Master's thesis

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

Morten Pedersen

On-Chip Ultra-Wide Global Interconnect Buses using Wave-Pipeline SERDES

Master's thesis in Electronic Systems Design
Supervisor: Prof. Kursun, Volkan
Co-supervisor: M.Sc. Lund, Morten W.
June 2022

NTNU
Kunnskap for en bedre verden

Morten Pedersen

On-Chip Ultra-Wide Global Interconnect Buses using Wave-Pipeline SERDES

Master's thesis in Electronic Systems Design
Supervisor: Prof. Kursun, Volkan
Co-supervisor: M.Sc. Lund, Morten W.
June 2022

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

NTNU
Norwegian University of
Science and Technology

Abstract

There has been a trend of decreasing scaling of process technology for a long time. As technology size decreases, the complexity increases, and the size of chips become bigger. The clock frequency has not increased at the same rate. Instead, the focus has been on multi-core systems and parallelism. These two factors combined have led to a bottleneck in interconnects. The solution to this bottleneck has for a long time been to increase the width of parallel buses. This solution has led to severe routing congestion for chips of extreme parallelism, such as GPUs. Routing congestion leads to higher power consumption and increased area usage.

This thesis explores the possibility of solving this routing congestion problem by reducing the lines on the bus. This is done by researching the wave-pipelining scheme and serializer-deserializer (SerDes) solutions. It is expected that a SerDes solution, with the extra logic needed to implement such a solution, will increase power consumption. The question is by how much, and if it is an acceptable amount for low-power and battery-powered chips compared to the area saved by utilizing such a solution.

The thesis presents a solution capable of a five-to-one reduction of data lines on the bus while keeping the same throughput as a parallel solution. A simple power analysis was conducted on the proposed solution and a parallel solution, which did not include power consumption from the lines on the bus. This power analysis showed an increase of 3.95 times in power consumption for the proposed solution compared to a parallel solution. However, it is expected that a reduction in the number of lines will have a positive impact on power consumption for the proposed solution. The proposed solution shows a reduction of lines on the bus by approximately 50%. Additionally, there is a 17.94% reduction in routing length.

Abstrakt

Det har lenge vært en trend med minkende størrelse på prosess teknologi. Men når teknologien blir mindre, øker kompleksiteten og størrelsen på chipene. Klokkefrekvensen har ikke økt med samme tempo. Istedenfor har fokuset vært på multi-kjerne systemer, og parallellisering. Disse to faktorene kombinert har ført til en bottleneck i interconnectene. Denne bottlenecken har lenge blitt løst ved å øke bredden på de parallelle bussene. For chiper med brede parallelle busser, som GPUer, har dette ført til at ledningsstiene på chipen kommer i konflikt med hverandre. Videre fører dette til høyere strømforbruk, og areal brukt på chipen.

Denne masteroppgaven prøver å løse disse problemene ved å redusere antallet ledningsstier på chipen. Dette kan gjøres ved bruk av wave-pipeline prinsipper, og serialisering-deserialiseringskretser (SerDes). På grunn av økt størrelse på kretsene ved bruk av SerDes er det forventet at strømforbruket vil gå noe opp. Spørsmålet er hvor mye, og er det akseptabelt for low-power og batteridrevne chiper.

En løsning som kan redusere antall ledninger med fem til en blir presentert. I tillegg holder løsningen samme ytelse som en parallell løsning. En enkel strømanalyse, som ikke tok ledningene på bussen med i beregningene, ble gjennomført. Denne strømanalysen viste en økning på 3.95 ganger i strømforbruk. Det er forventet at en reduksjon av ledninger vil gi en reduksjon i strømforbruk for løsningen, sammenlignet med en parallell løsning. I tillegg har løsningen en reduksjon i antall ledninger på omtrent 50%, og en reduksjon i rutelengde på 17.94%.

Preface

This thesis marks the end of a two-year long, highly educational, journey through my master's degree program in electronic systems design at NTNU. The thesis is a continuation of a project report for the specialization course TFE4590 - Specialization Project. The problem statement was proposed by ARM Norway, and the thesis started in January 2022. The thesis has been conducted in collaboration with ARM Norway.

While a master thesis is time-consuming, it has been highly educational and has helped me connect my knowledge, as well as expanding my knowledge, on various subjects. The master thesis has also been fun and challenging. Initially, a lot of time and effort was spent doing research and implementing the proposed solution. Additionally, a bigger than first anticipated part of the thesis was spent synthesizing the solution. This was mainly due to the optimization part of the synthesis tool and will be further elaborated on in the thesis. All figures, tables, and plots presented are original works by me, unless explicitly stated.

I would like to thank Per Olav Flø of ARM Norway for his time, dedication, and knowledge in routing of the design. I would also like to thank my advisor, Morten Werner Lund of ARM Norway, for the time and effort he has provided while guiding me through this master thesis. It would not have been possible to finish this thesis within the timeframe without the insight and knowledge provided by him.

