Resource Sharing between SoCs through a Dedicated SerDes-channel

Casper Vikström, Kesheng Wang
ca5516vi-s@student.lu.se
ke2131wa-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Per Andersson, Joachim Rodrigues

Examiner: Pietro Andreani

June 13, 2024

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

In modern information technology there is an increase in demand for high-speed communication. Looking at the popular applications of AI and machine learning, a trend is to implement dedicated hardware accelerators for faster and more efficient computing. However this puts more of a demand on the ability to transfer the large amounts of data in the form of weights, input data and results to and from the accelerator.

This sort of system could be realized on a single SoC and a high-speed network with an increasing number of dedicated accelerators and components right on the chip, however two separate SoC leads to more flexibility with only the need to implement a high-speed communication, like the one suggested in this thesis.

In this thesis a dedicated high-speed SerDes channel was implemented and verified on two Digilent Nexys4 boards. The implemented physical channel connects the two separate Wishbone buses in their SoCs, allowing for read and writes across SoCs by the channel. The LiteX SoC environment was used as the base for the SoC and AMD Vivado primitives for the physical SerDes implementation. Along with this was the specially designed custom communication protocol, enabling the communication between the Wishbone buses.

Acknowledgement

We want to thank the EIT department for the opportunity of writing this thesis, a special thanks to our supervisors Joachim and Per which have both helped us getting started with the topic and also continued support through out the process. An extra special thanks goes out to Per for his invaluable tips and guidance that allowed us to always proceed in the right direction.

Popular Science Summary

The race for computing power that drives the advancements of popular AI and machine learning applications, such as LLMs like ChatGPT, often relies on dedicated hardware. These performance requirements for speed and efficiency emphasize the need for high-speed data transfers between different components. Whether it is hundreds of high-end GPUs performing inference or clusters with four dedicated accelerators, like those proposed by Yajie Wu et. al. [1], high-speed communication is crucial for making everything work efficiently. A popular high-speed serial link choice and the link used and implemented in this paper is a SerDes channel, which is commonly used in Gigabit Ethernet, PCIe, and other data transmission protocols.

High-speed serial data allows for the extension of an existing SoC by incorporating additional SoCs and their resources like larger memory. This enables the storage of weights or data outside the accelerator chip, significantly reducing cost and area, as on-chip memory (such as SRAM) tends to be more expensive and larger in area.

In this thesis an existing SoC structure was modified to include a custom communication protocol over a SerDes channel, allowing read and write operations to be issued on a receiving SoC. This demonstrates the capabilities of a dedicated high-speed SerDes communication channel for resource sharing. The work includes the creation of the physical SerDes channel along with the communication for the Wishbone bus connected to the SoC's CPU.

The proposed implementation proves that resource sharing is possible and promising but some improvements towards the reliability is left for future work.

Table of Contents

1 Introduction **1**
1.1 Related work . . . 1
1.2 Thesis outline . . . 1

2 Background **3**
2.1 Serial Data Communication . . . 3
2.2 Serializer and Deserializer . . . 4
2.3 Direct Memory Access . . . 6
2.4 LiteX Framework . . . 7
2.5 Clock Domain Crossing . . . 9

3 Design and Implementation **11**
3.1 Overall Architecture . . . 11
3.2 SerDes System Design . . . 11
3.3 SoC Bus Interface . . . 20
3.4 LiteX SoC Integration . . . 24
3.5 Hardware Implementation . . . 25

4 Verification and Results **29**
4.1 Verification Environment and Tools . . . 29
4.2 Prototyping Verification Results . . . 31

5 Discussion and Future Work **35**

References **37**

List of Figures

2.1 Example of single-ended and differential connection. 3
2.2 Demonstration of 10-bit bitslip operation. 6
2.3 Cache incoherence write. 7
2.4 Wishbone Master and Slave interfaces. 8
2.5 Wishbone single read and write cycle. 8

3.1 Block diagram of the communication setup with two independent SoCs. 12
3.2 Block diagram of the SerDes system. 12
3.3 Ports of the ISERDESE2 and OSERDESE2 primitives. 13
3.4 FSMD of the bitslip controller. 15
3.5 8b/10b encoder module diagram. 16
3.6 FSMD of the OSERDES controller. 17
3.7 FSMD of the ISERDES controller. 19
3.8 Differential input buffer primitives. 20
3.9 Differential output buffer primitives. 20
3.10 FSMD of the SerDes bus interface. 27
3.11 Asynchronous FIFO block diagram. 28
3.12 Signal channels between two FPGAs based on TMDS_33 standard. . 28

4.1 Overall experimental set-up. 30
4.2 Zoomed view of the differential input/output interface. 30
4.3 Waveform from LiteScope containing the signals of interest during a write operation. 31
4.4 Waveform from LiteScope containing the signals of interest during a read operation. 32
4.5 Waveform from LiteScope of the TX FIFO's response to a write operation. 32
4.6 Waveform from LiteScope showing the OSERDES Controller behaviour during a write operation. 33
4.7 Waveform from LiteScope showing the ISERDES Controller behaviour during a write operation. 33
4.8 Waveform from LiteScope showing the start of the bitslip process. . . 34
4.9 Waveform from LiteScope showing the idle behaviour of the bitslip controller. 34

4.10 Waveform from LiteScope of the RX FIFO's response to a write operation on the receiving SoC's side. 34

List of Tables

2.1 A sample of a few data characters from the 8b/10b scheme. 5
2.2 A sample of a few control characters from the 8b/10b scheme. . . . 5

3.1 Memory map of the SerDes's bus interface. 21

List of Acronyms

AI	Artificial Intelligence
CPU	Central Processing Unit
CDC	Clock domain crossing
CNN	Convolutional Neural Network
DC	Direct Current
DMA	Direct Memory Access
DDR	Double Data Rate
DRAM	Dynamic Random Access Memory
EMI	Electromagnetic Interference
FPGA	Field Programmable Gate Array
FSMD	Finite State Machine with Data path
FIFO	First-In-First-Out
IO	Input-Output
IP	Intellectual Property
LSB	Least Significant Bit
ML	Machine Learning
MSB	Most Significant Bit
PLL	Phase Locked Loop
RAM	Random Access Memory
SERDES	Serializer-Deserializer
SNR	Signal-to-Noise Ratio
SDR	Single Data Rate
SOC	System on Chip
UART	Universal Asynchronous Receiver-Transmitter

Chapter 1
Introduction

With the advancement of modern information technology, the demand for large-scale high-speed data computation and transfers has been increasing across various fields. For instance, in the popular fields of Machine Learning (ML) and Artificial Intelligence (AI), there is a significant requirement for both extensive data processing and high-speed data transfer. More specifically, substantial amounts of data need to be transferred to and from memory, in addition to the computational tasks involved. Nowadays, the utilization of external high-speed devices like hardware accelerators has become common to accelerate computations in these applications. However, the bottleneck often arises from the data transfers between chips. These challenges serve as the motivation for this work, which aims to research and investigate high-speed SerDes (Serializer-Deserializer) interfaces as a solution to the data intensive work.

1.1 Related work

SerDes interconnects are a well researched topic and are used in many places to achieve high bandwidth. Notably in gigabit Ethernet but also in niche areas more related to the specific application proposed in this thesis.

One example is the 2x2 Convolutional Neural Network (CNN) accelerator chiplet cluster purposed by Yajie Wu et al. [1]. Calculation tasks are split up between the four chips and the serial-link between them, enabling the data-transfer being a SerDes channel of 1 Gbps.

1.2 Thesis outline

The goal of this project is to develop a SerDes channel to achieve high-speed data transmission and hardware resource sharing between System on Chips (SoCs). This project is used as a prototype to demonstrate the concept using a Field Programmable Gate Array (FPGA) as a verification environment.

The thesis work is organized as follows:

- **Chapter 2: Background.** Introduces the reader to key concepts and components used in the design, as well as the working environment.

- **Chapter 3: Design and Implementation.** Describes the overall SerDes-channel architecture. Details the design methods of specific components and hardware implementation.
- **Chapter 4: Verification and Results.** Describes functional verification of each key component and displays the results.
- **Chapter 5: Discussion and Future Work.** Summarizes the general attributes of the SerDes-channel, discusses potential areas for improvement.

Chapter 2
Background

2.1 Serial Data Communication

Although parallel communication can send multi-bit signals in each clock cycle, due to the influence of clock skew in the system, the time difference between different bits arriving at the next level register during high-speed data transmission may cause multi-bit signals to be sampled incorrectly. Therefore, for high-speed data transfers serial data communication is often the preferred choice. As more bandwidth is required, the only solutions for parallel channels are increasing the clock frequency or bus width. This leads to more problems with the signal integrity, cross-talk between wires and reflections. For the serial stream even the clock can be embedded in one data line removing many problems with the parallel channels [2]. Serial gigabit transceivers achieve transfer speeds between 1 and 12 Gbps and also has no massive simultaneous switching output problems and brings electromagnetic interference improvements [3]. There also is multi-gigabit transceivers which utilize multiple serial links for achieving even higher bandwidth.

2.1.1 Differential signaling

In low-speed and short-distance data transfer, single-ended signaling is often chosen to transmit electrical signals over wires. By connecting a wire to a reference voltage (ground), only one wire is needed to carry the changing voltage that represents the data. In contrast, the method using two complementary voltage signals to transmit a single data signal is called differential signaling. In Figure 2.1 shows signal transmission on two different signal line connections [4].

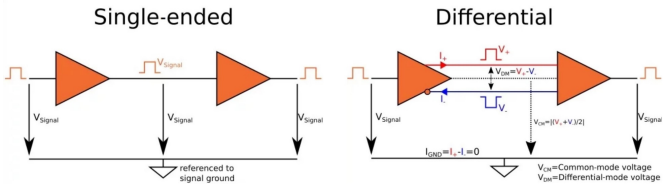

Figure 2.1: Example of single-ended and differential connection.

Even through single-ended signal is the simplest signal transmission method,

its performance in noise immunity is far inferior to differential signaling [5]. Since the transmitted information is picked up by detecting the difference between the two voltage signals at the receiving end, common-mode interference such as Electromagnetic Interference (EMI) and crosstalk that is equally added to the inverted and non-inverted signal lines can be effectively suppressed.

Compared with single-ended signals, the form of complementary signals and the improved noise suppression of differential signals allows it to maintain the adequate Signal-to-Noise Ratio (SNR) with a relatively lower voltage swing. Differential signals require smaller voltage transitions, which means lower power consumption, reduced electromagnetic interference, improved signal integrity, and support for higher signal transmission rates, thereby providing more reliable and efficient signal transmission in high-speed and high-density communication systems [6].

2.1.2 Differential I/O on FPGA

Differential signaling is used in high-speed serial links because of the reduction in common mode noise as well as even order distortion terms. Smaller voltage swings are also used in order to lower the dynamic switching power [7]. Two popular Input-Output (IO) standards that has differential signals include, LVDS_25 (Low-voltage differential signaling 2.5V) and TMDS_33 (Transition minimized differential signaling 3.3V).

