Morten Pedersen

Compacting On-Chip Ultra-Wide Global Interconnect Buses

Project Report
TFE4590 - Specialization Project

Trondheim, December 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU

Norwegian University of Science and Technology

Project report for the specialization project TFE4590

Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Abstract

The trend of decreasing process technology size leads to the increasing complexity levels for both logic and memory in integrated circuits. This increasing complexity, coupled with the fact that the maximum clock frequency does not follow this trend, leads to a problem; how can it be avoided that interconnect buses becomes a bottleneck for the system. The currently most used solution to this problem is parallelism. For some systems, like the GPU, that utilize extreme parallelism, this is starting to lead to routing congestion. Routing congestion leads to lower data rates, higher power consumption, and higher area usage.

This report explores alternatives to a parallel solution for reducing the number of wires on the interconnect buses. The report also presents a solution capable of a four-times reduction in the number of wires. The proposed solution trades the wire reduction with higher power consumption while maintaining the same performance.

Preface

This report is the outcome of a task given in the course TFE4590 - Specialization Project for the Faculty of Information Technology and Electrical Engineering at the Norwegian University of Science and Technology (NTNU) and supervised by ARM Norway. The project has been both fun and challenging, even though it has been constrained by time.

I spent the first month of the project reading papers and orienting myself in the field. A big part of the project was spent designing the proposed solution. The last few months were spent documenting the process in this report.

I made all the figures in the report, except for the waveform figures, as can be seen in Section 4.2 and Section 4.3. These figures are snippets from the Xilinx Vivado simulation tool.

I would like to thank my supervisors, Morten Werner Lund (ARM Norway) and Volkan Kursun (NTNU), for their guidance. It would not have been possible to complete this project within the timeframe without the insight they provided.

Contents

Abstract iii

Preface iv

List of Tables vii

List of Figures viii

List of Abbreviations ix

1 Introduction 1
- 1.1 Overview . . . 1
- 1.2 Description . . . 1
- 1.3 Motivation . . . 2
- 1.4 Report Outline . . . 2

2 Background 3
- 2.1 Buses . . . 3
- 2.2 Mesochronous and MultiPhase Clocking . . . 4
- 2.3 Timing Requirements . . . 5
- 2.4 The XOR Gate . . . 5
- 2.5 Gate Equivalents . . . 6
- 2.6 Clock Gating and Fan-out . . . 6
- 2.7 Design for Testability . . . 7
- 2.8 The Mixed-Module Clock Manager Module . . . 8
- 2.9 Serializer and Deserializer . . . 8
  - 2.9.1 Shift Register-Based Serializer-Deserializer . . . 9
  - 2.9.2 Wave-Pipeline Serializer-Deserializer . . . 9
  - 2.9.3 Double-Edge Triggered Flip-Flop Based Designs . . . 10

3 Implementation 12
- 3.1 Proposed Solution . . . 12
- 3.2 Data Path . . . 13
  - 3.2.1 Serializer . . . 13
  - 3.2.2 Deserializer and Multiphase Synchronizer . . . 14
- 3.3 Control Signals . . . 15
  - 3.3.1 Handshake . . . 15
  - 3.3.2 Control Signals for the Serializer . . . 15
  - 3.3.3 Control Signals for the Deserializer . . . 16
- 3.4 Timing . . . 16

4 Results and Discussion **18**
4.1 Methodology . . . 18
4.2 Start and Ending of a Transfer . . . 18
4.3 Backpressure . . . 19
4.4 Performance and Area . . . 20
4.5 Power consumption . . . 20
4.6 Further Discussion . . . 22

5 Conclusion **23**

6 Future Work **24**

Bibliography **25**

Appendices

A Large Figures **27**

B Source Code **29**
B.1 Top Module . . . 29
B.2 Serializer Module . . . 31
B.3 Serializer Control Module . . . 33
B.4 Deserializer Module . . . 36
B.5 Deserializer Control Module . . . 38
B.6 Serial Buffer Module . . . 39

List of Tables

4.1 Power consumption for the clock trees 21

List of Figures

2.1 Example of Multiphase Clocking 4
2.2 Truth table and symbol for the XOR gate 6
2.3 Simple clock gating 6
2.4 Example of how clock gating can be implemented 7
2.5 Block diagram for the MMCM module 8
2.6 A shift register-based Serializer design 9
2.7 An implementation of a Wave-pipelined Serializer-Deserializer 10
2.8 A double edge triggered flip-flop design 11
2.9 An implementation of a double edge triggered Serializer-Deserializer 11

3.1 Overview of the purposed solution 12
3.2 Data path for the serializer 13
3.3 Data path for the deserializer and the multiphase synchronizer 14
3.4 State Diagram for the Finite State Machine 16
3.5 Timing for the serializer 17
3.6 Timing for the deserializer 17

4.1 An example of both a start and ending of a transfer 19
4.2 An example of backpressure 19
4.3 Comparison of total power consumption 21

A.1 An example of both a start and ending of a transfer (Large version) 27
A.2 An example of backpressure (Large version) 28

List of Abbreviations

DesRxValid Deserializer to Receiver Valid signal

DesSerReady Deserializer to Serializer Ready signal

DETFF Double edge triggered flop-flops

DFT Design for testability

FIFO First In First Out

FPGA Field-Programmable Gate Array

FSM Finite State Machine

GE Gate Equivalents

GPU Graphics Processing Unit

IC Integrated Circuit

MMCM Mixed-Mode Clock Manager

SerDes Serializer-Deserializer

SIMD Single Instruction Multiple Data

RxDesReady Receiver to Deserializer Ready signal

SerDesValid Serializer to Deserializer Valid signal

SerTxReady Serializer to Transmitter Ready signal

TxSerValid Transmitter to Serializer Valid signal

WP SerDes Wave-Pipelined Serializer-Deserializer

Chapter 1

Introduction

1.1 Overview

The rapid improvements in process technology scaling lead to an ever-increasing transistor density on a single Integrated Circuit (IC) chip. The complexity of IC designs has also increased, with a higher density of logic and memory. Maximum clock frequency has not followed this trend and has been steady at about 4 GHz for the last couple of years. The focus has instead been on parallelism.

The first step in this direction was the multi-core processor [1]. The ideology behind the multi-core processor is that multiple simpler cores can execute instructions in parallel and with a lower frequency. When compared to a single-core solution, this can be used to maintain or even increase the performance of the overall system with lower power consumption. The power consumption is lower because the frequency can be halved and, with that, the supply voltage can be lowered. The first Graphics Processing Unit (GPU) was introduced around the same time. The GPU takes advantage of the highly parallel structures found in image processing by using Single Instruction Multiple Data (SIMD). With SIMD, it is possible to perform the same instruction on multiple data sets, which do not have dependencies on each other, in parallel. To be able to calculate with the extreme parallelism that the core philosophy, of the GPU, is built on, the data bus between the cache and the cores needs to have high enough performance so that the bandwidth of the bus does not become a bottleneck in the system.

As the performance of the GPU has increased, the interconnect buses have become wider to mitigate the bandwidth problems the increasing performance leads to for the bus. In turn, this leads to problems of its own that are starting to show.

1.2 Description

Each wire on the bus does not scale as well as the transistors. This, in addition to increasingly wider buses, causes routing congestion [2]. Routing congestion leads to lower data rates, higher power consumption, and larger area usage. As the wire gets smaller, the resistance in the wire increase while the capacitance stays the same. This means that the RC delay in the wire increase, while in contrast, the transistors typically speed up. In addition, with each new process technology, the wires are placed with more space between them to reduce capacitance and crosstalk. The delay through the wire is dependant on the length of the wire [3]. With increased routing congestion, it becomes harder to match the length of the wires in a parallel solution, leading to a variable delay between the individual wires that need to be handled.

Another problem with fully parallel solutions is that the leakage power is higher than for serial solutions. This is due to the higher number of repeaters, as there are more wires on the bus that require repeaters when compared to a serial solution.

1.3 Motivation

Resembling how the trend shifted from single-core to multi-core processors, the interconnect buses must now look towards serial solutions to fix the routing congestion it is facing [4]. This could be done using a time-multiplexing scheme at a higher clock rate than the system clock, either alone or combined with multiple serial data lines. In this report, different techniques to reduce the number of wires in the bus are explored. The serial design must not reduce the performance compared to a fully parallel counterpart. It must also be viable when power consumption and area utilization are considered. A new, proposed solution is also introduced. The solution utilizes multiphase clocking and is explained more thoroughly in Chapter 3.

1.4 Report Outline

The report is structured as follows: In Chapter 2 required background knowledge is introduced, in addition to existing solutions. Chapter 3 presets the proposed solution, including the control signals. In Chapter 4 the results are presented and discussed, with a focus on performance, power, and area. And lastly, in Chapter 5 the conclusion can be found.

Chapter 2

Background

This chapter introduces the required background knowledge for the proposed solution. In addition, existing solutions that can be used to reduce the number of wires in interconnect buses are explored.

2.1 Buses

A bus is the fundamental building block for transferring data in a system and between multiple systems. Buses allow two or more devices to communicate with each other, and transmit data. It is common for systems to have both internal and external buses. An internal bus is used between components in a system, while an external bus is used between multiple systems. Four characteristics describe buses:

- Frequency - The clock frequency of the bus.
- Transfer speed - The number of bits transferred per clock cycle.
- Throughput - The number of bits transferred on the bus per second.
- Width - The number of bits that the bus can transmit at the same time.

The four characteristics are highly dependant on each other, and changing one of them would lead to the others changing also. For example, if the width is lowered, the transfer speed would also be lower. If the frequency stays the same, the throughput is also lower. For this example, when changing one of the characteristics, two of the three remaining characteristics also change.

Buses can be single-ended or differential. A single-ended bus consists of a data wire in addition to a ground wire. The measurement of the bits on the bus is taken based on the voltage difference of these two wires. A differential bus also consists of two wires. But in this case, the wires are floating, meaning not referenced to ground, and complementary of each other. While a differential bus is costlier, more complex, and requires more circuitry, there are some advantages. A differential bus can send data over longer distances, with a higher data rate. This makes differential buses more useful in scenarios that require high data rates over long distances, like a keyboard for a personal computer. The main benefit of differential signaling is the possibility of filtering out noise. Common-mode rejection is used for this. Common-mode rejection means that when noise can be found on both wires, the noise on the two wires cancels each other out. As the signal is complimentary and the noise is not, the signal is strengthened, and the noise is canceled when the two signals are added together.

This does not mean that differential signaling is noise-free. If the noise is present in only one of the wires, the noise remains after common-mode rejection. One of the most well-known standards that utilize differential signaling is the USB.

Buses can also be either synchronous or asynchronous. A synchronous bus has an added clock wire on the bus. This makes the bus fast, but different components connected to the bus must operate at the same clock frequency. Note that there are ways to get around this, mainly using bus bridges to interface two different-speed buses with each other, but this will not be discussed in this report. An asynchronous bus, on the other hand, does not use clocking signals at all. Instead, handshake signals must be used to establish connections. Synchronous and asynchronous buses will be explored more, later in this chapter, in Section 2.9.