Contents

Abstract iii

Abstrakt v

Preface vii

List of Tables xiii

List of Figures xv

List of Abbreviations xvii

1 Introduction 1
- 1.1 Overview . . . 1
- 1.2 Description . . . 1
- 1.3 Motivation . . . 2
- 1.4 Reliability Questions . . . 2
- 1.5 Requirements and Specifications . . . 2
- 1.6 Outline . . . 3

2 Background 5
- 2.1 Serializer-Deserializer . . . 5
- 2.2 Pipelining and Wave-Pipelining . . . 6
  - 2.2.1 Pipelining . . . 6
  - 2.2.2 Wave-pipelining . . . 7
- 2.3 Standard Cell Library . . . 10
- 2.4 Mesochronous Systems and Signaling . . . 12
  - 2.4.1 Asynchronous Signaling . . . 12
  - 2.4.2 Receiver Clock Generation . . . 13
  - 2.4.3 Source-synchronous Signaling . . . 13
- 2.5 Delay Elements . . . 14
  - 2.5.1 Digitally Controlled Delay Lines (DCDL) . . . 14
  - 2.5.2 Delay Uncertainty in DCDLs . . . 17
- 2.6 Wave-pipeline SerDes . . . 18
  - 2.6.1 XOR Based Solution . . . 18
  - 2.6.2 Multiplexer Based Solution . . . 18
- 2.7 Timing for a Wave-pipeline Serializer-Deserializer . . . 20
- 2.8 Multiple Buffering . . . 21
- 2.9 Timing Calibration . . . 22
- 2.10 Design for Testability . . . 24

3 Implementation **25**
3.1 Proposed Solution . . . 25
3.1.1 The Connection Between the Serializer and Deserializer . . . 26
3.2 Data Path . . . 26
3.2.1 Serializer . . . 27
3.2.2 Deserializer . . . 27
3.2.3 Digitally Controlled Delay Lines (DCDL) . . . 30
3.3 Control Signals . . . 30
3.3.1 Control Signals for the Serializer . . . 30
3.3.2 Control Signals for the Deserializer . . . 32
3.3.3 Handshake . . . 33
3.4 Calibration . . . 33
3.5 Timing for the Proposed Solution . . . 35
3.6 Design for Testability in the Proposed Solution . . . 37

4 Results and Discussion **39**
4.1 Methodology . . . 39
4.2 Important Design Choices . . . 42
4.2.1 Choice of Mesochronous Clock Scheme . . . 42
4.2.2 Choice of DCDL . . . 43
4.2.3 Choice of Buffer Scheme . . . 44
4.2.4 Other Design Choices . . . 44
4.3 Simulation Results . . . 44
4.4 Results . . . 46
4.4.1 Performance . . . 46
4.4.2 Power Consumption . . . 46
4.4.3 Routing . . . 48
4.5 Further Discussion . . . 49

5 Conclusion and Future Work **53**
5.1 Conclusion . . . 53
5.2 Future Work . . . 54

Bibliography **55**

Appendices

A Feedback from ARM **59**
A.1 Background . . . 59
A.2 Evaluation . . . 59
A.3 Conclusion . . . 59

B Source Code for the Serializer **61**
B.1 Serializer Wavepipe . . . 61
B.2 Serializer Multi Buf . . . 64
B.3 Serializer Data . . . 66
B.4 Serializer Control . . . 68

C Source Code for the Deserializer **73**
C.1 Deserializer Wavepipe . . . 73

C.2 Deserializer Multi Buf 76
C.3 Deserializer Data 78
C.4 Deserializer Control 81
C.5 SR-Latch 83

D Source Code for the Digitally Controlled Delay Lines 85
D.1 DCDL Module 85
D.2 Custom NAND Module 87

List of Tables

3.1 The truth table for the SR AND-OR latch 29

4.1 Power consumption for the proposed solution 47
4.2 Power consumption for a parallel solution 47

List of Figures

2.1 An overview of the modules and bitflow for a Serializer-Deserializer 6
2.2 A simple pipelining scheme 6
2.3 A simple wave-pipelining scheme 8
2.4 A simple layout of standard cells 11
2.5 Start and stop bit in the UART protocol 13
2.6 The input/output and the timing of a delay element 14
2.7 Inverter based Digitally controlled delay line 15
2.8 NAND-based Digitally controlled delay line 16
2.9 Glitchless NAND-based Digitally controlled delay line 16
2.10 Process corner model 17
2.11 The serializer of the XOR based Wave-pipeline SerDes 18
2.12 The Wave-Front Train solution. 19
2.13 The timing for the pilot and data bits in the WAFT solution 20
2.14 A double buffer solution for the WAFT SerDes 21
2.15 Example of autobaud when the transmitter has a lower baud rate 22
2.16 Example of autobaud when the transmitter has a higher baud rate 23

3.1 Overview of the proposed solution 25
3.2 A simplified view of the serializer- and deserializer units 26
3.3 The data path for the serializer in the proposed solution 27
3.4 The data path for the deserializer in the proposed solution 28
3.5 The SR AND-OR latch and the circuitry surrounding it 29
3.6 The Finite State Machine for the control signals of the serializer 31
3.7 How the deserializer buffer input and output are selected 33
3.8 A visualization of a calibration scheme 34
3.9 The delay path for bit 0 36