2.2 Serializer and Deserializer

A general SerDes consists of a few blocks which will be explained in more detail in the following sections but first a brief overview and general structure and how these blocks connect to form the complete SerDes.

For the transmitter encoding of the signal is firstly done, this is through the 8b/10b encoder. After this encoder the data is now 10 bits wide and is sent as parallel data to the serializer. The parallel data of 10 bits is serialized and sent over the serial link at a clock frequency of 10 times the system frequency. Therefore each system-clock edge a full packet has been sent. This clock frequency is fed through a Phase Locked Loop (PLL) and if Double Data Rate (DDR) is selected then that faster clock need only be 5 times faster than the system.

For the receiver it is similar but the flow is reversed. Another aspect that needs to be handled is also the synchronization between the receiver and transmitter. So firstly is the deserialized which parallelized the serial signal into ten bits, it is also driven by both the fast clock and the system clock. The next block is logic to check after some known pattern and if it is not found, try to re-align the transceiver. This is done through the bitslip module and controller. Then is the decoder which decodes the 10 bits to the original 8.

2.2.1 8b/10b encoding

8b/10b encoding is a scheme invented at IBM in 1983 and is used in applications such as gigabit Ethernet. The scheme encodes each 8-bit number to 10-bit according to a algorithm and assures proper Direct Current (DC) balance in the

transmission lines. Each 8 bit value has both a positive and negative value, meaning either one more 1 (+) or one more 0 (-). This is possible due to the extra two bits leaving room to select only some of the 10-bit characters which have more balanced composition of 1's and 0's. The encoding also makes sure that there are sufficient transitions of the signal to accurately recover the clock from the data line.

There are two types of characters, data and control characters. The data characters are the data you want to encode and an example of a few of those characters can be seen in Table 2.1. There are 24 control characters and 3 of them are so called commas, special characters that can be used for phase aligning between transmitter and receiver, these are K28.1, 28.5 and 28.7 and can be seen in Table 2.2. They work well for aligning because the sequence of their 7 first digits wont appear in any other combination of 10b characters, meaning there will not be any false positives while looking for the correct sequence [3].

Table 2.1: A sample of a few data characters from the 8b/10b scheme.

Character	8 Bits	10b positive	10b negative
D0.0	00000000	1001110100	0110001011
D1.0	00000001	0111010100	1000101011
D31.1	00111111	1010111001	0101001001

Table 2.2: A sample of a few control characters from the 8b/10b scheme.

Character	8 Bits	10b positive	10b negative
K28.1	00111100	0011111001	1100000110
K28.5	10111100	0011111010	1100000101
K28.7	11111100	0011111000	1100000111

2.2.2 Transceiver phase alignment

The receiver and transmitter of the SerDes are located on two different FPGAs, the clock signal generated by different oscillator have slight difference even through their frequencies are same, which will make the transceiver drift out of phase gradually. This means that re-alignment is required within a constant interval.

The bitslip operation is used to phase align transmitter and receiver. It is a built-in module in the ISERDESE2 primitive from AMD Vivado. When a bitslip is issued, the Most Significant Bit (MSB) of the parallelized data will be dropped and the Least Significant Bit (LSB) will be resampled. It looks like the parallelized data is only shifted around but in reality this resampling takes place. If the data is of n bits there will only be n useful bitslip operations until the signal wraps around

and just give the same patterns again. In Figure 2.2 shows that an example of a 10-bit bitslip operation.

Bitslip Iteration	10b Pattern
0	0011111010
1	0111110100
2	1111101000
3	1111010001
4	1110100011
5	1101000111
6	1010001111
7	0100011111
8	1000111110
9	0001111101

Figure 2.2: Demonstration of 10-bit bitslip operation.

2.3 Direct Memory Access

Direct Memory Access (DMA) is a method of data transfer that allows an I/O device to directly access the main memory, bypassing the Central Processing Unit (CPU) of a system. Traditionally, data transfers between I/O devices and main memory, or between different areas of main memory, require the involvement of the CPU. Every data need to be passed through CPU which makes CPU be totally occupied for the entire duration of the read or write operation. It can consume CPU processing time, and potentially limit data transfer speeds. Therefore, DMA is powerful for reducing CPU overhead, especially in those situations that require large amounts of data exchange.

2.3.1 DMA controller

DMA is typically managed by a DMA controller, which usually contains a memory address register, a byte count register, and other control registers. Generally, to carry out a DMA data transfer operation, the CPU of a system initializes the DMA controller with a count of the number of words to transfer, and the memory address to begin. The DMA controller then informs the external device to prepare for a data transfer and generates addresses and read/write control lines to the system main memory. Each time a byte of data is ready to be transferred between the external device and memory, the DMA controller increments its internal address register until the full block of data is transferred. Once a data transfer is completed, the DMA controller often sends an interrupt signal to the CPU to notify it of the completion.

2.3.2 Cache coherency

However, DMA can cause cache coherency issues on systems with caches. With a cache as a buffer, the CPU will not directly fetch or modify data in the main memory but the cache, so that future requests for some data can be served faster. In Figure 2.3 an example of the cache coherency issue is shown. If the CPU accesses the location of X in the memory, the X will be stored in the cache and subsequent operations will only involve this copy of X. Once external devices write a new value Y to the same location in memory via DMA, then the value at that location in memory and cache will be inconsistent. The same issue will occur for external devices read operations.

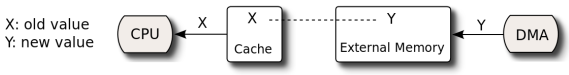

Figure 2.3: Cache incoherence write.

To address this issue, the system can include hardware to snoop the DMA controller. If a write/read operation occurs on the DMA controller, the cache controller is alerted to perform a cache invalidation or cache flushing. This can also be done in software, but will introduce some overhead to the DMA operation.

2.4 LiteX Framework

LiteX is a tool used for high-level-synthesis written in Python. It is built upon the library Migen and offers more extended tools for building SoCs. These tools include buses, interfaces, building blocks such as Random Access Memory (RAM), First-In-First-Out (FIFO), Universal Asynchronous Receiver-Transmitter (UART) and CPU cores. LiteX also supports different hardware languages for integration if an already existing module wants to be instantiated.

The LiteX tool has a complete environment where boot loader code is generated for the SoC and Vivado can be sampled and used right in the terminal for bitstream generation and loading the bitstream on to the FPGA.

LiteX also includes packages for compiling C code using a cross compiler. This study uses C program to drive the dedicated SerDes channel for functionality verification. The binary file generated by compilation can then easily be loaded to the CPU later using the litex_term which is their program acting as a serial monitor. Through this serial monitor it is also possible to read and write to memory locations, run the compiled C code, and in different ways interact with the SoC when it is running [8].

2.4.1 Wishbone bus

One of the key components of a LiteX SoC is the bus system, which is used to connect the various components of a SoC. The Wishbone bus is a standardized, open-source bus system that is widely used in FPGA-based SoCs such as LiteX SoCs. Wishbone bus is intended as a "logic bus", focusing primarily on the logical

aspects of communication between components of a SoC. It defines a set of logical signals and their timing requirements in terms of clock cycles. In Figure 2.4 an example of Wishbone Master and Slave interfaces signals is shown. By not prescribing the electrical characteristics or physical implementation details, Wishbone bus allows for greater flexibility and adaptability across different implementations and technologies, which greatly facilitates the reuse and integration of various Intellectual Property (IP) cores [9].

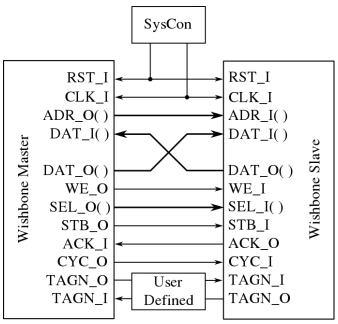

Figure 2.4: Wishbone Master and Slave interfaces.

So far as the interfacing through the Wishbone bus is concerned, the on-chip Wishbone bus architectures can be classified as: Point-to-point interconnection, Data flow interconnection, Shared bus interconnection and Crossbar switch interconnection. Specifically, SoCs under the LiteX framework adopt Crossbar switch interconnection for more flexible connection and communication. With more than one Master in a system, Wishbone bus requires bus arbiters to determine when each Master may gain access to the indicated Slave, but devices still maintain the same interface as the example in Figure 2.4.

In Figure 2.5 show Wishbone single read and write timing diagrams [9]. A Master interfaces initiate a transfer cycle by asserting [CYC_O]. Communication between the Master and Slave is based on a handshaking protocol. A MASTER asserts [STB_O] when preparing for data transfer and it remains asserted until the Slave asserts one of the cycle terminating signals [ACK_I], [ERR_I] or [RTY_I].

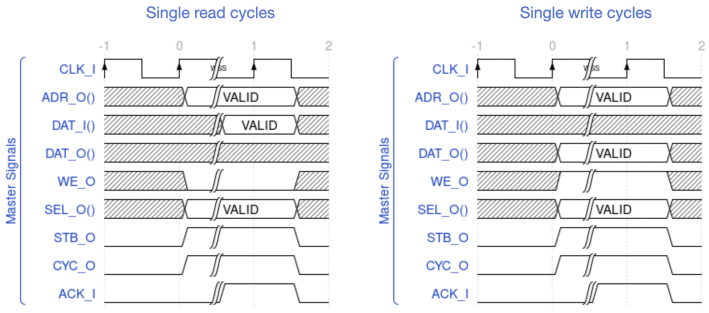

Figure 2.5: Wishbone single read and write cycle.

2.5 Clock Domain Crossing

Clock domain crossing (CDC) is an important concept in digital system design. A clock domain is a group of flip-flops or other sequential elements that are clocked by the same clock signal. When these elements need to communicate with elements in another clock domain, the transition of data across the boundary between these domains is termed a clock domain crossing, which require additional operations to ensure a successful crossing. When a digital system uses multiple clock signals with different frequencies or phases, this system is called a multi-clock system or an asynchronous system. As digital circuits become more complex, the need for effective CDC mechanisms becomes increasingly important to ensure data integrity and reliable communication between different clock domains.

The primary challenges associated with CDC are listed as follows:

- **Data Integrity.** Ensuring that the data transferred across clock domains remains accurate and consistent is crucial, especially for multi-bit data. This requires careful design techniques to handle potential timing issues and synchronization errors.
- **Metastability.** When a signal crosses from one clock domain to another, due to differences in clock frequency and phase, the sampling clock edge may meet an uncertain state of signal where a signal logic inversion occurs, leading to metastability. Metastability can cause unpredictable behavior in the receiving domain, resulting in data corruption or timing errors.
- **Timing Uncertainty.** Different clock domains may operate at different frequencies or phases, leading to uncertainty in timing relationships. This can make it difficult to ensure that data is correctly latched or interpreted by the receiving clock domain.

