Loss	NO

FBERT

Avg BER (in plot)	0.0075869
FBERT State	FBERT RUN
FEC Status	OFF

GTH

RX Data Valid	N/A
RX Header	N/A

Continous BER Count

Avg BER	0.0227103
Runtime (s)	330
#BER Received	458

Figure 5.6: The Qt program presenting the real-time data from the FPGA.

counter which solved that problem so two equal values still can be sent. While this is a brute force solution, it was easier to do rather than adding logic in the FPGA and re-implement all nine systems.

5.6 ASIC Power and Area

The ASIC simulations of the FEC circuits have been left outside of this work. It was planned for in case there was time, but getting all systems to work took up all time and had to be prioritized.

6 Discussion

In this chapter's sections, some parts of the thesis are discussed. Some observations made while working with the project are worth pointing out and should be of interest to anyone continuing with further development.

6.1 VHDL Implementation

All nine systems that were implemented worked on the FPGA, although only four were used in the experiment. One notable thing about the systems which can correct three errors was that the error injection to the Encoders did not work, i.e., the errors never showed up. The nine systems with four Transceiver Modules are thus untested. The Python scripts which generated all the projects are real time savers, and will be very helpful if other systems should be implemented as a new BCH circuit only should be placed in the folder with all the others. If it is possible to synthesize and implement Xilinx's IP components (the GTH wrapper) from a virtual terminal, a TCL script could be written to do so and be called from Python and thus automating the procedure even more. In that case, one would only have to call the script and come back a few hours later to test the results of all systems.

6.1.1 Word Expander and Compressor

The Word Compressor is the circuit I have spent most time on developing. It resulted in many versions and simulations. It took a couple of weeks to implement a fully synthesizable and area efficient version. I made several attempts and finally decided to use the one that needed an input width that is twice or equal to the output width (for a continuous output). The con is that it is not possible to use the BCH circuits with a block size of n = 127 because the GTHs have a 64-bit input.

The area of an Expander can probably be more efficient if the case statements would be replaced by another solution, perhaps something like the Compressor, i.e., separating the writing, reading and shifting of bits into different states.

6.1.2 The Data Generator and Scrambler

Because of the Scrambler, the TX output is essentially random although the Descrambler can restore the words, since it is using the same seed. Therefore, the sixteen words saved in the Data Generator are enough, and could also be argued to be too many. Because the Scrambler and Descrambler must have the same seed, the system will stop working if the fiber-optic channel is disturbed too much. When the GTH's *RX_VALID* signal is low, the received word will not be accepted by the RX System and the Descrambler will no longer de-scramble the subsequent words correctly. The systems have no reset function for this so the bit-file must be re-uploaded to restart the FPGA, but it can be avoided by not raising the attenuation over 19 dB, too.

6.1.3 The TX Compressor Buffer

This circuit is using more area than it needs to and can be optimized. The total area it is using is however small relative to the other components, thus any optimization was never attempted since it was easier to leave it working as it was.

6.1.4 Top Module

A Top Module with four Transceiver Modules has not been verified to work due to lack of time. There are still a lot of unused signals left from the example circuits that were removed. When Vivado synthesizes the projects, they are optimized away but they could be removed for easier reading a fewer warnings about unused signals.

6.2 The VC709 Board

6.2.1 Serial Communications

The serial communications between the FPGA works great but is at a low speed using the on-board USB to serial converter on the VC709. The Silicon Labs CP2103 USB-to-UART bridge device only supports a maximum data-rate of 1 MBaud. The I^2C components managed to work on a serial clock (SCL) of 400 kHz, even if the data-sheet of the Si5324 does not specify any maximum frequency. The design in the top module needs the UART to use a higher Baud-rate than the I^2C so it is certain that the UART is done sending data before the next byte from the I^2C is ready, therefore the SCL is 100 kHz.