4.1 Alternative ways to connect the modules in the proposed solution 39
4.2 The modules of the proposed solution 40
4.3 Synthesis console message 41
4.4 Design objects, as seen from the synthesis tool 41
4.5 Simulation of a transmission for the proposed solution. 45
4.6 Simulation of a transmission were RxReady is set low 46
4.7 Power consumption for a parallel solution and the proposed solution 48
4.8 Connections to a shader core. 49

List of Abbreviations

IC Integrated Circuits

SoC System on a Chip

CPU Central Processing Unit

GPU Graphics Processing Units

SerDes Serializer-Deserializer

PISO Parallel In Serial Out

SIPO Serial In Parallel Out

ASIC Application-specific Integrated Circuit

HDL Hardware Description Language

AOI AND-OR INVERT

UART Universal Asynchronous Receiver-Transmitter

DCDL Digitally Controlled Delay Lines

WAFT Wave-Front Train

PVT process-, voltage-, and temperature

autobaud Automatic baud rate detection

TxValid Transmitter Valid

TxReady Transmitter Ready

RxValid Receiver Valid

RxReady Receiver Ready

FSM Finite State Machine

tcl Tool Command Language

DFT Design for Testability

Chapter 1

Introduction

This chapter starts with an overview, followed by a description and the motivation for the problem. The design targets and requirements are then presented. Lastly, an overview of the thesis can be found.

1.1 Overview

With the rapidly improving process technology scaling for System on a Chip (SoC) Integrated Circuits (IC) designs, an increasing need for higher logic- and memory density on the chip has emerged [1]. Additionally, SoC IC designs are becoming increasingly larger, with a higher number of modules per chip [2]. While this has been happening, the maximum clock frequency of the system clock has remained steady, and the focus has instead been on parallelism. An example of this phenomenon is the Central Processing Unit (CPU) module on the SoC [3]. For a long time, the focus for increasing the performance of the CPU was on higher maximum clock frequency. Then the focus shifted to multiple cores for boosting the performance [4]. Because of this shift in focus, the maximum clock frequency stagnated. This shift also meant that the CPU designs could be smaller and simpler, while instructions were executed in parallel, with a lower average clock frequency, which gave a lower power consumption for the chips [5].

While this shift in focus had many positive effects on the chips, there are some notable negative effects. With the high-speed parallel transmission, there is a potential timing problem between the data lines in the parallel transmission that can cause errors. This timing problem is caused by skew between the data lines [6]. Additionally, while the transistors have scaled down in size, the pad area has not scaled down at the same rate. The reason for this is due to thermal and mechanical bridging issues. This can cause the area to be wasted by the pads, as they take up more space relative to the transistors [7]. Secondly, with a high number of parallel units, the demand for interconnect bandwidth on the chips is higher for both computational and storage units, with the issue being amplified in certain highly parallel chips such as the Graphics Processing Units (GPU). This has led to a bottleneck in interconnects, which for a long time has been solved by increasing parallelism [8]. With an increasing demand for higher bandwidth, this increasing parallelism leads to issues of its own.

1.2 Description

Increasing the bandwidth of the buses by increasing the width of the buses has for a long time been an adequate solution. With the parallel width of the buses becoming extreme,

it is apparent that this is not a lasting solution [9]. With the parallelism of interconnects becoming extreme, the routing congestion also increases. This routing congestion leads to higher power consumption, lower data rates, and larger area usage due to the high amounts of parallel wires with repeaters. In this thesis, routing congestion is, as in the literature, defined as a scenario where the routing resources in parts of the design exceed the routing supply [10].

1.3 Motivation

With a parallel to how the CPU design shifted from single-core to multi-core, the focus of interconnects must now shift to serial solutions to rectify the routing congestion issues [2]. This thesis is a continuation of a report that utilized a time-multiplexing scheme to do this. In practice this means that the solution works at a higher clock rate than the system clock. While this solution can be used to solve the routing congestion issues, there are issues related to the higher power consumption of the multiple clock trees that other solution does not have. While the main focus is to reduce routing congestion, low power consumption is desired. The reason for this is that the solution should be viable for SoCs in low-power or battery-powered systems. In this thesis, wave-pipelining is explored as a means to reduce the number of wires in the interconnects. Additionally, a proposed solution that utilizes wave-pipelining with a Serializer-Deserializer (SerDes) scheme is described.

1.4 Reliability Questions

While there are papers describing similar approaches to what is done in this thesis, there are none, to the author's knowledge, that try to prove the reliability of such designs in regard to a larger SoC design. Certainly not as an IP solution for widespread use across technology nodes. This is the main contribution of this thesis. Though the solutions described in this thesis might have the potential of becoming the industry standard for on-chip interconnects, there are timing uncertainties that need to be addressed. This leads to the question; can a wave-pipeline SerDes be implemented so that it can be reliably used across multiple technology nodes? Additionally, there are questions to be asked in regard to how technology-dependent the proposed solution in this thesis is. Can the proposed solution be ported across technology nodes? Or is such a solution too reliant on the characteristics of the standard cell library used for the implementation to do so?