Bus transmission modes are described using the terms simplex, half-duplex, and duplex. A bus that is simplex can only send data in one direction, and any component can only either be a transmitter or receiver. Half-duplex means that data can be sent in both directions, but not at the same time. Duplex can send data in both directions simultaneously, making the components on the bus both transmitters and receivers in parallel. An example of a simplex device can be a controller, while an example of a half-duplex device is a walkie-talkie, and an example of a duplex device is a phone. Sending directions on the bus is not considered for this report.

2.2 Mesochronous and MultiPhase Clocking

For mesochronous, similar to synchronous, the difference between the clock and any signal in the design is constant. But for mesochronous systems, signal events happen with a fixed and unknown phase relative to the clock. The difference between synchronous and mesochronous clocking is similar. A mesochronous network consists of multiple clocks of the same frequency but unknown phases relative to each other.

Multiphase clocking is based on the same principle as mesochronous networks, with the difference that the phase between the different clocks in the network is known. Figure 2.1 shows an example of multiphase clocking. In this example, the phase for each of the four clocks is 0 degrees, 90 degrees, 180 degrees, and 270 degrees. This gives four rising edges with 90-degree between them that can be utilized in the circuit.

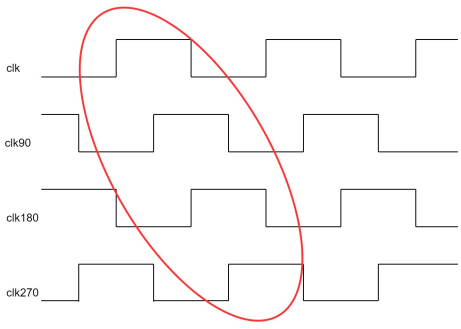

Figure 2.1: Waveform example of Multiphase Clocking

A negative aspect of multiphase clocking is the need for a clock tree for each clock. The need for multiple clock trees leads to higher power consumption when compared to a system

with a single clock. There is also a need for extra routing for the different clocks. Multiphase clocking will be explored further in Chapter 3.

2.3 Timing Requirements

It is important for any circuit that signals are ready to be used when they are needed. It is not only important for functionality but also for the reliability of the circuit. If a signal takes too long to propagate through the circuit, it would only be lucky if the output of the circuit is correct. It is therefore important that the delay throughout the circuit is controlled, and accounted for, in the design process.

For asynchronous designs, this means inserting delays where there is a possibility for race conditions. A race condition can happen if two signals arrive at slightly different times, which can lead to the output being in the wrong state for the time between the arrival of the two signals. For synchronous designs, propagation delay is the time it takes for a signal to pass from one clocked flip-flop to the next. But propagation delay is not the only delay that needs to be accounted for. Hold time and setup time also needs to be accounted for. Setup time is the minimum time the signal needs to be stable on the input of the flip-flop before a clock edge, while hold time is the minimum time the signal needs to be stable after the sampling clock edge. If the signal is not stable for the whole period of the setup-and hold time the sampling of the value becomes unreliable. The propagation delay, setup time, and hold time, summed, gives the minimum delay between two flip-flops. As the setup-and hold time are constant for all flip-flops in the design, the minimum delay is dependant on the variable propagation delay. The worst propagation delay in the design, meaning the propagation delay caused by the slowest path in the design, dictates the highest delay in the design. And by that, the highest clock frequency the design can operate.

2.4 The XOR Gate

The exclusive OR (XOR) gate can be constructed, as can be seen in Equation 2.1, using the three basic operators AND, OR, and NOT. The functionality of the XOR-gate, as can be seen in the truth table in Figure 2.2, is based on exclusivity. The output signal is only true when one, and only one, of the inputs, are true. The XOR-gate is important for binary additions in computers as it can perform modulo-2 operations.

$$A \oplus B = (A \cdot \overline{B}) + (\overline{A} \cdot B) \tag{2.1}$$

There is also another way to interpret the output of the XOR-gate. When input A is false, the output is the same as input B, and when A is true, the output is the inverse of B. This, in addition to D flip-flops, is used in Chapter 3 for the proposed implementation to implement functionality resembling multiplexers.

The switching power consumption of XOR-gates is significantly higher compared to basic gates, such as or-gates [5]. This is because XOR-gates has internal switching even when the output is not changed. An example of this is when input A is high, and input B is low, and they switch so that input A is low while input B is high. This behavior leads to internal switching even though the output of the XOR-gate does not change.

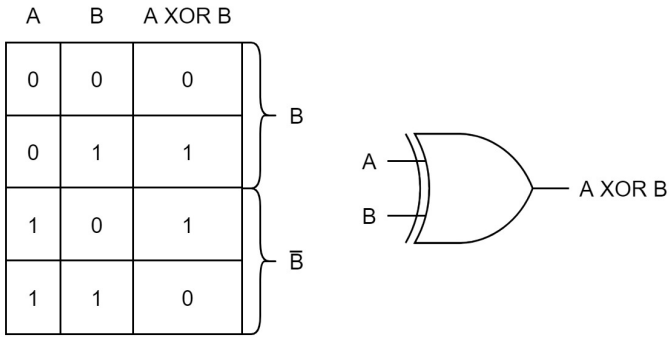

Figure 2.2: Truth table and symbol for the XOR gate

2.5 Gate Equivalents

Gate Equivalents (GE) is a unit of measurement for the relative size of a circuit [6]. GE is independent of technology size and the actual circuit. A single GE is defined as a two-input NAND-gate, which is consistent with four MOSFET transistors in static CMOS. A single D flip-flop with scan chain and a single output takes about 6 GE, while an XOR-gate takes roughly 2.33 GE. A FPGA has typically between 10 kGE and 1 MGE, and a fully-custom IC typically ranges between 10 MGE and 100 MGE.

2.6 Clock Gating and Fan-out

One of the biggest contributors to the power consumption in sequential circuits is the clock [7]. There is a combination of two reasons as to why that is. The first reason is that the clock is always switching, even when everything in the circuit is idle. The second reason is that the distribution of the clock in a clock network or a clock tree is big and adds to the capacitance of the circuit. When these two reasons are combined, the picture of how the clock is the biggest contributor becomes clearer. With the high capacitance and high switching activity, the dynamic power consumption is high. The clock trees also utilize clock buffers which, as mentioned in Section 1.2, adds slightly to the leakage power.

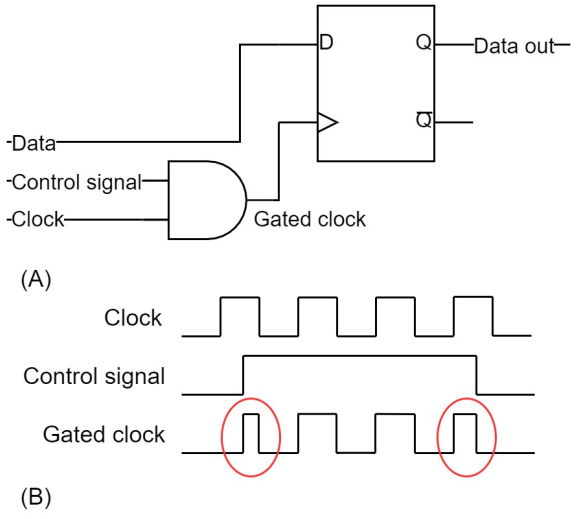

Figure 2.3: A: Simple clock gating
B: Waveform showing how glitching can happen

Clock gating is used to combat the high power consumption. The idea behind clock gating is that the clock can be switched off for the unused parts of the circuit. This can be done by branching off different clock signals from the main clock, grouping parts of the circuit together based on functionality. These new clock signals go through control logic where a control signal is used to decide if the clock should be turned on or not. There are several benefits to clock gating. The first is that the flip-flops in the design parts that are not currently used can not switch, reducing the

dynamic power consumption. The second reason is that the load of the rest of the clock tree is reduced when parts of the clock tree are turned off, reducing the total power consumption of the clock tree.

Great care needs to go into how the clock gating is implemented. A simple form of clock gating, as can be seen in Figure 2.3A, is to use a single AND-gate with the clock as one of the inputs and a control signal as the second input. This introduces the problem of glitching. When the control signal switches, the state of the clock signal is not taken into account. If the clock is at the end of a cycle when the control signal is enabled, the clock signal glitches, causing a positive clock edge to come out of synchronization. An example of this can be seen in the red circle to the left in Figure 2.3B. A glitchless, and by that, much safer way to implement clock gating can be seen in Figure 2.4. Here, a register is used to confirm that the control signal can only be used on the edge of the clock, making it much more stable, at the cost of an extra flip-flop.

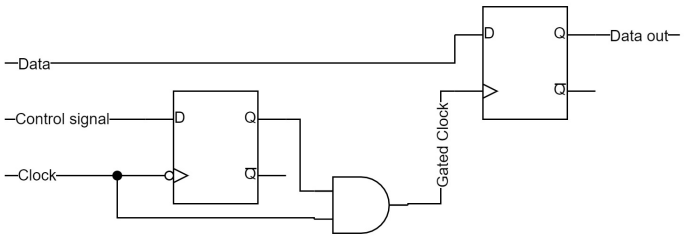

Figure 2.4: Example of how glitchless clock gating can be implemented

Fan-out describes the number of devices that any output drives. If the fan-out is too high, the device might not meet timing and voltage levels. A workaround for high fan-out is the use of buffers. Buffers isolate the output of a device from the inputs, and by that, the load. It is important to note that buffers increase the propagation delay, and the use of many buffers on a single path might affect the timing of the design. Note that fan-out is not exclusive to clock signals.

2.7 Design for Testability

Design for testability (DFT) refers to IC design techniques that add hardware to test the design after production. Testing designs is one of the most expensive and time-consuming parts of the design process. The main idea behind DFT is that it should be both easier and cheaper to test designs after fabrication so that bugs are caught before the IC is shipped out to customers, and by that, avoid a potentially massive cost due to recalls. Two important concepts to DFT is controllability and observability. Controllability refers to how easily one can control the value of the different signals in the IC when testing. Observability refers to how easily one could observe the value of a signal.

The most common method used in DFT is called scan chains [8]. A scan chain can be seen as one or more long shift registers of the flip-flops in the design. This shift register can then be used to observe the value in all flip-flops in the IC. With a predetermined pattern for the test stimuli, it is then possible to, for each clock cycle, look at the values in the flip-flops and determine if it is correct.

2.8 The Mixed-Module Clock Manager Module

Mixed-Mode Clock Manager (MMCM) is a module in 7-series Xilinx FPGAs [9]. MMCMs are used as a frequency synthesizer. MMCM can also be used as jitter filters and has deskew for the clocks. The main use for the MMCM that will be focused on in this rapport is the Fine Phase Shifting functionality, which can be used, in addition to the frequency synthesizer, to make multiple phase-shifted clocks.