To address these challenges, several techniques are commonly employed in digital design:

- **Synchronizer.** Two-Flip-Flop Synchronizer: A simple yet effective method involves using two flip-flops in series in the receiving clock domain. This reduces the likelihood of metastability by allowing the first flip-flop to capture any indeterminate state and the second to stabilize it. Multi-Stage Synchronizers: For higher reliability, more than two flip-flops can be used in series. This further reduces the probability of metastability but increases latency
- **FIFO Buffers.** Asynchronous FIFOs: These are used to manage data transfer between clock domains operating at different frequencies. An asynchronous FIFO uses separate read and write pointers managed by the respective clock domains, along with flags to indicate empty and full conditions.
- **Handshaking Protocols.** Request-Acknowledge Protocols: Handshaking protocols involve explicit signals to request data transfer and acknowledge receipt. This ensures controlled data transfer and can help manage timing differences between clock domains.

- **Dual-Port RAM.** Shared Memory: Dual-port RAM allows both clock domains to read and write to a shared memory space. Access control mechanisms, such as semaphores, can be used to manage concurrent access and avoid data corruption.

Chapter 3
Design and Implementation

The design consists of a setup of two FPGAs designed to emulate the environment and interaction between two individual chips. This setup enables resource sharing between the two FPGAs and their respective SoC. In this work one SoC features a RISC-V core, with room for a potential accelerator and a few I/O pins. The other, more comprehensive SoC is configured with the same RISC-V core and other integrated peripherals such as Dynamic Random Access Memory (DRAM), Ethernet as well as additional components.

This design environment enables the accelerator to execute on one of the SoCs, while the other SoC is designed to be a more complete SoC environment. Resource sharing plays a crucial part in this project and with the design and implementation of SerDes channels the two systems can complement each others strengths. Notably the implemented SerDes channel facilitates direct communication and resource sharing between the two FPGAs, allowing them to read and write to each other's peripherals and components. Especially relevant for the potential of this work would be the accelerator's capability to access the data stored in the larger DRAM hosted on the main SoC.

3.1 Overall Architecture

Figure 3.1 below shows the overall high-level structure and architecture of the project. The focus will be on implementing the SerDes communication link along with its custom communication protocol based on the DMA of the wishbone bus. The system bus interface and SerDes system and its submodules in this study can be seen in Figure 3.2. The following sections will introduce the design and implementation of these modules in detail.

3.2 SerDes System Design

In this section the complete SerDes Systems design and implementation along with its components will be introduced and elaborated on.

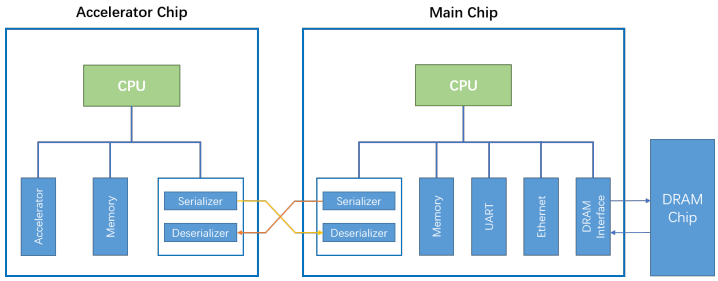

Figure 3.1: Block diagram of the communication setup with two independent SoCs.

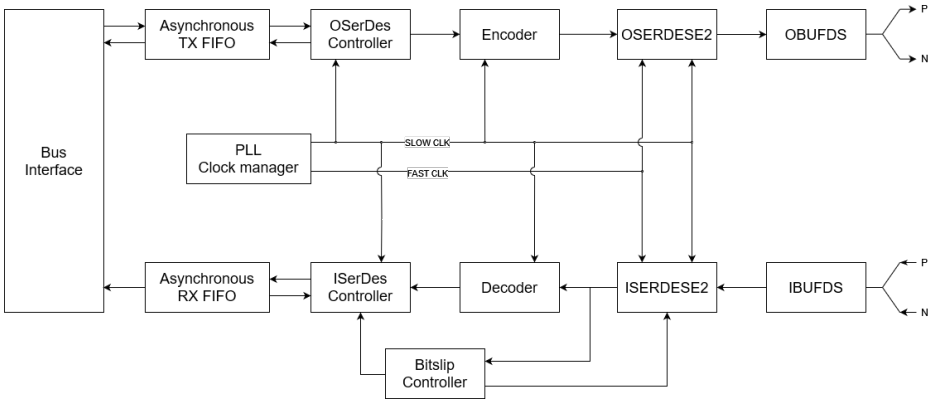

Figure 3.2: Block diagram of the SerDes system.

3.2.1 I/OSERDESE2 configuration

It was firstly expected that this project would implement a gigabit transceiver SerDes channel using the LiteX liteiclink pre-built function called "GTX" but as it was later realized the digilent Nexys4 board did not have the required resources for a gigabit transceiver channel. Therefore it was necessary to pivot in to using a more basic SerDes structure that could be supported by the hardware at hand.

This solution is using AMD Vivado primitives ISERDESE2 and OSERDESE2. These two circuits port definitions can be seen in figure 3.3. A fundamental difference between the UART and SerDes is that the SerDes is always transmitted something. It is up to the receiver to determine if it is useful data. For that we used the previously mentioned 8b/10b encoding scheme to decide on a start and stop control character. These two control characters indicates the data in between them are useful data that should be considered and used by the receiving side.

The UART phase aligns on each transaction of a byte. When a byte is transferred, a start bit transitioning from high to low is sent and a stop bit asserted high at the end of the byte transfer. This means that the UART is kept synchronized at all times. For the SerDes however it always sends data, and needs to be phase aligned using the bitslip like it was mentioned earlier. This bitslip module

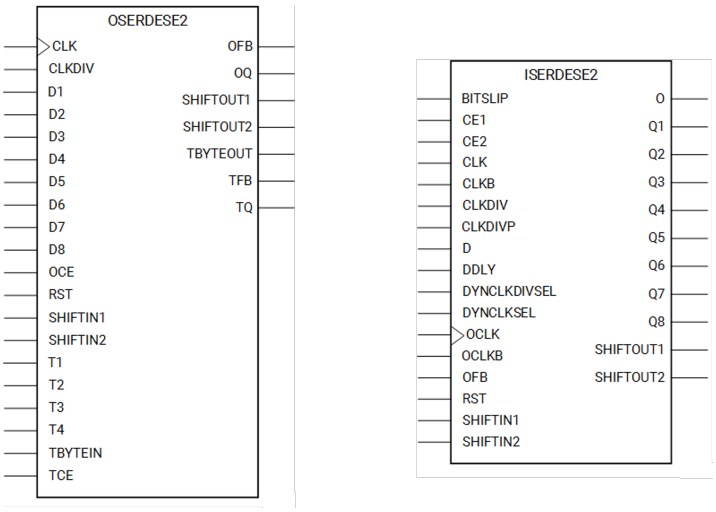

Figure 3.3: Ports of the ISERDESE2 and OSERDESE2 primitives.

is controlled via the bitslip pin on the ISERDESE2 and the controls were implemented according to the Architecture Manual AM002 [10]. Bitslip enable is held high for one slow clock cycle and after performing the operation, the control logic to check if the wanted correct sequence has appeared and the phase alignment is done needs waits for 3 clock cycles after issuing the bitslip again.

Data-rate is set to DDR. Meaning double data rate, so the data is sent for both the positive and negative edge of the fast clock. This means that the generated high-speed clock of the SerDes requires only half of the frequency it would in Single Data Rate (SDR) mode.

The clock-ports on the ISERDES and OSERDES are CLKDIV and CLK. CLKDIV is the lower frequency clock, setting the baseline of the frequency at 7.5MHz. The high-speed clock CLK has 5 times higher frequency than CLKDIV because of the use of DDR and the width of the sent/received data being 10 bits wide means that the frequency of the transmission line is 75 MHz.

The native width of the SerDes primitives are 8 bits wide but it is not unusual that wider data than that has to be transmitted. Therefore the primitives has a Master-Slave configuration which allows for cascading of two primitives. This configuration has some limits and the allowed bit widths are 10 and 14. To properly set up the cascaded SerDes configuration, one SerDes was set to Master with it's corresponding data Q1-Q8 concatenated to the final result and the Slaves data-ports Q3-Q4 connected to the final two MSB. For the ISERDES the SHIFTOUT from the master is connected to the SHIFTIN of the slave, and for the OSERDES the SHIFTOUT of the slave is connected to the SHIFTIN of the master.

While developing this circuit the OFB port of the OSERDESE2 was used instead of the physical connection in order to take the development step by step. The OFB port is useful for testing the transceiver since it is an internal connection and is therefore more reliable than the unpredictable characteristics of the Pmod ports, also the custom made connection was less known earlier in the process. The

OFB connects the ISERDES and OSERDES together, enabling data-transfer. To configure the circuit for the use of OFB the parameter OFB_USED must be tied high and the masters OFBs, connected. With this configuration the received data of the ISERDES was tested and confirmed.

3.2.2 PLL configuration

The PLL used in this project was the AMD Vivado PLLE2_BASE primitive. The primitive has pre-defined limits that has to be kept, limits such as the input reference frequency can not be lower than 15 MHz and the global division and multiplication factors used to derive the different output clocks must yield a result in the range of 800 MHz-2000 MHz. To achieve each of the different generated clock signals, the individual division and multiplication factors are applied and through them a good range of different clock frequencies can be achieved. In this implementation the clocks wanted were the 1x clock, 5x clock and the inverted 1x clock. After applying the multiplication and division factors, the clocks were then slowed down further with 3 clock dividers in-order to satisfy our limit of maximum output frequency over the serialized channel at 75 MHz.

3.2.3 Bitslip operation

Without the bitslip operation it was seen that the channel would in fact be out-of-sync most of the time and seldom receive the correct data. As explained in Section 2.2.1 the bitslip is used to synchronize the ISERDESE2 and OSERDESE2 so that the receiver samples the serial bitstream and outputs them in parallel and the correct order, not for example the 5 LSB of the previous character and the 5 MSB of the current character.

The problem is that the SerDes always sends and receives data and if the clocks differ, their phases will drift. The implemented solution is a bitslip controller. This state machine is on the receiving side, looking for the training pattern (K28.7). This Finite State Machine with Data path (FSMD) is shown in Figure 3.4. It will start of in the initial state waiting for 16 clock cycles until it goes to the bitslip state where it will issue a bitslip and then wait for 3 clock cycles before checking for the correct training pattern. It repeats this process until the correct sequence is found and then it goes in to idle and waits for the ISERDES to drift out phase, meaning one bit has shifted in the parallel data, it then starts realigning it again with the same process. When the transmitter is not sending data it always sends out the training pattern so that the bitslip controller can detect the shifting and ensure that it is aligned.