1.5 Requirements and Specifications

One of the main goals of this thesis is to determine the cost of area and power consumption compared to the reduction in wires and repeaters on the bus. Any solution that tries to reduce the wires on the bus will have a higher power consumption than a parallel counterpart, with the wires and buffers not considered. The reason for this higher power consumption is that it requires additional logic to implement. It is important to note that the different approaches to how a solution is implemented can have a major impact on just how much higher the power consumption is. For the proposed solution in this thesis, there are certain requirements and specifications:

– The solution should not have reduced performance compared to a fully parallel counterpart.

- The solution must provide at least four-to-one data per line performance, per system clock.
- The solution must be scalable, with a minimum width of 64 bits and at least up to a width of 2048 bits.
- The implementation must be done using the Hardware Description Language (HDL), SystemVerilog.
- The implementation must only use standard cell library cells, no custom logic, or analog blocks.
- The solution should only use a system clock as the reference clock and be mesochronous in nature.
- The solution must be scan test compatible.

1.6 Outline

The report is structured as follows: In Chapter 2, the required background knowledge is introduced, in addition to two example solutions that could be used to reduce the routing congestion. Chapter 3 presents the proposed solution, including its control signals, starting with an overview of the proposed solution, followed by an in-depth presentation of the modules in the system. In Chapter 4 the results for the proposed solution are presented and discussed, focusing on performance, power, and area. Additionally, this chapter includes the methodology for the proposed solution and discussions regarding important design choices made for the proposed solution. Lastly, the conclusion and discussions surrounding future work, can be found in Chapter 5.

Chapter 2

Background

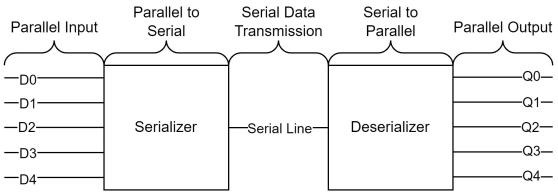
In order to understand how the proposed solution works, some background information is needed. In this chapter, this background information is presented. This includes principles, important designs, and examples.

Firstly, section 2.1 is an introduction to the serializer-deserializer scheme. Section 2.2 introduces the concept of wave-pipelining. Both of these sections help with the general knowledge needed for the proposed solution. Section 2.3 introduces and explains general information about standard cell libraries. This is done because an implementation must only utilize standard cell library cells, as mentioned in Chapter 1.5. Additionally, information about standard cell libraries is required for a discussion point, found in Chapter 4.5. In section 2.4, mesochronous systems are defined, and mesochronous signaling is presented. This includes three mesochronous signaling schemes. Delay elements and Digitally Controlled Delay Lines are presented in section 2.5. This is a vital part of wave-pipelining. Section 2.6 is about serializer-deserializer schemes that utilize wave-pipelining. This is followed by two examples of how such a scheme could be implemented, which aim to help with understanding how wave-pipeline SerDes works. In section 2.7, the timing for a wave-pipeline serializer-deserializer is presented. Section 2.8 introduces buffering schemes. In section 2.9, the concept of timing calibration is presented, in addition to an example. Lastly, in section 2.10, design for testability and scan chains are introduced.

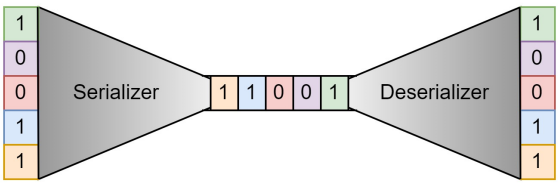
2.1 Serializer-Deserializer

A SerDes is a composition of blocks or modules used in high-speed systems at speeds up to the gigabit range. A SerDes can be used in systems with limited inputs and outputs, or with routing congestion caused by extreme parallelism to reduce the number of lines on the bus [11]. A SerDes typically consists of two blocks. Firstly, a Parallel In Serial Out (PISO) block, also called serializer, is connected to the parallel data on the input, and outputs the data in serial. The second block is the Serial In Parallel Out (SIPO) block, also called deserializer, which inputs the serial data from the PISO, and outputs parallel data. Figure 2.1a depicts the typical blocks of a SerDes and how they are connected, while Figure 2.1b shows the bit flow of the SerDes. Note that the direction of the transmission is important for the output to match the input. The PISO acts as a multiplexer circuit that converts the parallel data into serial data. The SIPO acts the opposite. For a high-speed SerDes the speed of transmission is typically increased in the PISO block, as more than one bit is transmitted on the serial line, compared to only one bit in a parallel line for each clock cycle [11]. On the other side of the transmission line, the SIPO converts the data from the serial output of the transmission line and back to parallel [7]. There are many possibilities of how these SerDes blocks can be made.