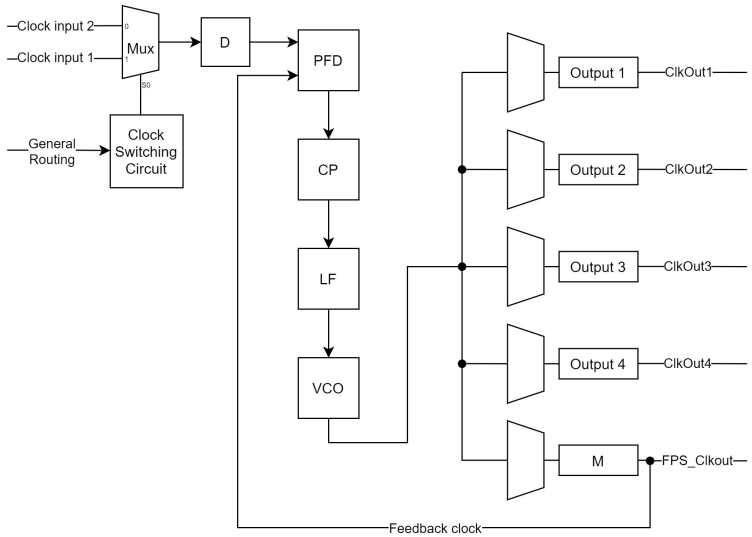

Figure 2.5: Block diagram for the MMCM module

A simplified block diagram for the MMCM can be seen in Figure 2.5. The reference clock is multiplexed in from either IBUFG, BUFG, BUFR, or BUFH. The reference clock is then used for a programmable counter divider (D), which can be used to divide the frequency of the reference clock in programmable ratios. The Phase-Frequency Detector (PFD) is used to compare the frequency and phase of the rising edge of the input clock and a feedback clock. The output of the PFD is a signal that is proportional to this comparison. A Charge Pump (CP) and Loop filter (LF) are then driven by this signal to generate a reference Voltage. The Voltage-Controlled Oscillator uses this reference voltage to generate up to eight output clocks of different phases, in addition to a variable phased clock, which is used for fine-phase shifting. An additional feature of the MMCM to note, which is not shown in the block diagram, is the Dynamic Reconfiguration Port (DRP). The DRP can be used to dynamically alter the frequency, duty cycle, and phase while the design is running.

2.9 Serializer and Deserializer

A Serializer-Deserializer (SerDes) is a compilation of two blocks used for transferring data over a reduced number of wires. The serializer, or Parallel Input Serial Output, inputs parallel data lines and reduces them to fewer, or even a single, data line. This line is then typically connected to a data bus. The deserializer, also called Serial Input Parallel Output, which is on the receiver side of the bus, restores the number of data lines to the original amount. Many different styles of SerDes exist, but it is typical for most solutions that the deserializer mirrors the serializer design.

In this section, existing SerDes solutions are introduced, and the positive and negative aspects of each solution are discussed. These solutions can be used as a comparison for the

proposed solution and are used to give an idea of what techniques can be used to reduce on-chip routing congestion.

2.9.1 Shift Register-Based Serializer-Deserializer

The shift register-based SerDes is one of the simplest forms of serialization and deserialization. It works on the principle of First In First Out (FIFO) as can be seen in figure 2.6. For the serializer, the data bits are inserted into the flip-flops through the multiplexer at the start of the serialization process. For each rising edge of the clock, one bit of the data is shifted onto the bus, and the remaining data is shifted one flip-flop closer to the bus. The deserializer is a mirrored version of this. The serial input line is shifted through the flip-flops until all the bits have been received and have reached a flip-flop. After this, the bits are multiplexed to the parallel output lines.

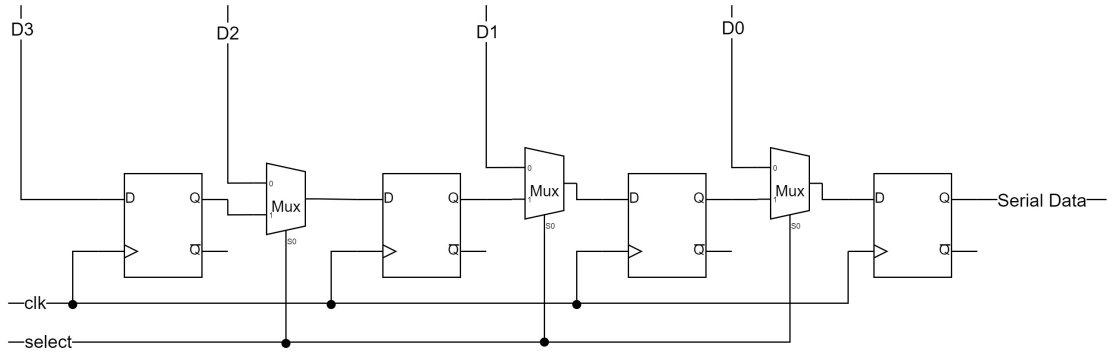

Figure 2.6: A shift register-based Serializer design

As only one bit is sent for each positive clock edge, the Shift register-based Serializer-Deserializer is not a good alternative to a pure parallel solution if the same level of performance is to be kept. For this to be possible, the clock frequency would need to multiply by n, where n is the number of bits to be serialized. This would lead to a high level of switching of the clock and thus an increase in power consumption. For this reason, the solution is not discussed further in this report, but rather kept to show a baseline solution.

2.9.2 Wave-Pipeline Serializer-Deserializer

The Wave-Pipelined Serializer-Deserializer (WP SerDes) is an asynchronous SerDes which utilizes delay elements to avoid signal racing in place of a clock. Multiple designs have been proposed for the WP SerDes, but the rest of this section will focus on the solution, as can be seen in Figure 2.7, proposed by B.C Hien et al. [10].

For the serializer, the transmission gates are used to make sure that the data signals do not propagate through the delay elements before an enable signal is set high. This results in a common timing reference. When the enable signal is set high, the data propagates through the delay elements and the transmission line as waves which are separated, in time, by the propagation caused by the delay elements. This makes the WP SerDes inherently different from other schemes, where shifting mechanisms are more commonly used.

For aligning the serializer with the deserializer, a pilot signal is used. These pilot signals are inserted at the beginning of each packet of data. When the pilot signal arrives at the last delay element of the deserializer, an enable signal goes high, which enables latches that pick up the data from the deserializer.

The WP SerDes is a complex solution. As it does not use any kind of storage for each wave, it is susceptible to noise which can cause errors. The WP SerDes also requires great precision of the design to be able to function as described. The design of the delay elements needs to be done with great care to reduce the potential for glitches. Which would have a devastating effect on the system, as there is no storage to separate the bits.

Another problem is the risk of desynchronization between the transmitter and the receiver. This leads to the need for extra sequence control signals, which adds to the complexity and slows the circuit down. In a synchronous circuit, these signals are not needed as the clock signal, in addition to storage elements, takes care of it.

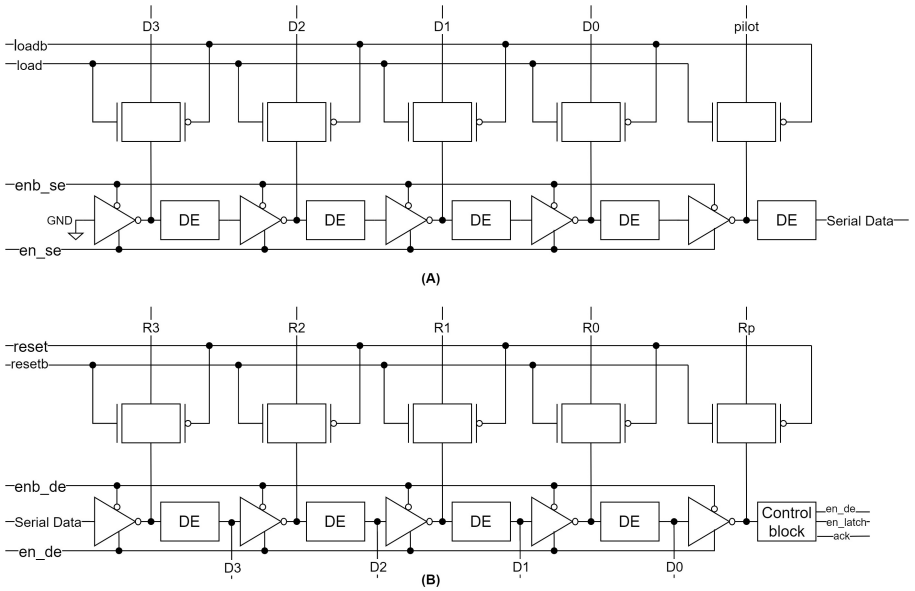

Figure 2.7: A: The serializer of the Wave-pipelined SerDes. B: The deserializer of a Wave-Pipelined SerDes

2.9.3 Double-Edge Triggered Flip-Flop Based Designs

Multiple designs utilize Double edge triggered flop-flops (DETFF). An DETFF, which can be seen in Figure 2.8, is a flip-flop design, consisting of two flip-flops, one positive edge triggered and one negative edge triggered, and a multiplexer. The DETFF is capable of sampling data on both the rising and the falling edges of the clock. This takes advantage of the second part of the clock cycle, which a normal single edge-triggered flip-flop does not, effectively doubling the efficiency. This can be used to make an efficient SerDes.

The serializer of one of these designs, which can be seen in Figure 2.9, is made by Jaiswal and Gamad [11]. The design is made using seven DETFFs and three clock dividers. The design works by utilizing time multiplexing to send each data bit on the serial line in turn. DETFF A is clocked using the system clock divided by two, the two DETFF B are clocked using the system clock divided by four, and the four DETFF C flip-flops are clocked by the system clock divided by eight. This makes it so that DETFF A switches once per positive edge of the system clock, DETFF B every second positive edge of the system clock, and DETFF C every fourth positive edge of the system clock. The pattern of the serialized data is indicated by the numbering on the parallel input lines from D0 to D7, in Figure 2.9.

There are some problems related to scan chains with a design that utilizes both rising and falling edges. If the rising-edge flip-flops sample the value before the falling-edge flip-flops, the scan chain loses coverage. The reason for this is that falling-edge flip-flops sample the same

value that the rising-edge flip-flops just sampled, making it so that the two flip-flops hold the same value. To combat this the falling-edge flip-flops must be placed before the rising-edge flip-flops. But it might not be viable to place the falling-edge flip-flops at the start of the scan chain in a large system. Another way to combat the problem is to place lockup latches between the rising-edge flip-flops and the falling-edge flip-flops. This adds to the complexity of the scan chain, and the system can, for large designs, end up with a high number of these latches.

For a DETFF design, none of these solutions can be used as the two flip-flops are placed side-by-side in parallel. And this leads to hold time violations in the scan chain path. While some solutions to this problem exist, these solutions either consist of inserting an extra delay between the two flip-flops or inserting hold buffers to space out the two flip-flops in time. The hold buffers wastes area, power, and performance, while still not completely guaranteeing that hold time violations do not occur.