6.2.2 The Si5324

Configuring the Si5324's registers can be done by either SPI or I^2C, however the only connection to it on the VC709 board is through its I^2C ports. Silicon Labs offers the software *DSPLLsim* (for free) to help generate the values for the registers for some chosen input frequency and desired output frequency. The quality of *DSPLLsim* is only mediocre in my opinion, as it will not allow a user to explicitly specify where the Si5324's reference clock should be taken from, i.e., the external crystal, or one of the differential pairs inputs. Only the latter ports can be chosen, so for to generate register values anyway, we can choose either and just set the input frequency to the reference crystal's.

With nothing at all connected to the input ports, the IC should use the crystal as its default input [17] so the choice of (wrong) input in *DSPLLsim* does not matter then, but the generated register values did not work. One probable cause is that *DSPLLsim* did not write the *Free-Run* bit in register (0), which is what makes the IC use the external crystal. I compared my values with some others that were posted on the Xilinx Forums to see that my *Free-Run* bit was not set (by *DSPLLsim*). A reply in the same thread replied the register values worked so I decided to copy all the values from the thread into the I^{2}C component's ROM. I kept the *DSPLLsim* version in the file as a comment. The Si5324 then correctly generated a 156.25 MHz signal so I did not bother so investigate the difference of register values in detail, e.g., if the denominator/nominator registers for the PLL division differed.

6.3 Project Development in Vivado

Close to the end of the thesis I discovered that my computer's setup caused my severe problems with Vivado. Almost all of my logic and components during its mapping and syntheses were removed and Vivado claimed this was because they were not used by anything else. The network file sharing (NFS) is most likely the cause why Vivado cannot run simulations either. The extreme case was when only LEDs were declared in a single project file and Vivado removed them because nothing was using them, but still synthesized something which makes no sense.

A voucher is included in the Connectivity Kit with a license for Vivado 2017 and the Virtex XC7VX690T FPGA on the VC709 board. I installed Vivado 2017 on my PC using that license and all the problems I had earlier disappeared.

Lastly about Vivado, all final code has been developed in Vivado 2017 projects, and projects cannot be downgraded to the 2016 version automatically. It is possible to open them in read-only mode and then save them as a copy, or something similar. I did not take any notes when I downgraded a project but it was successful.

6.3.1 Xilinx's Example Design

Using Vivado's Transceiver Wizard can be overwhelming to users who are not familiar to high-speed data links. With many settings of different acronyms, it can be difficult to be sure on what is really needed, what everything is or why you should change it. Most needed settings in the end is the selection of encoding scheme, the reference clock, a 64-bit bus width (maximum) and naturally, the serial speed to be 10.3125 Gbit/s.

It can take you a whole afternoon to try and find the example design for the Wizard generated IP component, if you do not already know that you must right click the IP file and click generate in the appeared menu. The design includes declarations of signals for ChipScope (Xilinx's debug software) and has one set for every GTH component which makes it hard to read.

The documentation for the transceiver wizard [8] states that to "use the example in hardware", one should connect the reset button on the board to the system. It is enough (and necessary) only with a reference clock, e.g., the Si5324, to start the system after the *.xdc* file with constraints have been changed accordingly, too. I am still lacking an explanation to why it was possible to toggle LEDs using the RX and TX clocks provided from the GTHE2_CHANNEL when the Si5324 was not used. The GTH transceivers absolutely need reference clocks and there are options for them in the Wizard, e.g., to use one PLL for four GHTs (QuadPLL).

One other way to enable the transceiver would be to manually assign all the 267 registers after careful reading of the data sheet and implement the reset components. There are much documentation from Xilinx that reference to other documents about small specific parts which is time consuming because of all acronyms and technologies mentioned.

7 Conclusion

This chapter summarizes the results and relates them to the goals that were set at the start of the project. There is possible further development of the resulting system which are discussed. Since the systems are portable, except for the circuits which adjust hardware specific to the VC709, some other development of similar systems on another FPGA board are mentioned.