Maybe the simplest of these SerDes blocks is the shift register-based solution, which shifts the bits out onto a serial line. Many of the existing SerDes solutions share the same problem of requiring the system clock to serialize the data by which the performance of the SerDes is slow as a result. An example of this was implemented by the author in an earlier work [12]. This work implements a time-multiplexed solution utilizing multiple clocks of different phases to speed up the serialization. While the time-multiplexed solution gave results with a performance matching a parallel solution, it did so with higher power consumption. This thesis will focus on SerDes implemented using wave-pipelining, which has the potential of lowering the power consumption when compared to the time-multiplexed solution at the cost of higher complexity. Wave-pipelining will be introduced in Chapter 2.2.2.



(a) Typical modules of a SerDes, and how they are connected



(b) Bit flow of a typical SerDes

Figure 2.1: An overview of the modules and bitflow for a Serializer-Deserializer

2.2 Pipelining and Wave-Pipelining

This section serves to introduce pipelining and wave-pipelining, and the differences between them.

2.2.1 Pipelining

Pipelining, hereafter called traditional pipelining, is a way of processing data, typically used in processors [13]. Traditional pipelining resembles the assembly line of a car factory, where a new part of the car gets added for each step. This means that the assembly of the next car can start as soon as the current car moves along down the assembly line, which will make better use of the available resources. The time it takes to assemble a single car might be the same, or even take longer, called latency in computing. But if there is a car under assembly on each step of the assembly line, the output of cars that are completely assembled is increased after the first car, called throughput in computing.

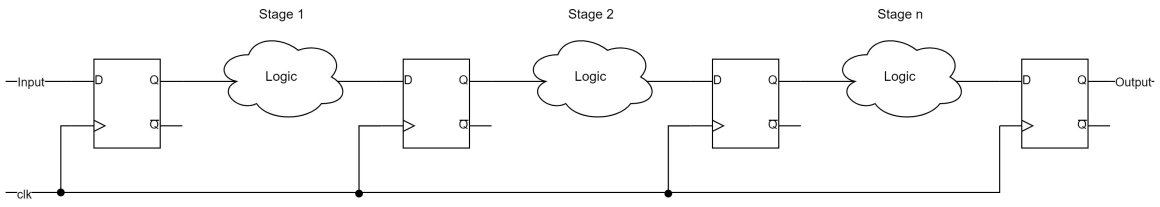


Figure 2.2: A simple pipelining scheme

Figure 2.2 shows how the pipeline scheme can be implemented in hardware. Each stage, which contains unique combinatorial logic, is separated by either register or latches,

functioning as a buffer. The latency is defined by the fact that any single input needs to go through all the stages to reach the output. On the other hand, a new input can be processed one clock cycle after the first one, and so on, making it so that new output is available for each clock cycle after the first one, giving a high throughput. The only requirement for a simple scheme like this to work is that the combinatorial logic between the registers does not induce more delay than what is available in a single clock cycle and that the setup and hold time for the registers are not violated [14].

Ideally, a traditional pipeline is synchronized so that each stage takes the same amount of time. This synchronization would make the signals flow through the system at a constant speed, with a constant delay between them. As the stages generally do not have the same propagation delay, buffers between the stages are needed. Traditional pipelining is constrained by the frequency of the clock as the signals of the different stages need to be sampled by the registers that separate the stages. For a pipeline that is isolated with a different clock domain than the rest of the system, the clock frequency might be constrained by the stage with the longest propagation path, also called the critical path. Because of this, every other stage, except the stage containing the critical path, must stay in a wait state while the stage containing the critical path finishes processing, as the frequency of the pipeline is dictated by the critical path.

The logic in a traditional pipeline circuit can not be shared between the stages. This means that traditional pipelines typically require more resources when compared to systems that can reuse parts of the circuit, making the area overhead higher. There is a second factor that causes the area overhead to be increased. This factor is the registers needed for the synchronization between the steps, and it is typically more of them in high-performance pipelines. Additionally, the power consumption is typically also higher. This is mainly due to the synchronization registers or latches in addition to the increased clock buffer area required to drive the clock used for the synchronizers. Another problem with the traditional pipeline is the latency that can not easily be reduced without modifying the functionality of the pipeline. The latency is defined by both the clock frequency of the system and the number of stages in the pipeline. With a high number of stages in the pipeline, the latency is also increased. In theory, each pipeline step can be reduced into small sub-blocks, which enables the clock frequency to be increased. However, this does not work in practice past a certain point, as the setup, hold, and propagation time for the registers can not be reduced [15]. These constraints cause a diminishing return as the constraints become the dominating part compared to the critical path of the pipeline. Additionally, the needed extra registers in this scenario require the clock network to be expanded, which increases the power consumption and area requirements further. Additionally, the increased clock skew from routing can become an issue with the higher number of clocked registers.

A number of the traditional pipeline problems are related to the synchronization steps between the stages. They increase the area required, leads to a higher power consumption due to the clock tree having to drive many registers, and have a higher latency due to the registers. But what if these synchronizers could be omitted?

2.2.2 Wave-pipelining

Wave-pipelining is based on the same principles as the traditional pipeline, with a key difference: The synchronization between the steps does not contain registers. Instead, logic gates are used as storage elements, where the propagation delay is the most important variable. This can be seen in Figure 2.3, where the delay elements are these storage elements. Delay elements will be introduced in Section 2.5. As the register are no longer a part of

the pipeline, the setup, hold, and propagation delay of the registers are no longer a factor when the pipeline stages are reduced to smaller sub-blocks. Additionally, as the registers are removed, so is the clock network used for clocking the registers, and with that, the power consumption is reduced.