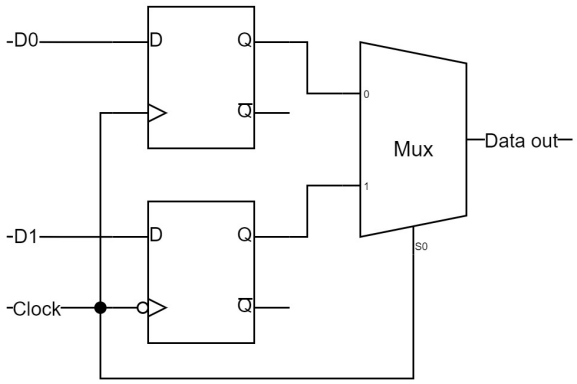

Figure 2.8: A double edge triggered flip-flop design

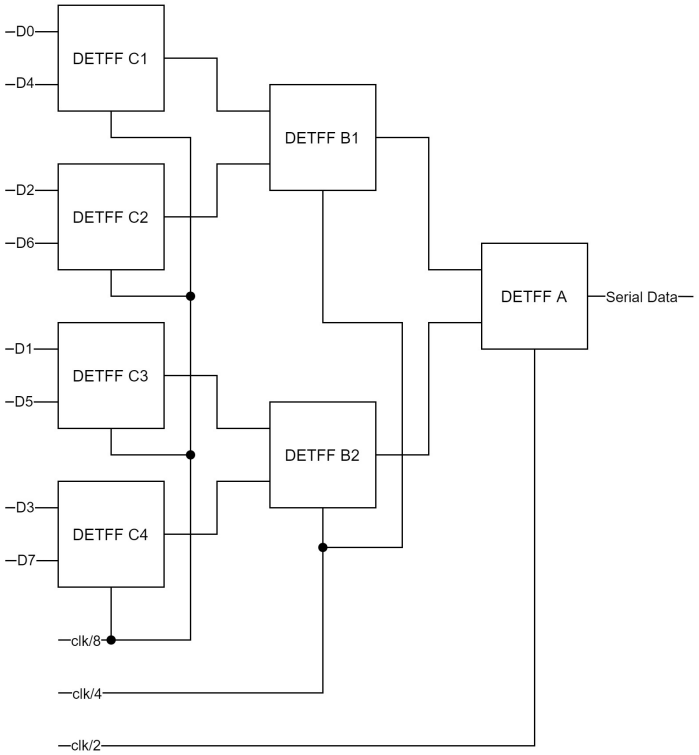

Figure 2.9: implementation of a double edge triggered Serializer-Deserializer

Chapter 3

Implementation

3.1 Proposed Solution

The proposed solution is a scalable four-to-one serializer and a one-to-four deserializer that utilizes multiphased clocking for both serialization and deserialization. Each unit is capable of transmitting 4 bits of data for each clock cycle, which makes it possible for a reduction of $\frac{3 \cdot n}{4}$ data lines at the same performance, where n is the number of data lines in the bus. An overview of the solution can be seen in Figure 3.1. The solution consists of a serializer and deserializer. In addition, a multiphase synchronizer is connected to the output of the deserializer. Four clocks of the same frequency with phases of 0 degrees (clk0), 90 degrees (clk90), 180 degrees (clk180), and 270 degrees (clk270) are utilized, as in the example in Section 2.2. Here, it is assumed that the clocks are available across the chip. A simple handshake system consisting of a Serializer to Deserializer Valid signal (SerDesValid) and a Deserializer to Serializer Ready signal (DesSerReady) is used between the serializer and deserializer. One transfer consists of 4 bits of data and is completed in three clock cycles of clk0, with the possibility of a new transfer for each clock cycle. The multiphase synchronizer is used to synchronize the data to clk0 once the data is deserialized, in addition to realigning the data. The design is capable of backpressure, which means that the transmitter has data to send, but the receiver can not receive it. When this happens, the design waits until the receiver can start to receive. An example of this is shown in Section 4.3.

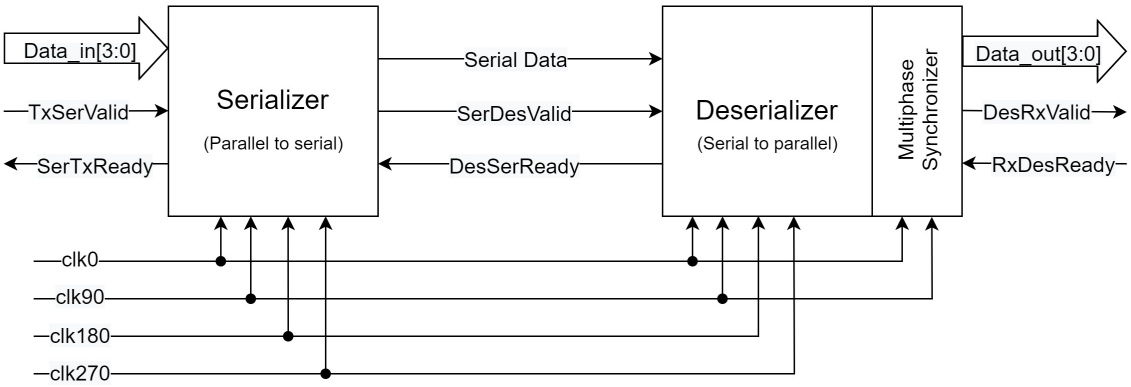

Figure 3.1: Overview of the purposed solution

3.2 Data Path

3.2.1 Serializer

Figure 3.2 shows the serializer of the design. The serializer has four parallel input lines, named D0 to D3, and a serial output line called Serial data. In XOR 1, XOR 2, and XOR 3, the parallel data, D0, D1, and D2, are either inverted or stay the same based on a feedback value from XOR 8, XOR 9, and Reg G, respectively. These values are stored in the three registers, Reg A to C, and D3 is stored directly in Reg D. clk0 is used for all four of these registers. The first bit in Reg A is either inverted back or remains the same based on the same feedback signal as XOR 1 before it is transferred on the serial line. The output of Reg B is either inverted or not based on the output of Reg A, in XOR 4. After $\frac{1}{4}$th of a clock cycle of clk0, at the rising edge of clk90, the output of XOR 4 is clocked into Reg E. The output of Reg E is either inverted again or not, based on the feedback value from XOR 9, in XOR 8. In XOR 7, the value is again either inverted back or not based on the output of Reg A. Which bit is transferred on the serial line is based on the rising edge of the four clocks, used for Reg A and Reg E to G. This will be expanded upon in Section 3.4

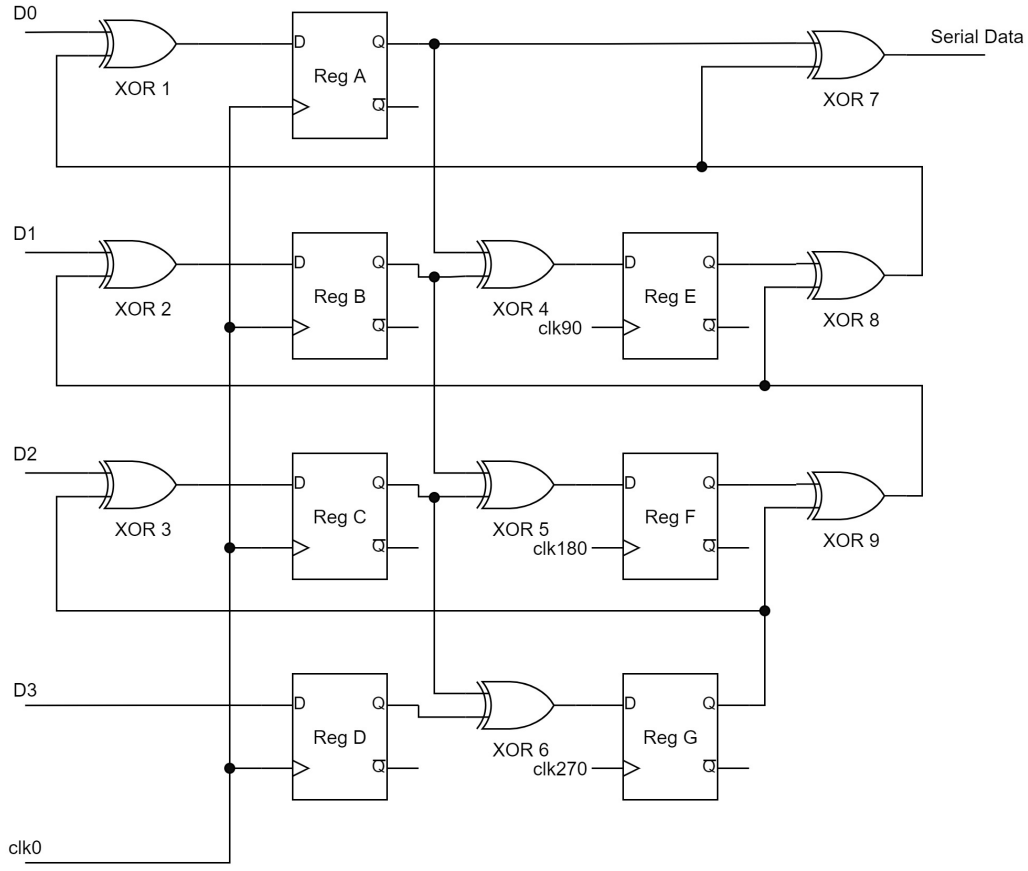

Figure 3.2: Data path for the serializer

XOR 2 and XOR 8 can be seen as a pair for the second bit, where if the second bit is inverted in XOR 2, it is inverted back in XOR 8. A pair can also be defined as two XOR-gates that share an input, but that does not mean that two XOR-gates can not be pairs if they do not share an input. With this in mind, the first bit goes through one pair of XOR-gates, and the second bit goes through two pairs of XOR-gates before it reaches the serial line. Note that

this way of looking at it does not work for the third and fourth bits. This is because XOR 5 and XOR 6 is not directly paired with any of the other XOR gates but rather a compilation of the registers that are clocked by clk0 above it. An example of this is the path of the third bit, D2. The bit is either inverted or not based on the output of Reg B in XOR 5, and either inverted back or not, in XOR 8. Before it is either inverted or not in XOR 7, based on the Reg A value. This means that if both reg A and B are high, Bit three is inverted in XOR 5 but not in XOR 8 before it is inverted back in XOR 7. This would make XOR 5 and XOR 7 a pair. If the value in Reg A is high and Reg B is low, The third bit is not inverted in XOR 5 but is inverted in XOR 8 and inverted back in XOR 7, making XOR 7 and XOR 8 a pair. The same reasoning works for D3 with one added XOR-gate, XOR 9.

A reset signal is not needed since the value stored in Reg A to C is dependant on the previous bits stored in Reg E to G. The serializer can receive any input with no regard for what is in the registers.

3.2.2 Deserializer and Multiphase Synchronizer

Figure 3.3 shows the deserializer and the multiphase synchronizer. The deserializer part of the design consists of Reg A to D, while the multiphase synchronizer consists of the remaining seven registers. The input of all four of the registers in the deserializer is connected to the serial line, and they are also clocked by a different phased clock. It should be noted that the first bit on the serial line, D0, is sampled by Reg A which is clocked by clk90, compared to the serializer where D0 is sent using clk0. This is done to account for the delay caused by the data being sent on the bus. This might need to be changed based on the length of the bus. For example, if the bus is long, the first bit might have to be sampled using clk180 instead. See Section 3.4 for more on this.