7.1 Goal Fulfillment

This project's initial main goal was fulfilled as we now can use the Xilinx's VC709 board for fiber-optic communication and observe a system's status in real-time.

A total of nine systems with different error-correcting circuits (using BCH) were successfully run on the FPGA, reporting bit-error rates and the system's status to the Qt program on a PC.

It was possible to conduct a small experiment just using the OLA-54, but the results as far as error correction on the fiber-optic channel were not entirely consistent with our expectations. Attenuating a signal as much as in the experiment is not done anywhere outside a lab, at least to the best of my knowledge. If a signal would lose so much power, amplifiers would be a better solution rather than BCH circuits.

The largest 40 Gbit/s system is using 44.11 % of the available LUTs, so there are unused logic resources left for DSP circuits. Also, none of the FPGA DSP components, which are specialized just for this purpose, are utilized either. The FPGA on the VC709 might not be large enough to implement both an FEC system and a DSP system, which is why another FPGA board is suggested in Section 7.3.

An electrical interface for replacing an Avago transceiver (Section 3.6) has been acquired, i.e., a SFP module with four SMA cables for the TX and RX differential pairs. The electrical interface will allow a new experiment to connect the FPGA to other interfaces, e.g., VCSEL amplifiers. With the electrical interface, it will be possible to use other lab equipment with the system, and with the help from MC2 this can become reality in the future.

Lastly, the ASIC evaluation of the FEC circuits was never performed due to shortage of time. The implementation of the systems was prioritized and took longer time than expected to finish, and exceeded well beyond the time-plan.

7.2 Further Development

A good start for further development would be where this thesis ended, i.e., start to investigate more on why the experiment's results turned out like they did. Adding a component for buffering up words with erroneous bits can perhaps help to analyze the cause. The BCH circuits could cause a higher BER when enabled than disabled. Thus, the circuits were probably correcting the wrong bits, i.e., already correct bits, which can happen when more erroneous bits than their correcting capacity are received. Therefore, it is interesting to analyze the received vectors to see how the errors appear, e.g., if the errors come in a burst or are more spread out.

Now that an electrical interface has been acquired, I suggest to learn more about how to access the GTH's DRP ports. Section 3.3 showed some electrical parameters that can be changed with the IBERT example design which is using the DRP ports.

The systems that are using four Transceiver Modules are still not verified to work, which must be completed in order to achieve a 40 Gbit/s system.

All nine (10 Gbit/s) systems work even if Vivado reports timing warnings. There is a problem with connecting circuits together from two clock regions, i.e., Transceiver Modules from the RX clock region (156.25 MHz) and the UART from the main clock (200 MHz) region. I would attempt to multiplex several UARTs to get around the problem, or learn about elastic buffers, which are used in the GTH's between their different clock regions.

7.3 Other Development

The results can become useful for any future FPGA projects, e.g., on the Xilinx VCU110 [19] development board. The VCU110 has more logic than the VC709 and 28 Gbit/s transceivers. Thus, with some changes to the current systems, one would acquire more logic for other circuits and be able to transmit data faster.

The VCU110 uses GTY transceivers, so the GTHs must be replaced. The VCU110 is equipped with reference clocks from Silicon Labs so the FSM of the I2C can be reused with new register values. I suggest to first make one working system and then change the project-generating script to have all the other systems generated.

Xilinx transceivers support channel bonding, i.e., combining several SerDes data streams into one which could be usable for high-speed ADC ICs which use parallel SerDes for a high sampling frequency. With a high-speed ADC it could be possible to sample a signal modulated with pulse amplitude modulation (PAM). The FPGA could then also implement DSP circuits which connects to the FEC system from this thesis. The largest 40 Gbit/s system was shown to take up 44.11 % of the total FPGA area, so I believe there are still area left for DSPs.

Bibliography

[1] Xilinx. *7 Series FPGAs GTX/GTH Transceivers*. Available at `https://www.xilinx.com/support/documentation/user_guides/ug476_7Series_Transceivers.pdf`. (Visited on 06/12/2017).