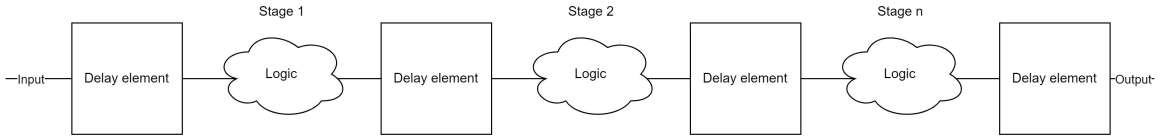


Figure 2.3: A simple wave-pipelining scheme

As the delay properties of the registers are not a concern for wave-pipelining, its creator, L. Cotten, called it maximum rate pipelining [16]. Wave-pipelining is based on an observation that the rate at which the signal can propagate through a circuit does not depend on the longest delay path, but the difference between the longest and the shortest path delays [17]. Based on this observation, a circuit can be made that makes use of propagation delay so that multiple signals that belong to different cycles of the clock can propagate through the circuit simultaneously.

Wave-pipelining requires much consideration for timing. As there are multiple signals propagating through the pipeline, which can be seen as waves, it is important that the different signals never interfere with each other while propagating through the pipeline. For this to be possible, a constructive skew is needed. A constructive skew is a skew intentionally created between two signals. Additionally, the elements that the wave-pipeline consists of are also important. Different elements have different propagation delays, which must be accounted for when constructing a wave-pipeline. Knowledge of technology and process to be used, is important for the propagation delay. The reason for this is that the propagation delay for two otherwise identical cells in two different process technologies can be widely different.

While no registers are needed between the stages in the wave-pipeline, there is still a need for registers at the input and the output. The constructive clock-skew is the skew between the input and the output registers and is depicted using the symbol Δ. D_{min} and D_{max} respectively depict the minimum and maximum propagation delay in the combinatorial logic between the input and output registers. In addition, there are the uncontrollable factors:

- T_s - setup time
- T_h - hold time
- D_R - propagation delay through a register
- ΔCK - the worst-case uncontrolled clock skew at a register.

For the correct operation of a wave-pipeline system, the timing for the clocking of the output register is critical. The register must be sampled after the data has propagated through and the last data has arrived but before the first data bit of the next clock cycle arrives. This will be further elaborated on in Chapter 2.7. Equation 2.1 is the equation for T_L, which is the time the data must be sampled by the output register relative to the input register. In this equation, N is the number of clock cycles after the data is sampled by the input register, and T_{CK} is the clock period. This equation can be used to calculate the constructive clock skew based on the total propagation delay of the combinatorial logic in

the wave-pipeline. The equation is also important to make sure the data has arrived at the output register before any value is sampled. Even though N indicates that a wave-pipeline can handle data signals propagating for multiple clock cycles, in this thesis, it is assumed that N is kept at a value of one. This means that no data signals should propagate through the wave-pipeline for longer than one clock period. The reason for this restriction is the lowered complexity. Additionally, the longer the propagation path through the wave pipeline is, the higher the possibility for variations are. This is due to differences in propagation delay through a logic gate caused by differences such as if the data bit is zero or one. This will be further elaborated in Chapter 2.5.2.

$$T_L = N \cdot T_{CK} + \Delta \tag{2.1}$$

It is also important that the data is sampled before the next wave of data arrives so that each wave of data in the wave-pipeline does not interfere with each other. This is shown with the lower bound of T_L in Equation 2.2. Here, D_R is the propagation delay of the input register, as it is sampled at the start of the wave-pipeline. The lower bound is the lowest time a signal can propagate through the wave-pipeline without interfering with the next wave. Note that the only variable in the equation is the total propagation through the wave-pipeline, which means it is important to make sure the total propagation is lower than the maximum propagation to avoid interference between the different waves at the output of the wave-pipeline.

$$T_L > D_{max} + D_R + T_S + \Delta CK \tag{2.2}$$

At the other end of the spectrum is the upper bound of T_L. The upper bound is the lowest amount of time a signal can propagate through the wave-pipeline without interfering with the last signal. This scenario is shown in Equation 2.3. Here, the additional hold time T_h is added for the output register. The uncontrollable clock-skew, ΔCK, is added to both Equation 2.2 and Equation 2.3 because it impacts when the clock edge arrives at the output register, and with that, when the output register samples the data.

$$T_L < T_{ck} + D_r + D_{min} - (\Delta CK - T_h) \tag{2.3}$$

When combining the formula for clocking the latest data and the earliest data, we get Cotton's maximum rate pipelining condition, which can be seen in Equation 2.4. This equation determines the lowest possible clock period for the wave-pipeline. $(D_{max} - D_{min})$ shows how the minimum clock period is determined by the difference in path delays as stated earlier in the section, and $(T_s + T_h + 2\Delta CK)$ is a clocking overhead from the two registers at the input and output.