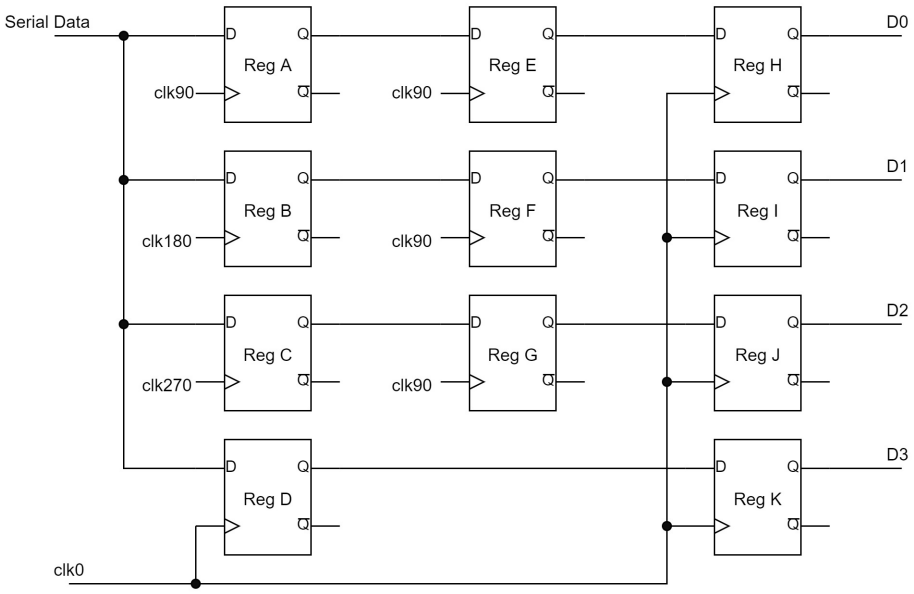

Figure 3.3: Data path for the deserializer and the multiphase synchronizer

After the data has been sampled by the deserializer, the four bits are all in different phases. To get them all back to the same phase as the rest of the system, the multiphase synchronizer is used. Reg E to G is used to delay the three bits D0, D1, and D2 so that D3 is realigned, compared to the three other bits. If these three registers had not been there, the value for D3 would not have the time to update before it should be ready on the output,

making it so that the old value from the previous transfer would be sampled instead of the new value. This will be expanded upon in Section 3.4. Lastly, the four bits are clocked into registers that are clocked using clk0, making sure that the data is in the same clocking phase like the rest of the system.

3.3 Control Signals

The control signals consist of the three sets of valid and ready signals that can be seen in Figure 3.1. Additionally, there are enable signals for clock gating in both the serializer and deserializer. A Finite State Machine (FSM) is used for the control signals in the serializer.

3.3.1 Handshake

The system uses a simple handshake protocol, consisting of a valid signal that indicates that there is valid data to be sent and a ready signal that indicates that data is ready to be received. A handshake is defined as the first clock cycle that both valid and ready is high at the same time, and a beat is defined as a clock cycle where both valid and ready is high, and data is transferred. A transfer is a single data sending of one beat, while a transaction is a burst of transfers. The system is based on the AMBA AXI handshake protocol [12]. There are six rules that the handshake must follow:

- The valid signal can stay high and data will continuously be sent if the ready signal is also high.
- If the valid signal is set low, the transaction is finished.
- A source must not wait for the ready signal to be set high to set the valid signal high.
- If valid is set high, it can not be set low until a handshake has occurred.
- If the ready signal is set low, the current transfer must be finished if one is taking place.
- If ready is set low, the data for the next transfer must be held in the transmitter until the ready signal is set high again.

Nothing should happen if not both valid and ready are high, and the focus should then be to use as little power as possible while still being able to respond to a change promptly.

3.3.2 Control Signals for the Serializer

The FSM for the serializer, which can be seen in Figure 3.4, consists of 3 states: idle, wait, and transfer. The inputs for the FSM are the DesSerReady and the Transmitter to Serializer Valid signal (TxSerValid). The outputs are the SerDesValid signal and an internal enable signal used to enable the clock gating in the serializer. The enable signal will be described in more detail later in this section. The idle state is for when the TxSerValid signal is set low. Clock gating is used to minimize the power while the serializer waits for transmission to start. When the TxSerValid signal is high, SerDesValid is immediately set high, and a check is done as to whether the DesSerReady signal is high. If it is, the next state is transfer. If not, the next state is wait. In the wait state, only a check is done as to whether the DesSerReady signal is high. If it remains low, the FSM should stay in the wait state. Else, the next state is transfer. Note that in the wait state, the FSM is only allowed to either remain in this state or move on to the transfer state. The reason for this is the fourth rule for the handshake signals, which states that if the valid signal is set high, it can not be set low until a handshake occurs. Similar to the idle state, the wait state also utilizes clock gating to minimize power consumption.

The last state is transfer. The FSM can only be in this state if both the SerDesValid signal and the DesSerReady signal are high. The FSM stays in this state until either the TxSerValid signal is set low, which indicates that the transfer is completed and the FSM should return to the idle state, or until the Receiver to Deserializer Ready signal (RxDesReady) is set low, at which the FSM returns to the wait state where it stays until the RxDesReady signal is set high again.

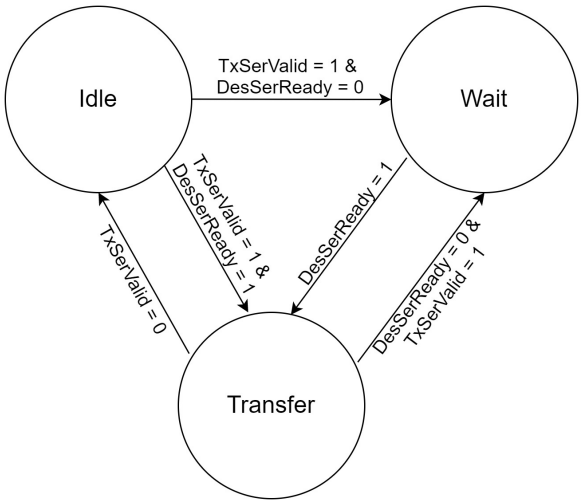

Figure 3.4: State Diagram for the Finite State Machine

The enable signal for clock gating is split into two signals. The first enables the four D flip-flops that are clocked by clk0, while the second enables the three phase-shifted registers Reg E, F, G. This is done so that the serializer can hold data in the four registers clocked by clk0 while in the wait state without transmitting the data on the serial data line, and by that, losing the data.

3.3.3 Control Signals for the Deserializer

The control signals for the deserializer consist of the DesSerReady signal and the Deserializer to Receiver Valid signal (DesRxValid), in addition to an enable signal for the registers in the deserializer. The RxDesReady signal is connected directly to the DesSerReady signal. This is done to increase the response time in situations where there is already valid data to be sent from the source. The SerDesValid signal is connected to the DesRxValid signal through a register that is clocked by clk0. This creates a clock cycle delay of one, which makes time for the deserializer to have the data ready when the receiver expects it.

The enable signal for the registers in the deserializer is made up of the SerDesValid signal and the DesSerReady signal, with one clock cycle delay. This will be expanded upon in the next section.

3.4 Timing

In this section, the timing of the serializer, deserializer, and between them is elaborated. In addition, the motivation for the timing of certain key control signals is looked at.

The timing of serialization of a data byte can be seen in Figure 3.5. The four bits of a transfer corresponds to the rising edge of one of the four clocks, e.g., D0 is transmitted using clk0, and D1 is transmitted using clk90. Each bit is available on the serial line for a period of $\frac{1}{4}$ of a clock cycle of clk0 before the next bit is transmitted, and the following transfer starts at the positive edge of the next clock cycle of clk0.

It would not be possible to sample the serial line in the deserializer at the same clock edge that the signal is transferred to the line from the serializer. In addition to this, there is setup and hold time to account for. This is the reason why the deserializer samples the values 90 degrees shifted compared to the serializer. The shifting has, in addition to the delay on the line, the added benefit of moving the signal's sampling window. This makes the design more robust, with more time for transferring data on the bus, before sampling in the deserializer.

In addition to this, it makes setup and hold time more comfortable. Figure 3.6 shows the timing for how the deserializer samples the value compared to the four phase-shifted clocks. The bits are shifted about $\frac{1}{4}$ of a clock cycle as the data is transmitted on the line. With this figure, it becomes more apparent that the value of D3, which is marked in red, becomes a problem without the added layer of registers for bit zero to three as shown in Figure 3.3 with Reg E to G. If these registers had not been there, transfer 2 would hold the D3 value of transfer 1, at the output of the multiphase synchronizer, instead of the D3 value that appears at the next rising edge of clk0.

Synchronization of control signals between the serializer and deserializer is done using registers to delay signals used for the three handshakes. An example of this is the RxDesReady signal. The signal is received in the deserializer from the transmitter and directly connected to the DesSerReady signal. In the serializer, the DesSerReady signal is connected to Serializer to Transmitter Ready signal (SerTxReady) through a register, to delay it. This lets the serializer react to the DesSerReady signal by changing state in the FSM, and enable Reg A to D before the SerTxReady signal is set high, if the TxSerValid is already high.

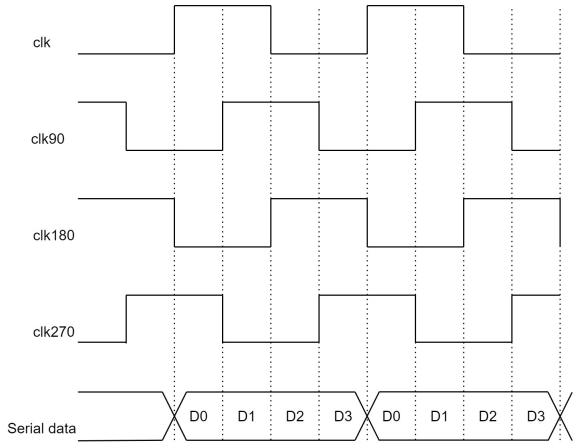

Figure 3.5: Timing for the serializer

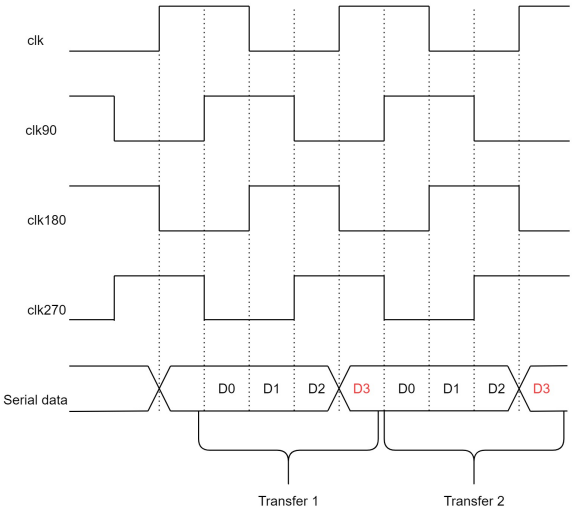

Figure 3.6: Timing for the deserializer

The TxSerValid signal is delayed in the serializer by a clock cycle before it is sent to the deserializer as the SerDesValid signal. This is done so that the serializer has enough time to both; check if the DesSerReady signal has been set high and start the transfer. The reason for this is that the serializer takes a clock cycle before the first bit is ready on the serial data line before the deserializer is enabled. In the deserializer, the SerDesValid signal is delayed by two clock cycles before it is sent to the receiver as the DesRxValid signal. This is done to account for the two clock cycles the multiphase synchronizer spends aligning the data to clk0.