[2] A. Athavale and C. Christensen. "High-speed serial I/O made simple". In: *Xilinx Inc* 4 (2005).

[3] P. Horowitz and W. Hill. *The art of electronics*. Cambridge University Press Cambridge, 2015.

[4] A. Brinton Cooper. *6.0 Decoding BCH and RS Codes*. Available at `https://www.ece.jhu.edu/~cooper/ERROR_CONTROL_CODING/06dec.pdf`. (Visited on 06/12/2017).

[5] Xilinx. *VC709 Evaluation Board for the Virtex-7 FPGA*. Available at `https://www.xilinx.com/support/documentation/boards_and_kits/vc709/ug887-vc709-eval-board-v7-fpga.pdf`. (Visited on 06/12/2017).

[6] Xilinx. *Vivado Design Suite User Guide*. Available at `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_3/ug893-vivado-ide.pdf`. (Visited on 06/12/2017).

[7] Xilinx. *7 Series FPGAs Data Sheet*. Available at `https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf`. (Visited on 06/12/2017).

[8] Xilinx. *LogiCORE IP 7 Series FPGAs Transceivers Wizard v3.3*. Available at `https://www.xilinx.com/support/documentation/ip_documentation/gtwizard/v3_6/pg168-gtwizard.pdf`. (Visited on 06/12/2017).

[9] Xilinx. *PicoBlaze 8-bit Embedded Microcontroller User Guide*. Available at `https://www.xilinx.com/support/documentation/white_papers/wp272.pdf`. (Visited on 08/23/2017).

[10] E. Kusse and J. Rabaey. "Low-energy embedded FPGA structures". In: *Low Power Electronics and Design, 1998. Proceedings. 1998 International Symposium on.* IEEE. 1998, pp. 155–160.

[11] A. Tanenbaum et al. "Computer networks, 4-th edition". In: *ed: Prentice Hall* (2003).

[12] WP Ranjula et al. "Implementation techniques for IEEE 802.3 ba 40Gbps Ethernet Physical Coding Sublayer (PCS)". In: *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2015 12th International Conference on.* IEEE. 2015, pp. 1–5.

[13] Texas Instruments. *LVDS Owner's Manual*. Available at `http://www.ti.com/lit/ml/snla187/snla187.pdf`. 2008. (Visited on 06/12/2017).

[14] Xilinx. *Virtex-7 XT VC709 Connectivity Kit*. Available at `https://www.xilinx.com/support/documentation/boards_and_kits/vc709/2014_3/ug966-v7-xt-connectivity-getting-started.pdf`. (Visited on 06/12/2017).

[15] Xilinx. *Get Smart About Reset: Think Local, Not Global*. Available at `https://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf`. (Visited on 08/23/2017).

[16] Silicon Labs. *SINGLE-CHIP USB TO UART BRIDGE*. Available at `https://www.silabs.com/documents/public/data-sheets/CP2103.pdf`. (Visited on 06/12/2017).

[17] Silicon Labs. *Any-Frequency Precision Clock Multiplier/ Jitter Attenuator*. Available at `https://www.silabs.com/Support%20Documents/TechnicalDocs/Si5324.pdf`. (Visited on 06/12/2017).

[18] JDSU. *OLA-54/55 SMART Optical Level Attenuator*. Available at `http://jdsu.fiberoptic.com/resources/SMART_ola55_ds_fop_tm_ae.pdf`. (Visited on 06/16/2017).

[19] Xilinx. *VCU110 Evaluation Board*. Available at `https://www.xilinx.com/support/documentation/boards_and_kits/vcu110/ug1073-vcu110-eval-bd.pdf`. (Visited on 06/12/2017).