$$T_{ck} > (D_{max} - D_{min}) + T_s + T_h + 2\Delta CK \tag{2.4}$$

The equation above can be used to make sure that the waves do not interfere with each other at the output of the wave-pipeline. But waves must not collide in transit either, meaning that the first wave must never arrive at a node before the last wave has propagated through. Equation 2.5 shows the constrain, which can be used to make sure the different waves in the wave-pipeline hold a distance in delay all the way through the wave-pipeline. In this equation, x is defined as the output of a logic gate in the wave-pipeline, and T_{sx} is the minimum time

that the node must be stable so that the signal can propagate correctly through the gate. Note that T_{sx} is equivalent to $(T_s + T_h)$, and $(d_{max}(x) - d_{min}(x))$ is equivalent to $(D_{max} - D_{min})$ in the maximum rate pipelining condition shown in Equation 2.4.

$$T_{CK} => d_{max}(x) - d_{min}(x) + T_{sx} + \Delta CK \tag{2.5}$$

The previous conditions and equations can be used to find a minimum clock period for any number M of waves. With two additional parameters, T_{min} and T_{max}, which is the minimum and maximum propagation delay for all the logic gates. With Equation 2.6 and Equation 2.7, it is possible to get a two-sided constraint of the clock period which can be seen in Equation 2.8. It should be noted that if the M variable is constrained to a value of one, the lower bound for T_{ck} disappears, and there is only an upper bound left. This would make the wave-pipeline equal to a traditional pipeline.

$$T_{min} = D_r + D_{min} - \Delta CK - T_h - \Delta \tag{2.6}$$

$$T_{max} = D_r + D_{max} + T_s + \Delta CK - \Delta \tag{2.7}$$

$$\frac{T_{max}}{M} < T_{ck} < \frac{T_{min}}{M-1} \tag{2.8}$$

2.3 Standard Cell Library

Standard Cell Libraries are used in semiconductor design of Application-specific Integrated Circuits (ASICs). They are used as libraries with, for the most part, digital components that can be used to implement a bigger design [18]. One of the advantages of using standard cell libraries is the abstraction level it provides. A designer does not have to think in terms of transistors, but rather logic representations of basic building blocks, such as NAND-gates or full adders. Another advantage of standard cell libraries is that it makes it possible to split the design into two parts: a logical functionality part and a physical implementation part. This makes it possible for two different designers to both focus on and specialize in each part. Some additional advantages of using standard cell libraries are that the design time is much lower, with a lower risk because the libraries used are extensively tested. There are also some disadvantages of standard cell libraries. Firstly, designs can not be customized at the transistor level. This means that the transistors of the design cannot be designed optimally for any specific use case. Secondly, the standard cell library approach might not be economical as all masks used to produce the design has to be produced aswell [19].

Standard cell libraries do not use fully custom layouts but are instead arranged in standard cells in neat rows [20]. Examples of these layouts are inverters, logic gates, and even half and full adders. Standard cells are made based on a set of rules:

- The height of any cell must be constant, and be of one or more predetermined heights.
- It must have the voltage source in a metal line on top and the ground in a metal line at the bottom.
- P-well needs to cover the top half of the standard cell.
- Inputs and outputs need to be provided in the metal layer.

All the rules are in place to make sure that it is easier to use the cells when physical implementation is done. The first rule makes sure that the cells line up in neat lines when arranged in rows. This rule also makes it easier to make complete metal lines at the top and bottom of the line of cells for the voltage source and ground, respectively, when considering rule number two [19]. Note that this does not mean that the height is only predetermined, as the height of a single row. The height can also be, for example, double height, where the cell takes up a height equal to two rows. Rule number three is not as set in stone as the other rules, where the P-well does not always have to cover the top half, but it does need to cover the same proportions in all the cells consistently. The last rule makes it easier to connect the output of one standard cell to the input of the next, simplifying the routing between the standard cells.

More complex logic cells than simple combinatorial logic cells such as NAND-gates exist. There are both more complex combinatorial logic cells, which consists of multiple logic gates, and sequential cells such as flip-flops and latches. Some examples of more complex combinatorial cells are the AND-OR INVERT (AOI) and the full adder mentioned above. Which cells the library consists of is dependent on the library itself, but normally all libraries contain the most important or most used cells. The sequential cells exist in configurations with different inputs and outputs, such as positive and negative edge triggering, resets, non-inverted and inverted outputs, and enable. Another important factor in standard cells is drive strength. Each cell has multiple versions with different output stages of various sizes. If a cell has a larger output stage, the cell can drive more logic and typically has a lower propagation delay, while a smaller drive strength requires less area and has lower leakage.

As the standard cell libraries are typically optimized for speed, specialized cells used for clocking exist that are optimized for minimal skew instead. This does not only include the clock buffers, which generally is the only logic that should be used on clock nets, but also integrated clock gates, used for clock gating. In addition to standard cells, some special cell types can be used. One of these cells is called a filler cell and is used to fill the space between two cells. These cells ensure continuity in the connection of the power supply and ground lines at the top and bottom of each cell row. Another type of cell is called level shifter cells. These cells are used to pass signals between voltage domains.