Chapter 4

Results and Discussion

In this chapter, the results for the proposed solution are presented and discussed. This includes area utilization, performance, and power consumption, in addition to further discussion about the proposed solution.

4.1 Methodology

The implementation was made using Xilinx Vivado design suite. The multiphase clocking was made using the MMCM in the clock wizard IP integrated with Vivado. The implementation was split into several modules called: Serializer, Serializer Control, Deserializer, and Deserializer Control. The code was written in the Hardware Description Language (HDL) SystemVerilog and tested using the integrated simulation tools in Vivado. A simple testbench was made that made semi-random input data, in addition to some different scenarios for TxSerValid and RxDesReady signals. The SystemVerilog code can be found in Appendix B.

4.2 Start and Ending of a Transfer

In Figure 4.1, and Appendix A for a bigger version, an example of both the start and end of a transfer can be seen. In this example, *data_in* is the parallel data into the design, while *data_out* is the parallel data out after serialization. The ending of a data transfer is indicated by TxSerValid being set low. The last data value that is sent is decided based on the last data value that is on the input while TxSerValid is still high at the positive edge of clk0, which in this case is "1100". After this, the output does not change until the start of the next transfer. Before the TxSerValid signal is asserted right before 760 ns, the input changes to ”1010”, which is the new data to be sent, before the TxSerValid signal is set high. This indicates that there is new valid and stable data on the input, and a new transfer can happen. As the RxDesReady signal is high, a new transaction can take place.

Something to note is that the data takes three clock cycles to get from the input to the output. This is mainly caused by the multiphase synchronization step, as it takes two clock cycles for the data to get through it. Another thing to note is that, while in a transaction, new data can be sent for each clock cycle of clk0.

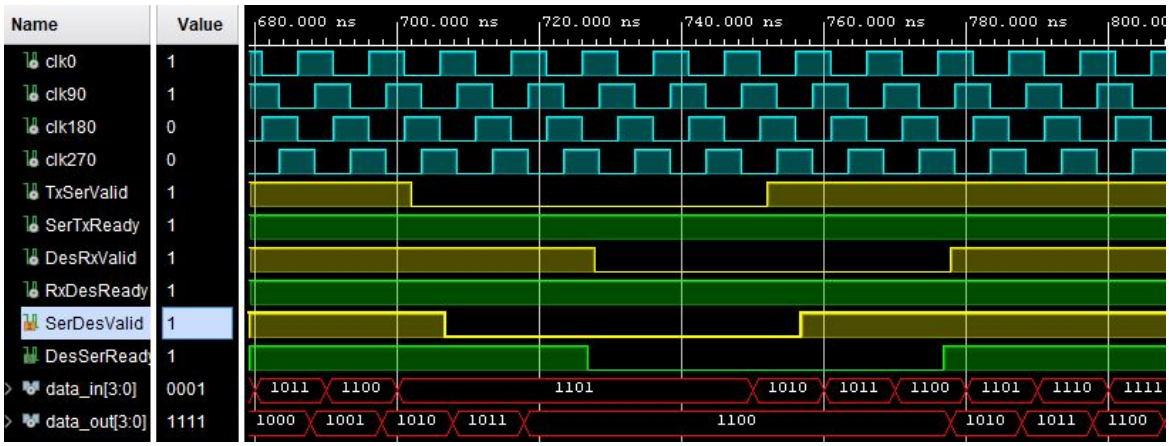

Figure 4.1: An example of both a start and ending of a transfer

4.3 Backpressure

In Figure 4.2, and in Appendix A for a bigger version, an example of backpressure can be seen. Backpressure occurs when the RxDesReady signal is set low while the TxSerValid signal remains high. This leads to both the serializer and the source holding the data, waiting for the receiver to be ready to receive data again. Which transfer is held in the wait state is determined by the first rising edge of clk0 that the RxDesReady signal is low. In this case, the value is ”0100”. The data is held until the RxDesReady signal is set high again. After which, it takes three clock cycles of clk0 for the signal to be ready on the output of the multiphase synchronizer.

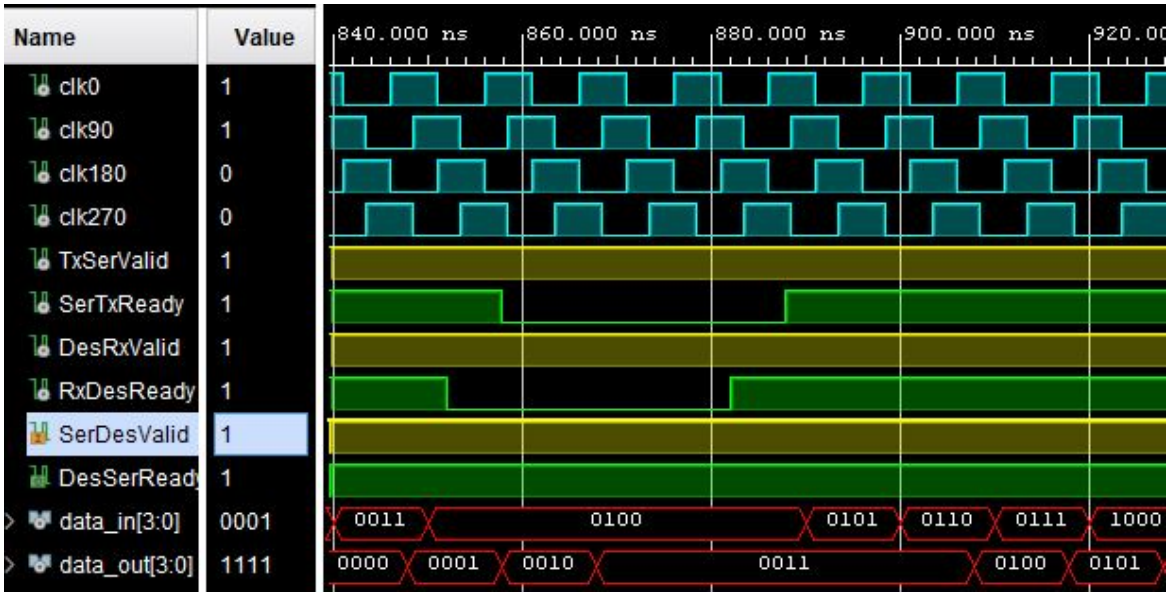

Figure 4.2: An example of backpressure

It should be noted that the output values of ”0010” and ”0011” are transferred after RxDesReady signal is set low. This is in line with rule number five in Section 3.3.1, in addition to the fact that any given signal requires three clock cycles of clk0 to get from the input of the serializer to the output of the multiphase synchronizer.

4.4 Performance and Area

The proposed solution has a high throughput, transmitting 4 bits every clock cycle. There is a drawback when compared to a fully parallel solution. The added circuitry causes data to take three clock cycles to reach the output at the start of a transmission, as mentioned in 4.3. If the number of bytes transferred is high, the control signals become negligible, and the proposed solution holds the same performance as a fully parallel solution. If the proposed solution is used between the core and the cache in a GPU, this is not an unrealistic assumption.

The proposed solution consists of seven flip-flops and nine XOR-gates for the serializer, eleven flip-flops for the deserializer, in addition to the circuitry needed for the control signals. The implemented design does take up more area when compared to a fully parallel solution. That is if the removed wires and repeaters from the serial solution are not taken into account, but only the flip-flops and XOR-gates. For a fully parallel design that is 64 bit wide, there are 128 flip-flops combined for both the transferring and receiving sides. For the proposed solution with the same width, there are 288 flip-flops and 144 XOR-gates. The proposed solution takes an estimated total of 2063.5 GE, while a fully parallel solution takes an estimated total of 768 GE. This makes the proposed solution 269 %, or 2.7 times, larger than a fully parallel solution. Note, again, that this is without the reduction of the area, due to the wires and buffers of the bus, and this is expected to have a major impact on the reduction of the area. The author of this report did not have access to tools that could give an estimate of the impact of the reduced number of wires and buffers on the area, for this project.

4.5 Power consumption

The power consumption of the design was estimated using power analysis with 7 nm technology. These numbers are normalized as the frequency the analysis was run at can not be shared due to confidentiality. The test design is a 64-bit wide solution, which means the solution consists of 16 units, and the control and data paths were separated. The Power analysis was conducted by switching the input of the design between every even bit and every odd bit set high, or between $0x55555555$ and $0xAAAAAAAA$, for each clock cycle of clk0.

The total power consumption for the serializer is an average of 2.28 μW for each unit. Of this 9.66% is leakage power, and the remaining 90.34 % is dynamic power. The total power consumption for all 16 serializer units is 36.48 μW. For the deserializer, the average is 2.06 μW total power for each unit. Of the total power consumption 11.01 % is leakage power and 88.99 % is dynamic power. The total power consumption for all the 16 deserializer units is 32.89 μW. The control signals for both the serializer and deserializer contributes with 1.21 μW and 0.28 μW, respectively, making the total power consumption of the system 70.75 μW, excluding the data bus and, with that, the data bus buffers.

An interesting point is that the serializer has higher dynamic power consumption when compared to the deserializer, even though the serializer contains fewer flip-flops than the deserializer. The reason for this, as explained in Section 2.4, is due to the higher switching activity in XOR-gates. The deserializer has slightly higher leakage power due to the extra flip-flops when compared to the serializer.

A parallel solution was also implemented, and the power consumption was tested using the same power analysis with the same technology. The total power consumption for this solution is 21.42 μW, with half of it coming from the transmitter and the other half from the receiver. It can be seen based on the total power consumption of these two solutions that the

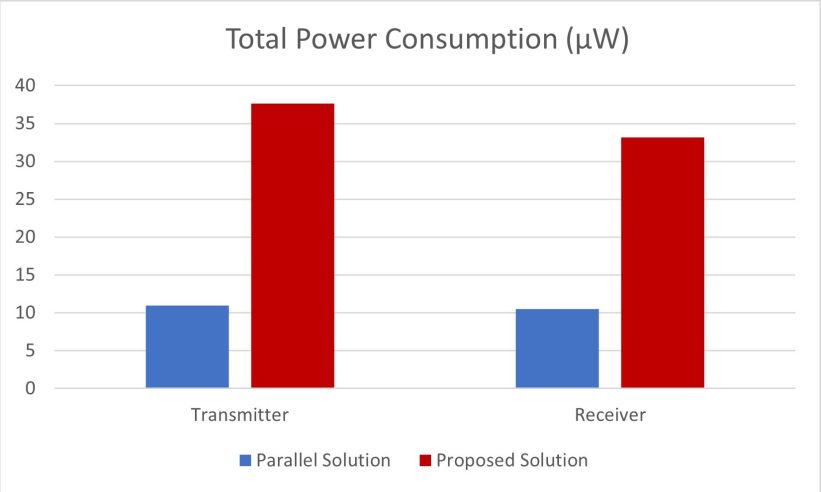

Figure 4.3: Comparison of total power consumption between the proposed solution and a parallel solution

proposed solution utilizes 330 %, or 3.3 times more power than a parallel solution. While this might sound like a lot initially, it is important to remember that either solution would be a small part of the system and a small part of the total power consumption of the system. The freed-up area from the reduction of wires can be significant. These numbers do not take into account the reduction of buffers on the bus, in addition to the lower power consumption due to the reduction of the number of wires in the bus. The reason for this is that the author did not have access to analyzing tools for this.