The time between each synchronization depends on the clock frequency difference of the two boards. For the two boards used in this project about 75000 clock cycles passed before a new bitslip operation was needed. A problem that arises from our bitslip controller functionality is that there is a possibility that the ISERDESE2 is bitsliping at the same time that the sender wants to transmit data. There is no bi-directional communication so there is no way for the transmitter to know if the data was lost.

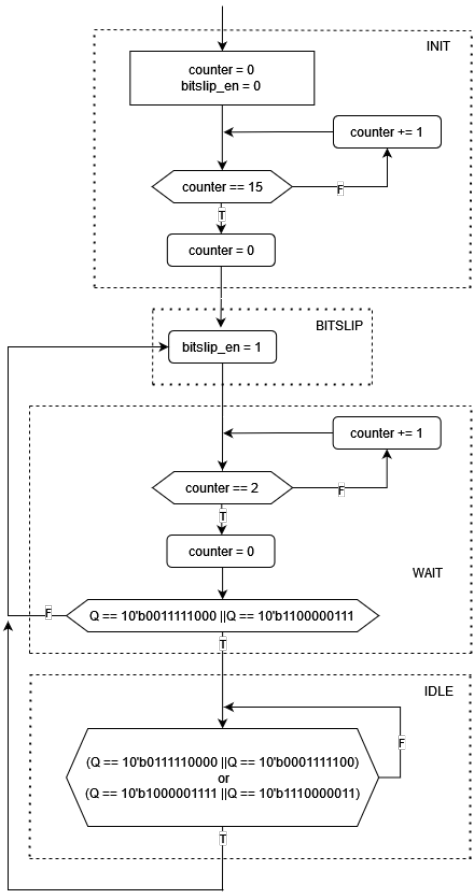

Figure 3.4: FSMD of the bitslip controller.

3.2.4 8b/10b encoder and 10b/8b decoder

8b/10b encoding is a method used for high-speed serial data transmission, primarily aimed at achieving DC balance and reducing error rates. The basic principle is to encode each 8-bit input data into 10-bit output data by controlling the number of 1 and 0 to achieve DC balance and by limiting the number of consecutive identical characters/symbols to reduce error rates. This SerDes system uses the 8b/10b encoding scheme developed by IBM.

The encoding process consists of two main stages: 5b/6b encoding and 3b/4b encoding, and disparity control. In Figure 3.5 shows the 8b/10b encoder module diagram and the relevant inputs and outputs[11]. The 8b/10b encoder module pre-defines the lookup tables for 5b/6b and 3b/4b encoding. The incoming 8-bit symbols is divided into two portion. The low 5 bits and the top 3 bits of data are encoded into a 6-bit group (the 5b/6b portion) and a 4-bit group (the 3b/4b portion) separately. If the current byte is a control character (K) and the low 5 bits are equal to 28 (decimal), a special control character code '110000'

is generated. Otherwise, regular encoding (D) is performed through the lookup table. The balance of the encoding result is calculated as a reference for disparity control. 3b/4b encoding is similar to 5b/6b encoding.

8b/10b coding is DC-free, meaning that the long-term ratio of ones and zeros transmitted is exactly 50%. To achieve this, the difference between the number of ones transmitted and the number of zeros transmitted is always limited to plus or minus 2. This difference is known as the running disparity (RD). The 5b/6b code is a paired disparity code, and so is the 3b/4b code. Each 6- or 4-bit code either has an equal number of zeros and ones (a disparity of zero), or comes in a pair of forms, one with two more zeros than ones and one with two less (non-zero disparity). When a 6- or 4-bit code with a non-zero disparity is used, the disparity control submodule must choose between positive and negative disparity encoding in a way that switches the running disparity. This means that codes with non-zero disparity should alternate to maintain DC balance. Finally, these two code groups are concatenated together to form the 10-bit symbol that is transmitted on the wire. Disparity In and Disparity Out are used for cascading use cases.

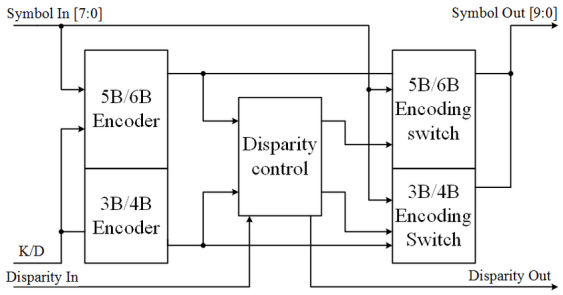

Figure 3.5: 8b/10b encoder module diagram.

The 10b/8b decoder converts 10-bit encoded input symbols back into 8-bit data and a control signal (K), while also detecting invalid symbols. The process involves reversing the 5b/6b and 3b/4b encoding stages used during the encoding process. Similarly, the 10b/8b decoder divides the input 10-bit symbols into two groups, one of 6 bits and one of 4 bits, and decodes the two groups of data using the pre-stored lookup table. Control characters are identified by specific 6-bit codes ('001111' and '110000'). If these codes are detected, the K signal is set, and the corresponding 3-bit decoded value is retrieved from the control character tables. Invalid symbol detection is accomplished by counting the number of 1s in the input 10-bit symbol. A valid symbol should have exactly 4, 5, or 6 ones. If the count is outside this range, the invalid signal is set, indicating that the symbol is invalid.

3.2.5 OSERDES controller

In high-speed SerDes systems, the OSERDES controller is a critical module responsible for coordinating and controlling the timing and content of data transmission. In this SerDes system design, the OSERDES controller works closely

with the 8b/10b encoder and the OSERDESE2 primitive. While the OSERDESE2 primitive handles the actual parallel-to-serial conversion, the OSERDES controller manages and controls various signals and modes during data transmission.

The OSERDES controller receives parallel data from the CPU, including commands, addresses, and data. It schedules the transmission of this data at appropriate times, ensuring the orderly and correct flow of data throughout the system. The OSERDES controller determines when the OSERDESE2 primitive should transmit different types of data from the CPU side and when the OSERDESE2 primitive is idle and sends idle data. Between these two modules is the 8b/10b encoder, which only encodes the input parallel data and generates a delay of one to two clock cycles, and does not actually control the OSERDESE2 primitive. The OSERDES controller is also responsible for implementing the customized communication protocol in this SerDes system.

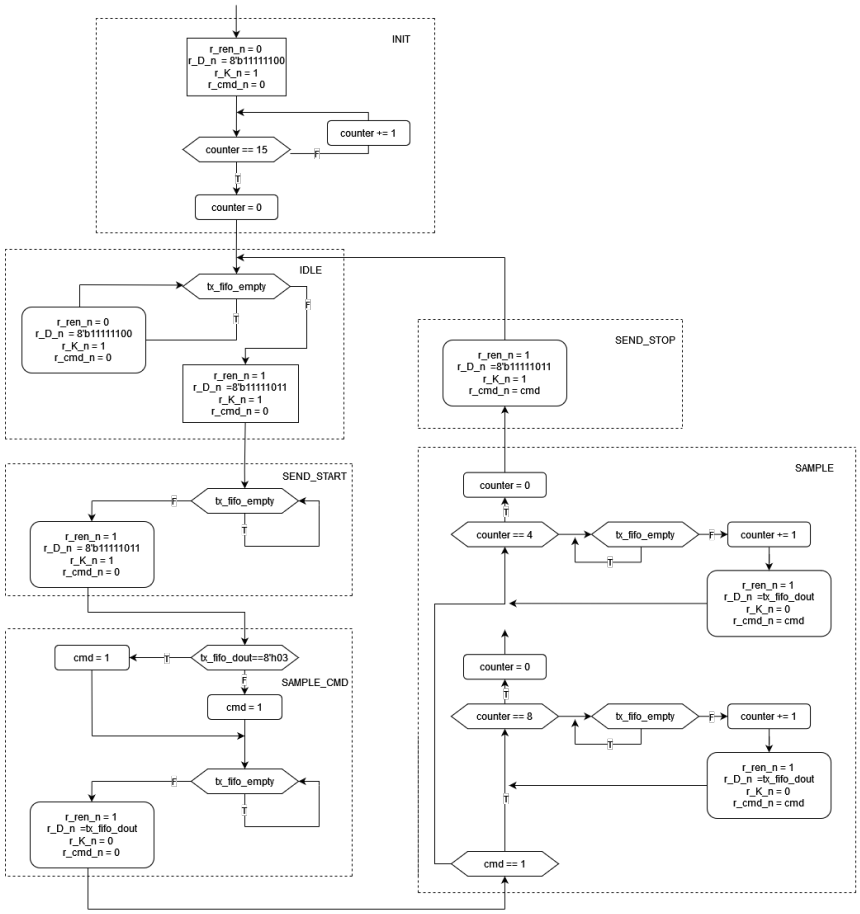

Figure 3.6: FSMD of the OSERDES controller.

Figure 3.6 shows the FSDM of the OSERDES controller. After waiting for 15 clock cycles in the INIT state to ensure that the system is stable, the state

machine will enter the IDLE state. Training pattern (K28.7) is used during high-speed data transmission to calibrate the link and ensure reliable data transfer. Therefore in the IDLE state, the OSERDES controller expects the OSERDESE2 primitive to always send the training pattern. At the same time, the 8b/10b encoder needs to be informed that the data to be encoded is a control character rather than a data character. The OSERDES controller will continue to send the K28.7 character and the asserted control flag K to the 8b/10b encoder until it detects that the previous stage has available data to send. Subsequently, the state machine enters the SEND_START state and sends a special control character (K27.7) as a start control character for a new frame to be sent, which is also part of the protocol in this SerDes system. At the same time, a read enable signal is asserted so that the data sent by the previous stage can be sampled in the next cycle. After entering the SAMPLE_CMD state, it detects the first byte sampled from the previous stage, determining whether the CPU initiated a read operation or a write operation, and forwards this data character to the encoder. In the next clock cycle, the state machine will enter the SAMPLE state and decide how many data characters should be sampled in this state based on the previous read/write command judgment. Whenever the data valid flag of the previous stage does not exist, counting and sampling are stopped, and waiting for valid data to enter again. During this period, the read enable signal is always asserted. When the counter reaches the threshold, it will be cleared and the state machine will enter the next state SENT_STOP. The OSERDES controller will send a control character according to the customized protocol to indicate that the frame transmission has ended. Finally, the state machine returns to the IDLE state to wait for the next transmission.

In summary, the OSERDES controller completes the construction and transmission of a data frame, which is an important part of the implementation of the customized protocol in the SerDes system.

3.2.6 ISERDES controller

In high-speed SerDes systems, the ISERDES controller is essential for managing the deserialization of incoming serial data. It works in conjunction with the ISERDESE2 primitive and 10b/8b decoder in the SerDes system design. While the ISERDESE2 primitive handles the actual serial-to-parallel conversion, the ISERDES controller will identify the received parallel data and decide what data to forward to the next stage, usually the bus interface, based on the customized protocol.