A
Python Scripts

The Python scripts are made for generating any Vivado project which implements a transceiver system for any BCH circuit. It starts off with an old project which were tested to work but now has several files removed which are dependent on what kind of BCH it should use, and those files are generated by the project-generating script. It also makes a simulation project for each Vivado project.

project_generator.py The script which imports all the other scripts and generates new Vivado projects to separate folders. It starts off with searching a defined directory which contains all the BCH VHDL files. In that directory the sub-directories should be named, e.g.,

```
t1_k247_n255
t2_k493_n511
t3_k993_n1023
```

so the script can read out the *t*, *k* and *n*. It uses the values to name components such as the Compressors, Expanders and project names. After the project folder has been made and the reference project has been copied, the common files

```
BER_calculator.vhd
BER_circuit_64_bit_input.vhd
compressor_buffer.vhd
data_generator.vhd
exdes_TX_logic.vhd
FBERT.vhd
gtwizard_0_block_sync_sm_alt.vhd
gtwizard_0_descrambler_alt.vhd
gtwizard_0_scrambler_alt.vhd
i2c_master.vhd
reference_data_package.vhd
RX_synchronizer.vhd
TX_synchronizer.vhd
UART.vhd
VC709_I2C_inits.vhd
```

are copied over to the folder

```
/import_these
```

while the files for the simulation project are copied over to

```
/import_these_loopback_files
```

The script then iterates over all FECs it found and sequentially generates the following:

- Using tm_generator.py
  Calls *make_tm(k,n)*
  Returns the Transceiver Module
- Using bch_peterson_generator.py
  Calls *make_bch(k,n,t)*
  Returns a top module for the decoder.
- Using expander_generator.py twice
  Calls *exp_generator(IN_WIDTH, OUT_WIDTH, TX_VERSION)*
  Return one TX version and one RX version
  Recalling the difference between the two:

```
body += "if in_rdy = '1' and enable_in = '1' then"
   ... body code ...
if TX_VERSION:
  body += "--else\n" + tab4 + "--out_rdy_r <= '0';"
else:
  body += "else\n" + tab4 + "out_rdy_r <= '0';"
```

- Using compressor_generator_v2.py twice
  Calls *gen_compressor(IN_WIDTH, OUT_WIDTH)*
  Returns the compressors for TX/RX
- Using encoder_generator.py
  Calls *enc_generator(k,n)*
  Returns the encoder top module which is based upon the file which was generated with MATLAB. This version adds the error_ injection signal.
- Using exdes_top_generator.py
  Calls *(K_WIDTH, N_WIDTH, nr_of_transceivers)*
  Returns the top module. Here "exdes" is probably short for "example design", and the module is based upon Xilinx top module.
- Using transceiver_module_tb1_generator.py
  Calls *tb_generator(k,n)*
  Returns the testbench.

ber_circuit_generator.py This script will generate the circuit that compares two vectors and outputs the number of bits which differs.

At the top of the script the variable “IN_WIDTH = int(512)” should be changed to a desired integer and be a power of two. Generates files are named, e.g.,

```
BER_circuit_64_bit_input.vhd
```

word_expander_output_data_generator.py Uses a list which has its size defined by the variables *INPUT_WIDTH* and
INPUT_ROM_ROWS. The script iterates through a list of desired

OUTPUT_WIDTHS. For each *OUTPUT_WIDTH* all possible combinations of outputs an expander can generate are written to a file named, e.g.,

```
word_expander_package_for_D64_N1023.vhd
```

which then can be used for simulations with the test-bench.

synch_pattern_randomizer.py This script was used to generate the data which is sent out by the TX side at start-up. The data is used to tune the RX PLL and is designed to be close to the ideal sync pattern "10101010...", but the ideal pattern contains valid 64b/66b headers and is therefore randomized instead. The script however will not allow the same bit value appear more than three times in a row, and will invert those bits in order to get the "best" frequency of the pattern as possible without sending valid headers. At the end pf the script the vector is printer out to the terminal.

compressor_tb_generator_v2.py Contains a function *gen_compressor_tb(IN_WIDTH, OUT_WIDTH)* which simply generates a test-bench for Word Compressors.