Table 4.1: Power consumption for the clock trees in both the proposed solution and the parallel solution

Clock	Power Consumption (μW)
Proposed Solution	
clk0	13.36
clk90	6.73
clk180	2.94
clk270	2.94
Total	25.97
Parallel Solution	
clk0	11.19

There is also an additional impact on the power consumption from the added clock trees. The power consumption for both the proposed solution and the parallel solution can be seen in Table 4.1. For the proposed solution, clk0 has the highest power consumption due to the number of flip-flops that are driven by it. clk0 would have the same power consumption as the parallel clock if not for the additional flip-flops in the control logic. A point to note is that clk90 has significantly higher power consumption than clk180 and clk270. This is due to the three extra flip-flops in the multiphase synchronizer, Reg E to G which can be seen in Figure 3.3, that is driven by clk90. The total power consumption for the proposed solution is 25.97 μW. That is 232 %, or 2.32 times, higher than the parallel solution that utilizes a single clock.

4.6 Further Discussion

While it is possible to increase the number of clocks to improve the solution further, there are some reasons not to do it. A large part of the power consumption comes from the different clock trees, and it might not be worth the extra performance if the number of clocks is too high. For each new clock added, the time each bit is available, to be sampled, on the serial line is decreased. If there are too many clocks, the clock frequency would have to be reduced, and the performance of the bus would be lower. Note that this has not been tested in this project, and would be worth testing in a future project.

The critical path of the serializer starts at Reg F and Reg G and ends at the first repeater on the serial line. On the path, there are three XOR-gates, XOR 9, XOR 8, and XOR 7. There is a difference in time for the four bits to reach the serial line. As can be seen in Figure 3.2, the first bit only has to go through XOR 7, while the second bit has to go through XOR 8 and XOR 7, and bit three and four has to go through XOR 9, XOR 8, and XOR 7. This leads to a slight skew in how long each bit is available on the serial line. This would only be a problem if the clock frequency for the four clocks is high. If it were to become a problem, it might be possible to reduce the critical path for bits three and four by setting XOR 8 and XOR 9 in parallel, but this has not been tested.

Scan chains are much simpler to implement for the proposed solution compared to a DETFF solution. This is because the proposed solution utilizes four separate clocks instead of using both edges of a single clock. One scan chain per clock can be implemented, with no danger of timing violations as with the DETFF solution, as mentioned in Section 2.9.3.

One interesting point about this solution is the possibility of sending data to multiple cores simultaneously. It would, for example, be possible to send every even bit to one core and every odd bit to another core by using clock gating for the individual receiving registers in the deserializer. While this might be useful in some cases, it is more of an afterthought, and the solution is not designed for this. In addition, if the design were to be utilized like this, it would mean that the total bandwidth would be split between multiple transfers.

Something to consider is the number of flip-flops clocked using the same sub-clock. Sub-clock means a branch of the main clock behind a single clock gate. If there are too few, the power consumption of the branch for that sub-clock is higher than the power saved by clock gating. But if the number of flip-flops under the same clock gate is too high. The fan-out is high, and there are either reliability issues with both timing and voltage levels, or clock buffers that increase the propagation delay of the clock have to be used, which can also cause timing issues. Considering a single serializer unit, as shown in 3.2, with four flip-flops clocked by clk0, and the remaining three registers are clocked by clk90, clk180, and clk270, respectively. If clock gating for every serializer unit were to be used, each of the phase-shifted clocks, clk90, clk180, and clk270, would have clock gating for a single flip-flop making the overhead from the clock gating itself too high compared to the reduced power consumption. If a design consisting of, for example, 256 units where all the units share a single clock gate, there would be 1024 flip-flops clocked by clk0, and the fan-out might be too high. Therefore, a balance between the two might be four units, with 16 flip-flops clocked by clk0 and four flip-flops clocked by clk90, clk180, and clk270 each. This is most often handled by synthesis tools when a design is implemented on an FPGA. But as a part of a larger design, this is a trade-off that the designer must consider.

Chapter 5

Conclusion

This report has outlined the process of implementing a Multiphase Clocked Serializer-Deserializer, as a solution to reduce the number of wires on interconnect buses. The proposed solution was made using SystemVerilog and tested using the integrated simulator in Xilinx Vivado. The simulations verified that the solution works as intended. The solution utilizes multiphase clocking, meaning multiple clocks of the same frequency with different phases. The proposed solution is capable of 4-to-1 serialization and deserialization, which means a four-times reduction in the number of wires in the bus. The solution offers high throughput with four bits for each clock cycle and is capable of backpressure, meaning that the transmitter has valid data to be sent that the receiver is not ready to receive. The sum of the area for the proposed solution is 2.7 times larger than a fully parallel solution, excluding the area for wires and buffers, which is significantly smaller for the proposed solution. The power consumption for the proposed solution is 3.3 times higher than a parallel solution, excluding the power consumption due to capacitance on the wires and the power consumption of the buffers. The proposed solution is inherently capable of sending data to multiple receivers at the same time, with a reduction in bandwidth to each of the receivers.

This report serves as a good starting point for a project with a larger analysis. And given the possibility of exploring the impact on the area, power consumption, and performance by the wires and buffers, This project could be used to broaden the understanding of how the wires in interconnect buses can best be reduced, not only for the proposed solution but even existing solutions, or potential other new solutions.

Chapter 6

Future Work

Due to limitations in time and resources, it was not possible to do everything the author wanted to in this project. The design should be implemented on an Field-Programmable Gate Array (FPGA) as a way to verify that it works as intended. Some of the other solutions, namely the WP SerDes, and the Double-edge triggered SerDes, could be implemented for comparison. An analysis of the power consumption and area due to the reduction of wires, and buffers, should also be conducted. It would be worth experimenting with the number of clocks that are used for the proposed solutions to determine if there is a trade-off.

Optimization of the implemented solution is also a possibility. One example of this is reducing the critical path from Reg E to the serial line in the serializer by reducing the number of XOR-gates, from three to two, by setting two of the XOR-gates in parallel as mentioned in Section 4.6. Another example that was not tested in this project is the possibility of removing Reg D, E, F, and G in the deserializer and sending the fourth bit directly to Reg K from the serial line.

Bibliography

[1] P. Gepner and M. Kowalik, "Multi-Core Processors: New Way to Achieve High System Performance," in *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, pp. 9–13, Sept. 2006.

[2] P. Saxena, R. S. Shelar, and S. Sapatnekar, *Routing Congestion in VLSI Circuits: Estimation and Optimization*. Springer Science & Business Media, Apr. 2007. Google-Books-ID: whL4ipDg_AgC.

[3] J. S. Clarke, C. George, C. Jezewski, A. M. Caro, D. Michalak, and J. Torres, "Process technology scaling in an increasingly interconnect dominated world," in *2014 Symposium on VLSI Technology (VLSI-Technology): Digest of Technical Papers*, pp. 1–2, June 2014. ISSN: 2158-9682.

[4] R. R. Dobkin, A. Morgenshtein, A. Kolodny, and R. Ginosar, "Parallel vs. serial on-chip communication," in *Proceedings of the tenth international workshop on System level interconnect prediction - SLIP '08*, (Newcastle, United Kingdom), p. 43, ACM Press, 2008.

[5] Y. Ye, K. Roy, and R. Drechsler, "Power consumption in XOR-based circuits," in *Proceedings of the ASP-DAC '99 Asia and South Pacific Design Automation Conference 1999 (Cat. No.99EX198)*, pp. 299–302 vol.1, Jan. 1999.

[6] H. Kaeslin, *Digital Integrated Circuit Design From VLSI Architectures to CMOS Fabrication*. Cambridge, May 2008.

[7] Q. Wu, M. Pedram, and X. Wu, "Clock-gating and its application to low power design of sequential circuits," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 47, pp. 415–420, Mar. 2000. Conference Name: IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications.

[8] J. Aerts and E. Marinissen, "Scan chain design for test time reduction in core-based ICs," in *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*, pp. 448–457, Oct. 1998. ISSN: 1089-3539.

[9] E. Radon, "MMCM and PLL Dynamic Reconfiguration, `https://www.eeweb.com/mmcm-and-pll-dynamic-reconfiguration/`, urldate = 2021-09-21," Aug. 2012.

[10] B. C. Hien, S.-M. Kim, and K. Cho, "Design of a wave-pipelined serializer-deserializer with an asynchronous protocol for high speed interfaces," in *2012 4th Asia Symposium on Quality Electronic Design (ASQED)*, pp. 265–268, July 2012.

[11] N. Jaiswal and R. Gamad, "Design of a New Serializer and Deserializer Architecture for On-Chip SerDes Transceivers," *Circuits and Systems*, vol. 06, no. 03, pp. 81–92, 2015.

[12] "An introduction to AMBA AXI, https://developer.arm.com/documentation/102202/0200/Channel-transfers-and-transactions, urldate = 2021-12-19."