Figure 3.7 shows the FSMD of the ISERDES controller. The controller samples the characters and the control flag K output by the decoder in each clock cycle. In the IDLE state, the state machine will check whether the received parallel data is the start control character (K27.7) of a new data frame according to the K and the 8-bit data. At the same time, in order to prevent false triggering caused by misalignment of the transmitting and receiving ends due to channel drift, it is necessary to read the state of the bitslip module while detecting the frame start. Only when the bitslip module is in the IDLE or CORRECT state can the ISERDES controller believe that the channel is aligned and that the decoded

characters received are valid. If the trigger condition is met and the state machine enters the DATA state, the controller will forward the sampled 8-bit characters to the subsequent stages for further processing or transmission, and assert a flag. Here, it is also necessary to determine when the received data frame stops, signified by the stop control character (K28.4). If it is a stop control character, the valid flag of the data sent to the subsequent stage will be immediately cancelled to prevent the stop mark from being passed to the subsequent stages. After a data frame is transmitted, the controller returns to the IDLE state and waits for next data frame.

In short, the ISERDES controller mainly realizes the recognition of the start and stop control characters of the data frame, and passes the valid data characters between these two control characters to subsequent stages.

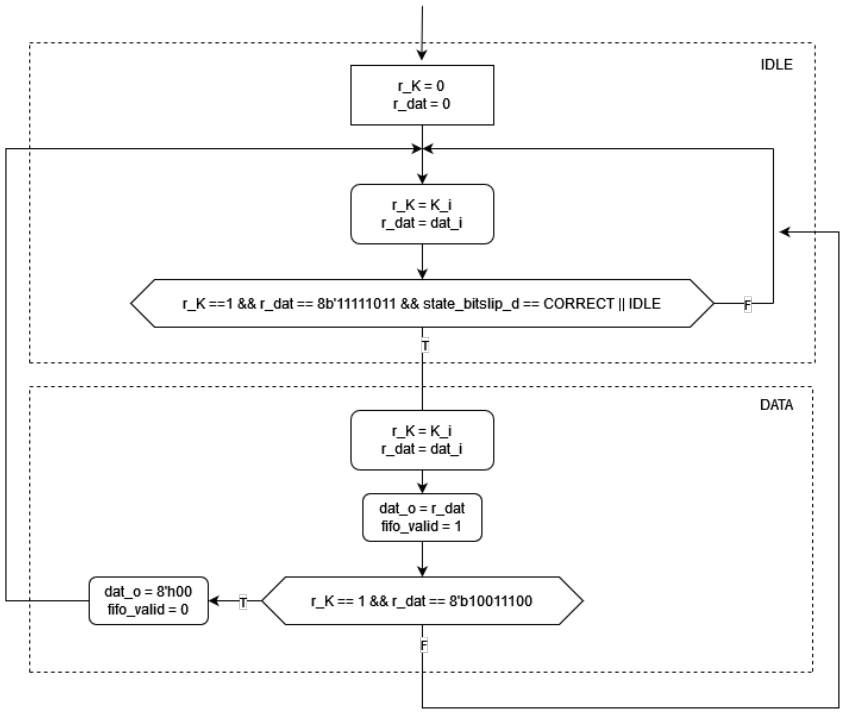

Figure 3.7: FSMD of the ISERDES controller.

3.2.7 IBUFDS and OBUFDS primitives

Considering that this study is based on FPGA to verify the high-speed data transmission between two SoCs, it is necessary to convert the single-ended signal into a differential signal according to Section 2.1.1 on the transmitter FPGA and convert the differential signal back to a single-ended signal on the receiver FPGA for subsequent processing. The usage and rules corresponding to the differential primitives are similar to the single-ended SelectIO primitives. Differential SelectIO

primitives have two pins to and from the device pads to show the P and N channel pins in a differential pair.

Differential signals used as inputs to 7 series devices use an input buffer. The generic 7 series FPGA IBUFDS and IBUFGDS primitive is shown in Figure3.8. The IBUFDS and IBUFGDS primitives are the same, IBUFGDS is used when an differential input buffer is used as a clock input. In this SerDes system, the high-speed differential signals received from the input pads on FPGA is converted into a single-ended signal through IBUFDS and then fed into the ISERDESE2 primitive.

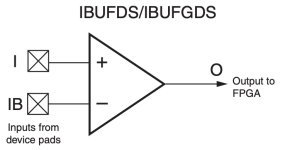

Figure 3.8: Differential input buffer primitives.

Similarly, an differential output buffer (OBUFDS) is used to drive signals from 7 series FPGA devices to external differential pair pads [12]. A OBUFDS primitive is shown in Figure 3.9. The single-ended signal OQ output by the OSERDESE2 primitive is converted into a differential signal through OBUFDS.

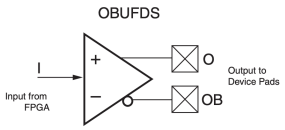

Figure 3.9: Differential output buffer primitives.

3.3 SoC Bus Interface

The high-speed SerDes system in this study is designed to achieve high-speed data transmission and resource sharing between SoCs. Therefore, in addition to establishing communication between a pair of high-speed SerDes channels, defining their dedicated protocols, and building related functional modules, it is also necessary and important to pay attention to the data transmission and control relationship between the SerDes system as a submodule of a SoC and other modules of the system, including CPU, main memory and other peripherals.

In the SoCs under the LiteX framework, all system modules are connected using Wishbone bus. The structural features and classic interfaces of Wishbone bus have been introduced in Section 2.4.1 and will not be repeated here. As a system bus, Wishbone bus protocol defines the communication participants as "Master" and "Slave", and the data transferring follows the master/slave mode. The master device, as the source of instructions, is responsible for allocating work and controlling data flow. The slave device is responsible for completing the work and can

generally only communicate with the master device. This section will discuss how the SerDes system communicates with other bus devices in two different identities separately.

3.3.1 Bus slave interface

In the application scenario envisioned in this study, one SoC hopes to access the main memory on the bus of another external SoC by initiating a basic local read or write. Here, we call the former "active SoC" and the latter "passive SoC". In an active SoC, the SerDes system needs to receive read and write requests issued by the local CPU through the bus, perform the forwarding of relevant cross-SoC read and write instructions, and give the local CPU a response that conforms to Wishbone bus protocol. Therefore, a bus slave device interface is required between this SerDes system and Wishbone bus to handle this communication process.

Table 3.1: Memory map of the SerDes's bus interface.

Memory	Width	Description
memory[0]	8-bit	Command byte: 0x03-write, 0x04-read
memory[1]	8-bit	Length byte: default is 0x01
memory[2:5]	32-bit	Address word to be sent
memory[6:9]	32-bit	Data word to be sent
memory[10:13]	32-bit	Temporarily store received word

In addition to the interface signals consistent with those described in Section 2.4.1, this bus slave interface also has some local memories as shown in Table 3.1. Part of these memories is used to record the information of a read/write operation initiated by the CPU, and the other part is used to temporarily store data read back from the external SoC. Once the local CPU wants to access the memory of the external SoC, it will achieves this indirectly by accessing this SerDes bus interface. Figure 3.10 shows the FSMD of the SerDes bus interface. The CPU initiates a read or write by pulling up the 'stb' and 'cyc' signals, where the 'stb' signal is visible to all bus devices and the 'cyc' signal is routed to a specific slave based on the address. At the same time, the 'we' signal is write enable, indicating whether the slave should store or retrieve data. Another difference between read and write operations is whether there is valid data on the data line, while the address line is the same. After the state machine detects changes in the 'cyc' and 'stb' signals in the IDLE state, it will enter the WRITE or READ state according to the logic of the 'we' signal.

If the state machine enters the WRITE state, the values on the address line and data line at this time are temporarily stored in the corresponding memories. Then the 'ack' signal can be pulled high to indicate to the CPU that the write operation has been responded to. Although the temporary storage operation requires more hardware resources, it can avoid occupying the bus and CPU for a long time, reducing CPU efficiency. When the TX state is entered in the next clock cycle, the SerDes bus interface will send the Command, Length, Address and Data sequence

in the memories to the subsequent TX asynchronous FIFO in sequence. After sending all bytes, the state machine will return to the INIT and IDLE states, waiting for the next bus operation.

If the state machine enters the READ state, the SerDes bus interface will only store the values on the address line. In the subsequent RX phase, Command, Length and Address sequence will also be sent, except Data sequence. After all bytes are sent, the state machine will enter the RECEIVE state and wait for the data to be read back. In this state, the state machine always checks the "empty" flag of the RX asynchronous FIFO connected to it. Once the FIFO is not empty, a byte is read from it. After four reads, a complete word has been stored in the local memory. At this time, an acknowledge signal can be sent to the CPU, and the word is placed in the data line waiting for the CPU to sample. Then, the state machine will also go to the INIT and IDLE states. Compared with write operations, read operations occupy the bus and CPU longer, and it is difficult to determine when the read data will be received. Sometimes it may cause a timeout and the CPU abandons the read operation, thus reducing the success rate of the read operation.

3.3.2 Bus master interface

In a passive SoC, the instructions received by the SerDes system require a bus master to complete the bus read and write. Using the LiteX framework a DMA was set up in the form of a Wishbone master. The Wishbone master was attached to the Wishbone bus by using the built in functions for creating a bus-interface and connecting the master to the bus. The addresses in on the Wishbone is byte addressable and therefore the last two bits of the address signal are discarded.

DMA based on UART

The initial DMA controller was based on the UART transceiver and protocol. This was a first step to verify and implement a bus master and slave in the form of a peripheral which was a simple block memory that was implemented. This peripheral could issue and respond to simple reads and writes on the Wishbone bus. Through this implementation the bus transaction protocols could be verified and data was both sent and received, to and form this peripheral. With the transactions on the internal bus functioning the inter-chip communication had to be realized. A class called UARTBone (UART-Wishbone) from the LiteX library was used as a interconnect between the physical UART and the bus. It is a bus master capable of issuing R/W on all of the connected slaves.

The LiteX class UARTBone integrates the physical UART functionality from the UART class as well as the specific protocol to interpret the data received over the channel. The protocol is as follows, first received is the command type, read or write, then the length of the data that is to be sent in bytes, followed by the 4 byte address. Depending on if the command is a read or a write the next state is decided. For the write operation the next state is the DATA-WRITE state where the data is written and if the length which was received earlier is longer than 1, the protocol will transition back and forth until the complete transaction is done.

The read operation is similar but there the data request has to be sent and then the read data from over the bus has to be sent back to the UART. Therefore the state machine waits in the DATA-READ state until it has received the data and then it also goes back and forth until the complete read has been done.

DMA based on SerDes

The SerDes and UART are both serial-data links and that is naturally the motivation for investigating the UART first because of its more developed infrastructure.

Due to the similarities of the UART and SerDes the DMA controller for the SerDes design is built around the same state machine. The only difference is the physical implementation of the communication channel. The UARTBone class is built upon another class called Stream2Wishbone and using that class with a different physical layer.

The pre-built UART class has a FIFO due to the different speeds at which it transfers the data and the surrounding components clock frequency. This means that the DMA controller which was built for the UART had the FIFOs control signals such as rx_fifo_empty and tx_fifo_full. The same signals was later used for the SerDes implementation as well, since when the receiving FIFO does not have anymore data, there should be no sampling. If the transmitting FIFO has no room for new data, the DMA controller needs to stay in the same state and wait to transmitted the needed data. If states are skipped crucial commands for the protocol wont reach the other side.

3.3.3 TX/RX asynchronous FIFO

In this study, the SoCs based on LiteX frame uses a 75 MHz clock, while the dedicated SerDes system uses a 7.5 MHz clock. Therefore, when the SerDes system, as a submodule in the SoC, needs to transmit data with other modules, it must consider the issue of data crossing clock domains. Asynchronous FIFO is a crucial component in digital systems for buffering data between two domains operating at different clock frequencies. Here, two asynchronous FIFOs are used to ensure smooth bidirectional data transfer without data loss or overflow.

Figure 3.11 shows the block diagram of the asynchronous FIFO used in this study. From the block diagram, it shows that the write pointer handler is driven by the write clock and the read pointer handler is driven by the read clock. Therefore, write pointer is aligned to the write clock domain whereas the read pointer is aligned to the read clock domain in an asynchronous FIFO. The asynchronous FIFO needs to determine its empty or full state by comparing the write pointer and the read pointer, but this will cause metastability problems during clock domain crossing. Write and read pointers are usually not 1-bit signals, so handling them across clock domains is more complicated. By converting binary numbers into Gray codes, the codes of any two adjacent numbers can be made to differ by only one binary digit. In this case, the problem of multi-bit signal crossing clock domains can be simplified to the problem of 1-bit signal crossing clocks, that is, using two-stage flip-flop driven by the target clock domain to synchronize them. It is worth noting that if Gray code is used, the depth of the FIFO must be

guaranteed to be a power of 2. Otherwise, when the pointer changes from the maximum value to 0, its Gray code is not continuous and metastability issue may occur.

In the write pointer handler, when the write enable signal is high and the FIFO status is not full, the write pointer will increase automatically. Similarly, in the read pointer handler, when the read enable signal is high and the FIFO status is not empty, the read pointer will increase automatically. The read pointer enters the write clock domain after Gray encoding and clock synchronization, where it is compared with the write pointer to generate the FIFO full flag. The write pointer enters the read clock domain after Gray encoding and clock synchronization, where it is compared with the read pointer to generate the FIFO empty flag. In the write pointer and read pointer handler, the Gray-coded read and write pointers can be directly compared without converting the synchronized Gray code into binary numbers for comparison. The generation of the FIFO empty flag is obvious, that is, when the read pointer is equal to the write pointer, it means that the FIFO has been read empty. To determine whether the FIFO is full, the read and write pointers need to be expanded. In this way, by comparing the different expansion bits but the same other bits, it can be determined that the read and write pointers differ by the width of the entire FIFO, which means the FIFO is full.

The FIFO submodule's asynchronous read and write operations are completed under the control of the read pointer and write pointer, empty and full flags, and read clock and write clock. Two asynchronous FIFOs are used in this study, one is the TX FIFO with fast write and slow read, and the other is the RX FIFO with slow write and fast read. As a bridge between the bus interface and the SerDes system, an asynchronous FIFO only needs to connect the appropriate control and flag signals to both sides to achieve safe and smooth data transfer.

3.4 LiteX SoC Integration

For the base of this thesis a existing build for the VexRiscv CPU was used along with a number of instantiated modules needed for the LiteX ecosystem to function. Some important modules are the cellular RAM, UART, Wishbone bus,

3.4.1 LiteX modules

UART

The UART is implemented on the LiteX SoC for communication with the system. It is needed for the provided terminal emulator LiteX-term to work. The UART features interrupts for proper communication with the CPU. The default UART is instantiated through the micro-USB port on the Digilent Nexys4 board but with the UART python function it was instantiated and connected to the Pmod connectors which was then connected to Pi-Pico boards running a USB to UART probe. The reason for this is that the default UART was in use during development to supply the logic analyzers waveform through the UARTBone interface.

3.4.2 Migen

Migen is a HDL in Python and serves as the backbone of the LiteX environment. It uses object oriented programming to instantiate modules such as the SerDes-channel master, Wishbone bus, etcetera. Each object has it's associated signals if they are made reachable with ".self".

Since it was the chosen language of LiteX most of the integration of different modules was done in Migen. It is a powerful tool with the capabilities of a comparable language like Verilog.

3.5 Hardware Implementation

3.5.1 FPGA platform

In this study, the Digilent Nexys4 FPGA development board is utilized as a powerful platform for hardware implementation and prototyping. This development board is based on the AMD Artix-7 FPGA (model XC7A100T-1CSG324C), providing over 100,000 logic cells, making it suitable for complex digital circuit design and verification. The Nexys4 is equipped with a wide range of peripherals, including switches, buttons, LEDs, seven-segment displays, a VGA interface, a USB-UART interface, and multiple Pmod expansion interfaces.

The Nexys4 development board features a dual analog/digital Pmod interface (Pmod JXADC), which is particularly notable for its capability to input/output differential signals. This functionality is essential for applications requiring high signal integrity and noise immunity, such as high-speed data transmission, precision analog measurements, and communication systems. The Pmod JXADC interface provides 8 input/output pins that can be configured as 4 differential pairs. Differential pair is composed of two traces, routed side-by-side, and that carry equal magnitude and opposite polarity signals on each trace. As mentioned in Section 2.1.1, differential signaling significantly reduces electromagnetic interference (EMI) and crosstalk, ensuring robust and reliable signal transmission over longer distances or in electrically noisy environments.

3.5.2 Physical interface and connection

The physical interface and connection part mainly includes the setting and allocation of FPGA pins, as well as the configuration of wires and interfaces.

TMDS (Transition Minimized Differential Signaling) is a differential I/O standard for transmitting high-speed serial data used by the DVI and HDMI video interfaces. TMDS is only available in HR I/O banks and requires a VCCO voltage level of 3.3 V. The IOSTANDARD is called TMDS_33 in AMD 7 Series FPGAs. The TMDS_33 standard requires external 50 ohm pull-up resistors to 3.3 V on the inputs end as it is a current drive. Low-voltage differential signaling (LVDS) is also a powerful high-speed interface in many system applications. Due to its low voltage swing, LVDS generally consumes less power compared to TMDS_33. However, since the test environment of this study is based on the LiteX framework,

the I/O bank voltage where the differential signal pins are located has been specified as 3.3V by other components in the SoC, so we can only choose TMDS_33 standard. In the constraint file, the input and output differential signals of this study need to be connected to the two pairs of differential pins in the I/O bank, and the pins are set to TMDS_33 standard.

Figure 3.12 shows the signal channel between two FPGAs based on TMDS_33 standard. The 100 ohm resistor in the dotted box in Figure 3.12 is not required by TMDS_33 standard, but because each Pmod port on the Nexys4 development board used in this study is connected to a 100-ohm resistor by default to protect the FPGA. This 100 ohm resistor can become a limiting factor when transmitting high-speed differential signals based on the TMDS_33 standard. Since the primary goal of this study is to verify the principles of using a SerDes channel to achieve the communication between two SoCs, this limiting factor does not affect the final conclusions. While this design are not worried about the the capability of FPGA to handle the target speed, the demonstration of the channel at the target speed is constrained by the specific development boards and the precise model of FPGA devices on those boards.

When using physical wires as channels to connect the differential signaling input/output terminals of two FPGAs, it is necessary to make sure their quality meets the requirements of high-speed differential signals. In differential signaling, a logic high is indicated when the non-inverted signal's voltage is higher than the inverted signal's voltage. Otherwise, it is a logic low is. The transition between these states occurs at the crossover point, where the two voltages intersect. One critical aspect of differential signaling is the necessity to match the lengths of the wires or traces carrying the differential signals, which is vital for achieving maximum timing precision, as the crossover point should correspond exactly to the logic transition. Any disparity in the lengths of the conductors results in a propagation delay mismatch, causing the crossover point to shift and potentially leading to timing errors and reduced signal integrity.

A twisted pair can be used as a balanced line can greatly reduce the effect of noise currents induced on the line by coupling of electric or magnetic fields. The idea is that the currents induced in each of the two wires are very nearly equal. The twisting ensures that the two wires are on average the same distance from the interfering source and are affected equally. The noise thus produces a common-mode signal which can be cancelled at the receiver by detecting the difference signal only.

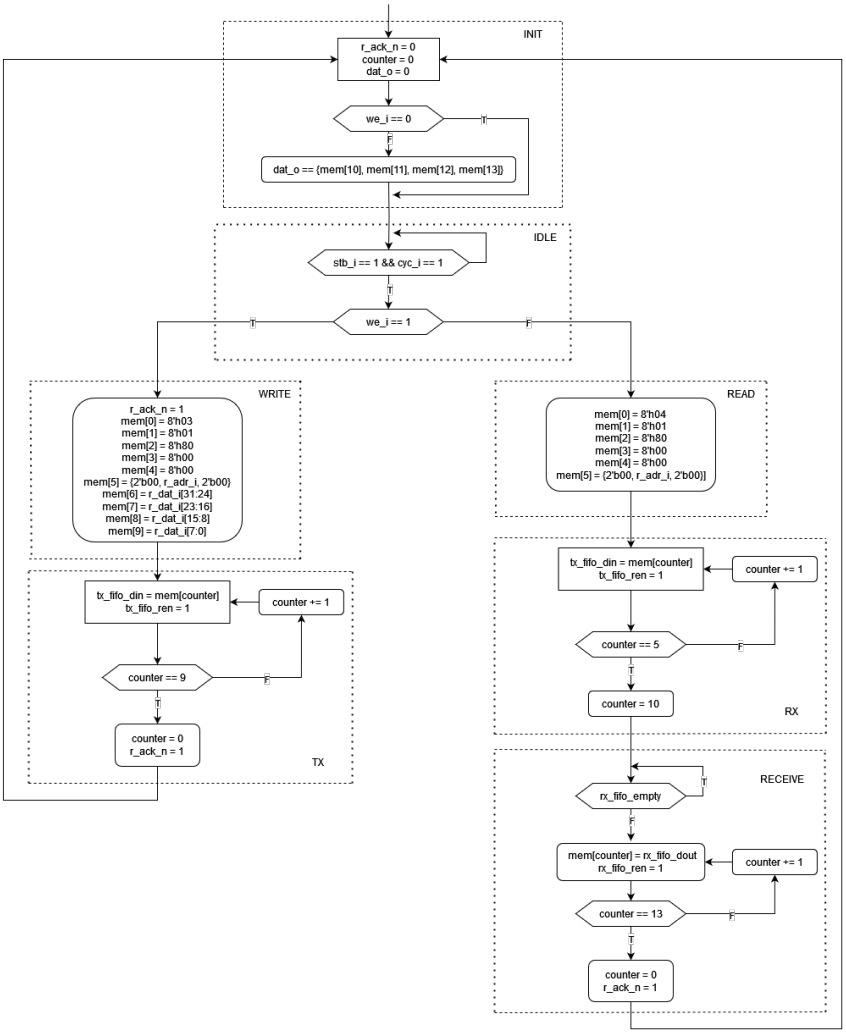

Figure 3.10: FSMD of the SerDes bus interface.