Appendix A

Large Figures

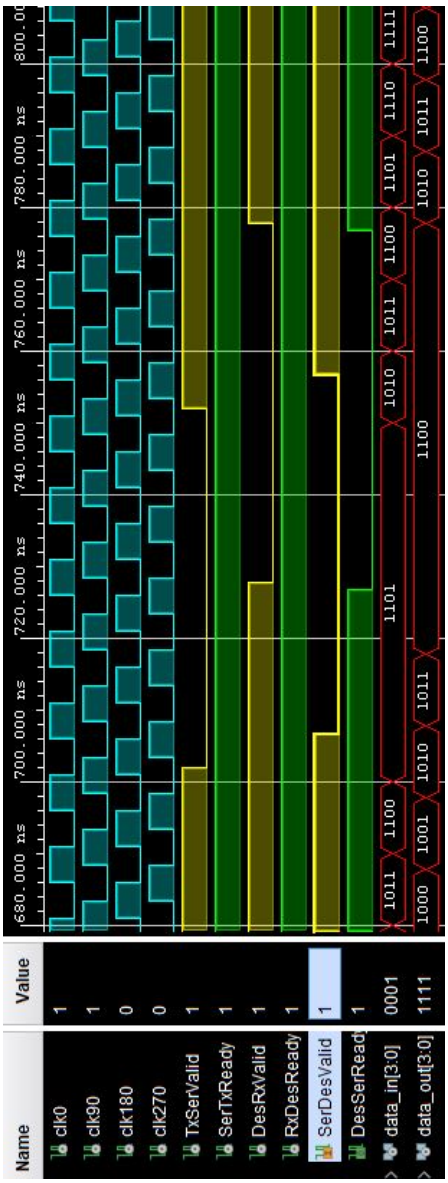

Figure A.1: An example of both a start and ending of a transfer (Large version)

Figure A.2: An example of backpressure (Large version)

Appendix B

Source Code

B.1 Top Module

```
module Top
  #(
    parameter PSIZE = 64,
    parameter SSIZE = 16
    )
        (
    input clk,
    input TxSerValid,
    output SerTxReady,
    input RxDesReady,
    output DesRxValid,
    input [63:0] dataIn,
    output [63:0] dataOut
    );

    wire clk0, clk90, clk180, clk270;
    logic [15:0] SerBufData;
        logic [15:0] BufDesData;
    wire SerEnable, DesEnable;
    wire SerDesValid, DesSerReady;
    wire EnablePhaseSer, EnablePhaseDes;

        Multiphase_clock
                Multiphase_clock (.clk(clk), .clk0(clk0), .clk90(clk90),
                ↪  .clk180(clk180), .clk270(clk270));

    Serializer_Control
                Ser_Control(.clk0(clk0), .TxSerValid(TxSerValid),
                ↪  .SerTxReady(SerTxReady), .DesSerReady(DesSerReady),
                ↪  .SerDesValid(SerDesValid), .enable(SerEnable),
                ↪  .EnablePhase(EnablePhaseSer));

    for (genvar g_i = 0; g_i < SSIZE; g_i++)
        begin:
```

```
                Serializer
                        Serializer1 (.clk0(clk0), .clk90(clk90),
                        ↪  .clk180(clk180), .clk270(clk270),
                        ↪  .DataIn(DataIn[4*g_i+:4]), .enable(SerEnable),
                        ↪  .EnablePhase(EnablePhaseSer),
                        ↪  .SerData(SerBufData[g_i]));
        end

    Deserializer_Control
                Des_Control(.clk0(clk0), .DesRxValid(DesRxValid),
                ↪  .RxDesReady(RxDesReady), .SerDesValid(SerDesValid),
                ↪  .DesSerReady(DesSerReady), .enable(DesEnable),
                ↪  .enable_phase(EnablePhaseDes));

        for (genvar g_i = 0; g_i < SSIZE; g_i++)
        begin:
                Deserializer
                        Deserializer1 (.clk0(clk0), .clk90(clk90),
                        ↪  .clk180(clk180), .clk270(clk270),
                        ↪  .SerData(BufDesData[g_i]),
                        ↪  .DataOut(DataOut[4*g_i+:4]), .enable(DesEnable),
                        ↪  .EnablePhase(EnablePhaseDes));
        end

        for (genvar g_i = 0; g_i < SSIZE; g_i++)
        begin:
                Serial_buffer
                        serial_buffer (.serial_data(SerBufData[g_i]),
                        ↪  .serial_data1(BufDesData[g_i]));
        end

endmodule
```

B.2 Serializer Module

```
module Serializer(
    input clk0, clk90, clk180, clk270,
    input [3:0] DataIn,
    input enable,
    input EnablePhase,
    output reg SerData
    );

    reg D_c, D_a, D_b, D_e, D_f, D_g, internal1, internal2;
    reg Q_a, Q_b, Q_c, Q_d;
    reg Q_e = 1'b0;
    reg Q_f = 1'b0;
    reg Q_g = 1'b0;

    // Registers
    always @ (posedge clk0)
    begin : Reg_A
        if (enable == 1)
            Q_a <= D_a;
    end

        always @ (posedge clk0)
    begin : Reg_B
        if (enable == 1)
            Q_b <= D_b;
    end

        always @ (posedge clk0)
    begin : Reg_C
        if (enable == 1) begin
            Q_c <= D_c;
        end
    end

        always @ (posedge clk0)
    begin : Reg_D
        if (enable == 1) begin
            Q_d <= DataIn[3];
        end
    end

     always @ (posedge clk90) begin
        if (EnablePhase == 1) begin
            Q_e <= D_e;
        end
    end
```

```
     always @ (posedge clk180) begin
        if (EnablePhase == 1) begin
            Q_f <= D_f;
        end
    end

     always @ (posedge clk270) begin
        if (EnablePhase == 1) begin
            Q_g <= D_g;
        end
    end

    // XOR Gates
    always @(internal1, internal2, DataIn, Q_a, Q_b, Q_c, Q_d, Q_e, Q_f,
    ↪  Q_g)
    begin: XOR
        D_a <= DataIn[0] ^ internal1;
        D_e <= Q_a ^ Q_b;

        internal1 <= Q_e ^ internal2;
        D_b <= DataIn[1] ^ internal2;
        D_f <= Q_c ^ Q_b;

        D_c <= DataIn[2] ^ Q_g;
        D_g <= Q_d ^ Q_c;

        internal2 <= Q_f ^ Q_g;

        SerData <= Q_a ^ internal1;
    end
endmodule
```

B.3 Serializer Control Module

```
module Serializer_Control(
        input clk0,
        input TxSerValid,    // Valid from Transmitter
        output reg SerTxReady,  // Ready to Transmitter

        input DesSerReady,  // Ready from Deserializer
        output reg SerDesValid, // Valid to Deserializer



        output reg enable,      // enable for input registers in Serializer
        output reg EnablePhase // enable for phase-shifted registers in
        ↪  Serializer
    );

    reg enable_i, enable_i2, enable_i3;

    parameter S_IDLE = 2'b00, S_WAIT = 2'b01, S_TRANSFER = 2'b11;
    reg [1:0] curr_state = 2'b00;
    reg [1:0] last_state = 2'b00;

    always @ (posedge clk0)
    begin : Control
        enable_i2 <= enable_i3 & DesSerReady;
        SerTxReady <= DesSerReady;
    end

    always_comb
    begin : enable_control
        case(curr_state)
            S_IDLE:
                begin
                    enable <= TxSerValid;
                    EnablePhase <= 0;
                end
            S_WAIT:
                begin
                    enable <= (enable_i3 & DesSerReady) | enable_i |
                    ↪  enable_i2;
                    EnablePhase <= (enable_i3 & DesSerReady) | enable_i2;
                end
            S_TRANSFER:
                begin
                    if (last_state == S_IDLE)
                        begin
                           enable <= (TxSerValid & DesSerReady) | enable_i |
                           ↪  enable_i2;
```

```
                        EnablePhase <= (enable_i3 & DesSerReady) |
                        ↪  enable_i2;
                    end
                else
                    begin
                        enable <= (enable_i3 & DesSerReady) | enable_i |
                        ↪  enable_i2;
                        EnablePhase <= (enable_i3 & DesSerReady) |
                        ↪  enable_i2;
                    end
            end
         default:
            begin
                enable <= TxSerValid;
                EnablePhase <= 0;
            end
     endcase
end

always @(posedge clk0)
begin : FSM
    case (curr_state)
        S_IDLE:
            begin
                enable_i3 <= 0;
                if (TxSerValid == 1 & DesSerReady == 0)
                begin
                    SerDesValid <= 1;
                    enable_i <= 1;
                    last_state <= curr_state;
                    curr_state <= S_WAIT;
                end
                else if (TxSerValid == 1 & DesSerReady == 1)
                begin
                    SerDesValid <= 1;
                    enable_i <= 1;
                    enable_i3 <= 1;
                    last_state <= curr_state;
                    curr_state <= S_TRANSFER;
                end
                else
                    curr_state <= S_IDLE;
            end
        S_WAIT:
            begin
                enable_i <= 0;
                enable_i3 <= 0;
                if (DesSerReady == 1)
                begin
```

```
                    SerDesValid <= 1;
                    enable_i3 <= 1;
                    last_state <= curr_state;
                    curr_state <= S_TRANSFER;

                end
                else
                    curr_state <= S_WAIT;
            end
        S_TRANSFER:
            begin
                enable_i <= 0;
                if (DesSerReady == 0 & TxSerValid == 1)
                begin
                    enable_i3 <= 0;
                    last_state <= curr_state;
                    curr_state <= S_WAIT;
                end
                else if (TxSerValid == 0)
                begin
                    enable_i3 <= 0;
                    SerDesValid <= 0;
                    last_state <= curr_state;
                    curr_state <= S_IDLE;
                end
                else
                    curr_state <= S_TRANSFER;
            end
        default:
            begin
                curr_state <= S_IDLE;
            end
    endcase
end //End FSM

endmodule
```

B.4 Deserializer Module

```
module Deserializer(
        input clk0, clk90, clk180, clk270,
        input SerData,
        input enable,
        input EnablePhase,
        output reg [3:0] DataOut
    );

    reg Q_a, Q_b, Q_c, Q_d, Q_e, Q_f, Q_g;

    // Input registers
      always @ (posedge clk90)
    begin : Reg_A
        if (EnablePhase == 1)
            Q_a <= SerData;
    end

    always @ (posedge clk180)
    begin : Reg_B
        if (EnablePhase == 1)
            Q_b <= SerData;
    end

    always @ (posedge clk270)
    begin : Reg_C
        if (EnablePhase == 1)
            Q_c <= SerData;
    end

    always @ (posedge clk0)
    begin : Reg_D
        if (EnablePhase == 1)
            Q_d <= SerData;
    end

    // Phase-shift registers first step
    always @ (posedge clk90)
    begin : Reg_Phase
        if (enable == 1) begin
            Q_e <= Q_a;
            Q_f <= Q_b;
            Q_g <= Q_c;
        end
    end

    // Output registers
    always @ (posedge clk0)
```

```
    begin : Reg_H
        if (enable == 1) begin
            DataOut[0] <= Q_e;
        end
    end

        always @ (posedge clk0)
    begin : Reg_I
        if (enable == 1) begin
            DataOut[1] <= Q_f;
        end
    end

        always @ (posedge clk0)
    begin : Reg_J
        if (enable == 1) begin
            DataOut[2] <= Q_g;
        end
    end

        always @ (posedge clk0)
    begin : Reg_K
        if (enable == 1) begin
            DataOut[3] <= Q_d;
        end
    end
endmodule
```

B.5 Deserializer Control Module

```
module Deserializer_Control(
        input clk0,
        output reg DesRxValid,  // Valid to Receiver
        input RxDesReady,   // Ready from Receiver
        input SerDesValid,  // Valid from Serializer
        output reg DesSerReady, // Ready to
        ↪  Serializer
        output reg enable,
        output reg EnablePhase
    );

    reg ready_i, valid_01;

    always @ (posedge clk0)
    begin : Control
        ready_i <= RxDesReady;
        enable <= SerDesValid & ready_i;
        valid_01 <= SerDesValid;
        DesRxValid <= valid_01;
    end

    always_comb
    begin
        EnablePhase <= SerDesValid & ready_i;
        DesSerReady <= RxDesReady;
    end

endmodule
```

B.6 Serial Buffer Module

```
module Serial_buffer(
        input serial_data,
        output serial_data1
    );

    wire serial_data_i1, serial_data_i2, serial_data_i3;

        assign serial_data_i1 = ~serial_data;
        assign serial_data1 = ~serial_data_i1;
endmodule
```