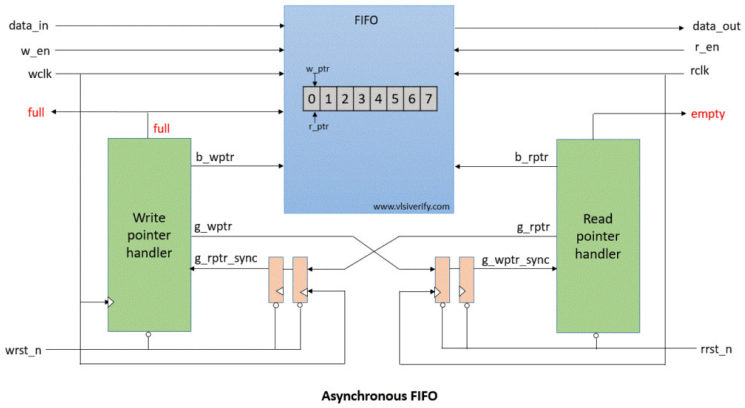

Figure 3.11: Asynchronous FIFO block diagram.

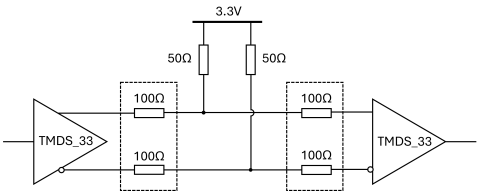

Figure 3.12: Signal channels between two FPGAs based on TMDS_33 standard.

Chapter 4
Verification and Results

The verification of the circuit was done in incremental steps that in the end lead to the complete circuit which embodies all of the different aspects of the smaller circuits for the final result. The incremental steps were, perform a read and write transaction on a LiteX SoC slave module over the Wishbone bus. Connect a simpler serial interface (UART) that issues commands over the serial channel, making it possible to communicate with the Wishbone bus on the other SoC. This was then expanded so that read and write operations could be sent over this serial channel and connected to this Wishbone bus.

The smaller circuits were analyzed throughout the process using the LiteScope as well as a integrated logic analyzer. This meant that the different parts could be confirmed and used as reference and throughout the development. However these results has been left out of the thesis as to not overcrowd the result.

4.1 Verification Environment and Tools

4.1.1 Experimental Set-up

Here, we deploy a SoC on each of the two FPGAs as a platform for verifying this high-speed SerDes channel, as shown in Figure 4.1. The set-up uses twisted pair as the connection of differential pairs on two different FPGAs. At the differential input of the FPGA, the port is connected to a 3.3 V power supply voltage through a 47 ohm resistor, which can be seen from Figure 4.2. In addition, it can been see that the Pi-Pico boards uses Dupont cables to connect to the FPGAs for serial communication between the PC and the FPGAs.

4.1.2 LiteScope Analyzer

LiteXScope was used for debugging the SoC through the UARTBone connection. To use the LiteScope analyzer the module was first instantiated with the Python function and added to the SoC with the signals under test connected to it.

With the hardware in place the LiteScope was used through the litex_server which is packaging the information in a way that it can be interpreted on the Wishbone bus. LiteScope was armed with specific triggers such as a expected received pattern, state and other control signals.

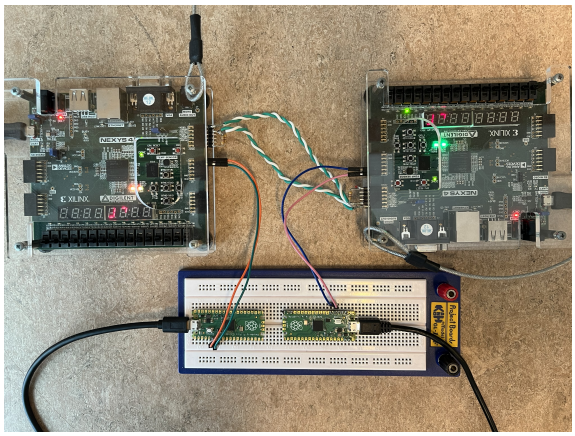

Figure 4.1: Overall experimental set-up.

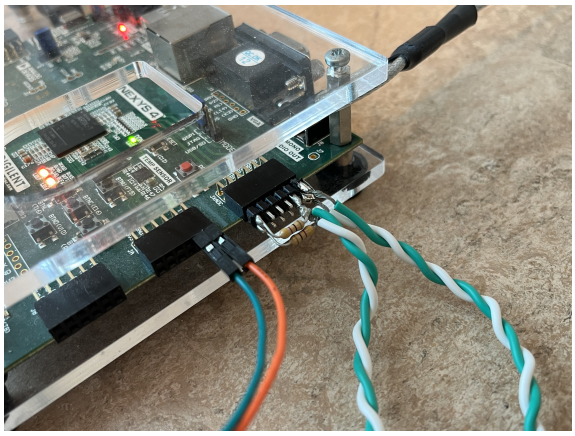

Figure 4.2: Zoomed view of the differential input/output interface.

4.1.3 LiteX Term

LiteX term was used to verify the more complete design when the communication channel was set up. Basic read and write operation were possible as well as uploading compiled C programs. These C programs consisted mostly of a few read and write operations done in a specific order however the potential is there to do more.

Through one of the SoCs a write was issued to the second SoC and then later read and displayed in the terminal, signifying a complete transaction and successful communication.

4.2 Prototyping Verification Results

In this section the verification results for the different submodules of the complete SerDes system will be presented. This acts as supporting material for the underlying design of the system. The studied signals are limited due to the LiteScopes limitations of only allowing around 32 bits to be studied at a time.

The implemented sub-modules of interest are presented in logical order, following the data flow throughout the read and write operations. The order is as follows, bus interface to access the requests from the CPU. TX FIFO to store the commands to be sent. OSERDES controller and encoder to send the commands over the channel. ISERDES controller with the decoder to receive the commands. The bitslip controller to align the ISERDES so the right data is received, and finally the RX FIFO connecting back to the bus interface.

4.2.1 Bus Interface

The bus interface receives the request from the CPU to issue a write operation through the litex_term according to the wishbone protocol proceeds to raise the corresponding signals, the cyc, stb and we. This lets the slave know that it is time to start sampling the upcoming data in form of cmd, length, address and the data to be written. This can be seen in figure 4.3 below, as the interface enters state one which is the WRITE state. Here all the information is stored in local registers to be sent out one byte at a time. The bytes are observed as the interface enters state three which is the state TX and the tx_fifo_din is assigned to the corresponding registers each clock cycle. Take note of the pattern as it will be a re-occurring sequence as it is followed through the systems different modules. The bus interface finally is done and returns to the INIT state five.

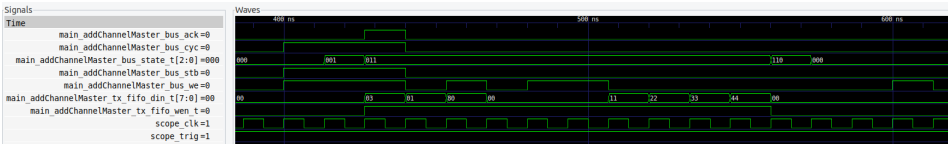

Figure 4.3: Waveform from LiteScope containing the signals of interest during a write operation.

In figure 4.4 a read is requested and the initial process is similar to the write, except the cmd being changed to the value four. What is being seen in the figure are the states and the RX FIFO signals. The state transitions are IDLE, to READ, to RX which is state four and that is when the sampled commands and data from the bus are sent. Then comes state five which is the receive state that takes a significantly longer time since it is waiting for a response from the other SoC. This response is the read data which can be seen on the rx_fifo_dout, 0x11223344 in hexadecimal radix. With the received data now retrieved and stored in internal registers the acknowledge signal is raised before entering the INIT state. In the INIT state the dat_o signal is assigned to the registers containing the data and sent to the bus along as the acknowledge signal is pulled low, signaling a completed

request, allowing the bus to be released and available.

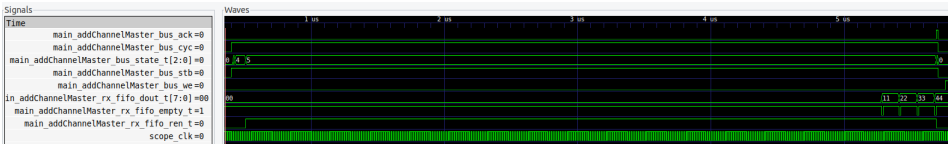

Figure 4.4: Waveform from LiteScope containing the signals of interest during a read operation.

4.2.2 Asynchronous TX FIFO

The asynchronous TX FIFO purpose is to store the data and commands to over the SerDes channel. This FIFO also allows for clock domain crossing which is necessary since the wishbone bus is running on the system clock while the SerDes has the custom PLL generated clock frequency appropriate for the communication link. This can be seen in figure 4.5 as the data is pushed in to the FIFO much faster on the tx_fifo_din data-line than it is popped out on the tx_fifo_dout data-line. It is also observed that for the FIFO popping operation the write enable signal is asserted high for the sequence of useful data. During the popping the read enable signal is asserted high for the duration of the data transmission, controlled by the upcoming module which is the OSERDES controller.

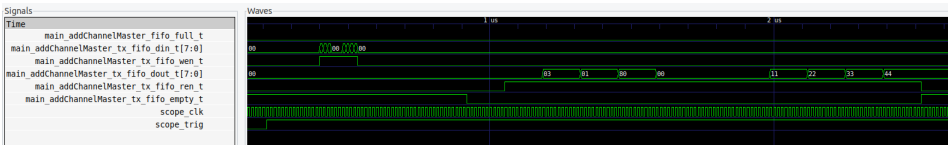

Figure 4.5: Waveform from LiteScope of the TX FIFO's response to a write operation.

4.2.3 OSERDES Controller

The OSERDES controller decides what to send out through to the OSERDES using the data stored in the TX FIFO. It packages this data with the necessary start and stop byte implemented in the protocol (K27.7) and (K28.4). In figure 4.6 this can be seen when D_sync starts outputting the start bit (F8 in hexadecimal) followed by the data for a write operation. The control mechanism can also be seen, when the TX FIFO is not empty it is known that there is valid data stored in the FIFO to ready to be sent. This is indicated in the waveform by the tx_fifo_empty signal going low and the following clock cycle the tx_fifo_ren is asserted high, popping the FIFO and outputting the correct data after the start byte and finishes with the start byte mentioned earlier.

In the figure the decoder is also verified with the D_enc signal, performing the 8/10b character conversion, translated from the D_sync and k_oser_sync.

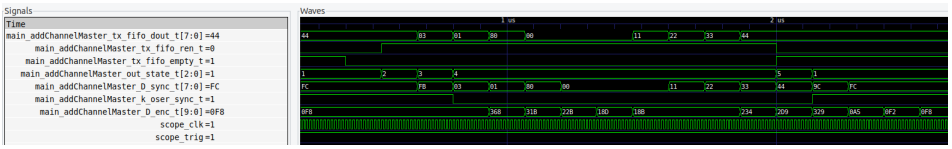

Figure 4.6: Waveform from LiteScope showing the OSERDES Controller behaviour during a write operation.

4.2.4 ISERDES Controller

When receiving the transmitted data the ISERDES controller checks for the start byte and when that comes it is known that the following bytes until the stop byte is received is useful data and should be handled accordingly. In figure 4.7 we can see that the 10 bit character received by the ISERDES is sent through the decoder and is converted to it's 8 bit equivalent as well as the k-signal indicating if it is a control character or not. Looking at Q_dec and rx_fifo_din it is seen that only the useful data is stored in the RX FIFO, indicated by fifo_valid being high while the rx_fifo_din data-line contains the transmitted useful data.

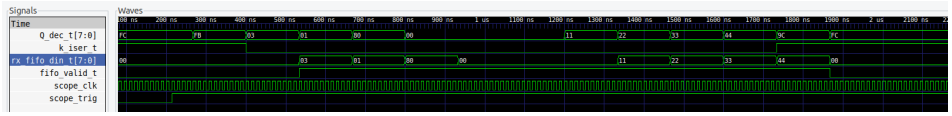

Figure 4.7: Waveform from LiteScope showing the ISERDES Controller behaviour during a write operation.

4.2.5 Bitslip Controller

The bitslip controller is in charge of keeping the ISERDES aligned. If it notices that the phase has drifted it will issue a bitslip, which is seen in figure 4.8. The training pattern has shifted and it needs to issue bitslips until it is properly in phase again. The state of the bitslip controller changes from IDLE, where it is in phase to the BITSLIP state (one) and then waits in the WAIT state (two) for three clock cycles so that the result can be compared with the correct pattern. The pattern does not match and therefore it transitions back to the BITSLIP for another try. This is continued until the correct pattern is observed. Since it takes too much time for it to find it, it can not be fit in to the figure but it does not give up until the pattern is found. With the correct pattern it will jump back to the IDLE state and wait until it notices another phase drift.

Figure 4.9 below shows the state of the bitslip controller, in charge of aligning the ISERDES. It is in the IDLE state during a correct write operation.

4.2.6 Asynchronous RX FIFO

The RX FIFO is much like the TX FIFO, both used for the clock domain crossing with the Wishbone interface. It also stores the data needed for communication

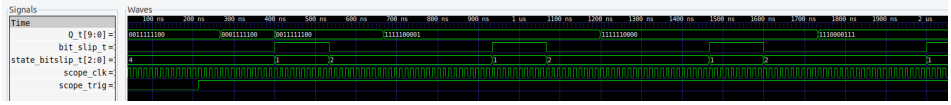

Figure 4.8: Waveform from LiteScope showing the start of the bitslip process.

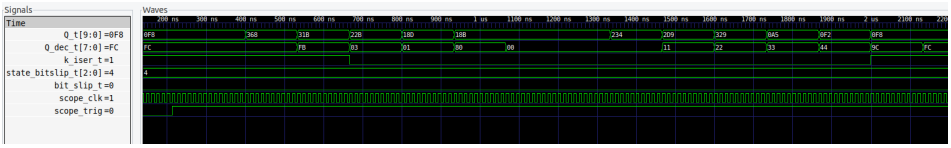

Figure 4.9: Waveform from LiteScope showing the idle behaviour of the bitslip controller.

with the Wishbone bus. It is seen in figure 4.10 that the commands and data for the write operation to be done on the Wishbone bus is sampled. The signal fifo_valid from the ISERDES controller is connected to the write enable of the RX FIFO and therefore when rx_fifo_dat_o has the correct values the FIFO samples the correct values which is seen when rx_fifo_empty goes low for a short period, one system clock cycle to be exact.

In the figure the Wishbone bus signals of SoC number two is also seen acknowledging the write operation on the bus. Meaning the write was executed on the receiving SoC.

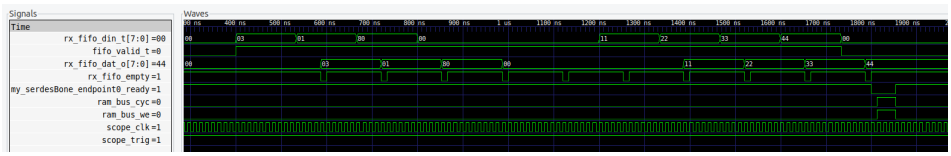

Figure 4.10: Waveform from LiteScope of the RX FIFO's response to a write operation on the receiving SoC's side.

Chapter 5
Discussion and Future Work

This thesis suggests a method for high-speed serial communication between two SoCs using a dedicated SerDes channel. It is a proven method that has been used before in other communication applications such as gigabit Ethernet and chiplet arrangements. The theoretical potential of this implementation is high compared to other serial communication channels such as the UART, achieving a factor of 10000 in transfer speeds. However this specific implementation with the hardware constraints of the Digilent Nexys4 Pmod ports not handling frequencies over 500 MHz and the custom made wire connecting the two SoCs susceptibility to interference restricted the ambitions of achieving gigabit speed. The performance of the communication channel had to be sacrificed for the stability of the data transfer. A much more modest speed of 75 MHz was settled for.

Even with the lower frequency in place there is still a significant amount of corrupted data. In hindsight some form of error handling should have been necessary to ensure completely reliable communication, as it is now the bit error rate is significant and can trigger unwanted state transitions or hinder necessary ones from ever happening. The custom communication protocol implemented with the different controllers in the system does not allow the sender to know what the receiver is doing and therefore the receiver can be occupied with other tasks such as phase aligning. If the receiver is performing bitslips in order to align the deserializer with the input stream, it expects the predefined character K28.7 to be sent. If data is instead being transmitted that data would be ignored without the knowledge of the sender.

The achievement of this thesis is the integration of the customized communication protocol with the already existing LiteX SoC framework, accessing the memories of another SoC, enabling reads and writes over a wishbone bus. This opens up for a lot of possibilities for different applications which requires access of each others data. The suggested application was a AI/ML accelerator which consumes a lot of data or has weights stored of chip due to cost and other restrictions. One could imagine a system which splits up the work of a CNN calculation on many individual smaller accelerators and then combines the separate sub-results in to a final accumulated result.

The result highlights the complexity of high-speed data transmission with numerous, not only for handling the bus protocols and what data to send, but also to adjust and minimize problems that stem from physical sources. The different reference clocks leading to phase drift. The thesis therefore incorporates some

analog aspects in to the work.

The verification of the circuits proves that the sub-modules work as expected but the path the data takes from the OSERDES and ISERDES has some errors. Especially looking at when the phase has drifted and a bitslip is soon needed, when the frame is close to the edge there are 1-bit errors. This makes the bitslip operation and control difficult since the received data is unreliable.

Enabling caching would be a good way of increasing the data transfer of actual data. With the protocol as is there are seven bytes transferred per four bytes of data, meaning a efficiency of around 57% for reading and for writing the transferred bytes are 11 and still only four useful bytes leading to an efficiency of around 36%. Even with a small block size of 8 there would be a significant increase to 91% and 82% respectively. The change is relatively easy to make, changing the sent length command to the wishbone bus from one to eight and increasing the counters in the state machines responsible for sending the data throughout the SerDes system.

References

[1] Y. Wu, T. Li, Z. Shao, L. Du, and Y. Du, *An Efficient Design Framework for 2×2 CNN Accelerator Chiplet Cluster with SerDes Interconnects*, in *2023 IEEE 5th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2023, pp. 1-5. DOI: 10.1109/AICAS57966.2023.10168573.

[2] F. Zhang, *High-speed Serial Buses in Embedded Systems*, Springer, 2020. https://link.springer.com/book/10.1007/978-981-15-1868-3

[3] A. Athavale, Carl Christensen, *High-Speed Serial I/O Made Simple; A Designer's Guide with FPGA Applications*, Xilinx, Inc, 2005. https://www.xilinx.com/publications/archives/books/serialio.pdf

[4] C. Cenci and P. Stocchetti, *High speed (balanced) differential signal measurement with single-ended oscilloscope (unbalanced)* [Internet], CBL Electronics, March 21, 2022. [Accessed: May 17, 2024]. Available from: https://www.cblelectronics.com.

[5] Yoo, Y. and Choi, B.-D. (2023) *Single-ended amplifier-based touch readout circuit with immunity to display noise*, Journal of Information Display. Edited by 01/01/2023, pp. 127–135. doi:10.1080/15980316.2022.2154863.

[6] Smutzer, C.M. et al. (2010) *A Passive Differential Termination and Signal Sensing Device for High-Speed (40 Gb/s) Test Applications*, IEEE Transactions on Instrumentation and Measurement, Instrumentation and Measurement, IEEE Transactions on, IEEE Trans. Instrum. Meas, 59(7), pp. 1775–1782. doi:10.1109/TIM.2009.2030715.

[7] F. Zarkeshvari, P. Noel, S. Uhanov, and T. Kwasniewski, *An overview of high-speed serial I/O trends, techniques and standards*, in *Canadian Conference on Electrical and Computer Engineering 2004 (IEEE Cat. No.04CH37513)*, 2004, vol. 2, pp. 1215-1220 Vol.2. DOI: 10.1109/CCECE.2004.1345340.

[8] LiteX GitHub Repository. EnjoyDigital, 2014-present. https://github.com/enjoy-digital/litex

[9] Richard Herveille, *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, rev. version: B4, 2010. By Open Cores Organization, 2010. https://www.opencores.org

[10] AMD, Inc., *Versal Adaptive SoC GTY and GTYP Transceivers Architecture Manual*, v1.3, November 26, 2023, [Online]. Available from: `https://docs.amd.com/r/en-US/am002-versal-gty-transceivers/Reference-Clock-Power`

[11] Nannipieri Pietro, DAVALLE Daniele and Fanucci Luca. *A Novel Parallel 8B/10B Encoder: Architecture and Comparison with Classical Solution*, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 2018, E101.A. 1120-1122. 10.1587/transfun.E101.A.1120.

[12] AMD, Inc., *7 Series FPGAs SelectIO Resources User Guide*, v1.10, May 8, 2018, [Online]. Available from: `https://docs.amd.com/v/u/en-US/ug471_7Series_SelectIO`