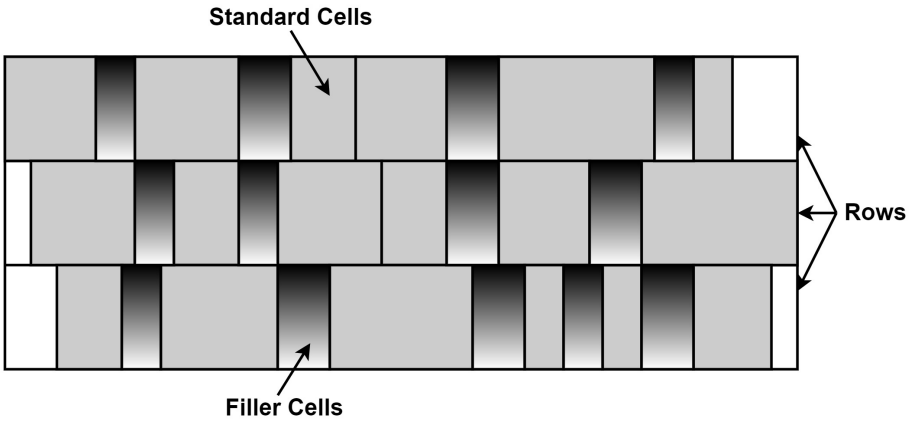


Figure 2.4: A simple layout of standard cells

The standard cell mini layouts are arranged in a larger layout, as can be seen in Figure 2.4. This layout makes it easier for the designer, as it enables them to utilize an HDL to specify the circuit, while synthesis tools translate this specification into the layout. Including

picking standard cells, place in rows, and connecting them to the design to provide the specified functionality. The wider the standard cell is, the more complex it is.

Each standard cell contains multiple files with information about different aspects of the cell:

- .V file containing a Verilog description of the cell, used for simulations.
- .gds file containing the layout of the cell.
- .lef file containing an abstract of the cell.
- .spi or .cdl file which contains a netlist for transistor-level simulations. This can be both with parasitics and without.
- .lib file which contains timing, area, and power characterization.

All the information that can be found in these files combined, defines the functionality of the cell. The timing characterization that can be found in the .lib file consists of propagation delay, rise time, and fall time. Note that the library contains separate files for the different process corners. Standard cells are composited into libraries that specify which standard cells the tools have available to make up the specified circuit. Standard cell libraries are typically vendor-specific, meaning they are related to whoever is fabricating the chip. The .lib file containing the timing, area, and power characterization, will be most useful for this thesis.

2.4 Mesochronous Systems and Signaling

In a mesochronous system, even though all the clocks have the same frequency, signal events happen with a constant but unknown phase relative to the system clock [21]. This can impose some difficulties when designing such a circuit if the only clocking scheme in the circuit is a global clock. If the circuit, such as in this thesis, consists of a transmitter and receiver side, and no effort was made to synchronize the two parts relative to each other, sampling on the receiver side would probably yield the wrong result. If this is not the case, it might not be stable over time as temperatures changes and the different components of the delay change. Because of this, it is important to find a stable solution. There are a few ways to solve it, and this section aims to introduce some of the best solutions.

2.4.1 Asynchronous Signaling

The first solution is a fully asynchronous handshake system consisting of request and acknowledge [22]. The request and acknowledge signals would accompany each transmission, making the system much more robust to timing uncertainties. A handshake occurs when both the request and the acknowledge signals are high simultaneously.

For an asynchronous signaling scheme to work, there would have to be two separate wires in addition to the data wires on the bus. If the goal of this thesis was to reduce the number of wires on the bus at all costs, this solution would not be feasible. However, while it is important to reduce the number of wires on the bus, a SerDes solution has to be reliable. The signaling scheme is an important part of this reliability, which will be elaborated on further later in the thesis. While the asynchronous signaling scheme is robust to timing uncertainties, which increases the reliability, the scheme has added delay and power consumption caused by the two wires and the handshaking between them. The added delay is caused by the need for the acknowledge signal to be rerouted back to the source and can potentially cause the latency

to increase. There is a different approach where only a single wire is added to the bus. The problem with this is that the wire would have to be bidirectional, adding complexity and problems, such as how to fine-tune the synchronization between the two wires when both the transmitter and receiver can not communicate on the wire at the same time. Note that it became apparent early in the research that this would not be a viable solution, and because of this no deeper research was conducted on the solution.

2.4.2 Receiver Clock Generation

Another solution is to generate the clock in the receiver, and by that, no additional information has to be transmitted other than the data. In one of these solutions, which is called Pausible-clocked systems, a locally generated clock is made typically using a ring oscillator [22]. The advantage of this is that it can be calibrated based on process variations so that the receiver can operate on a slightly different clock frequency than the transmitter. A big disadvantage is that the generation of the clock in the receiver adds substantial overhead in both area and power consumption. This solution also adds complexity to the system.

2.4.3 Source-synchronous Signaling

Source-synchronous signaling is used in many existing systems. This solution works by adding a timing signal which is transmitted from the serializer to the deserializer in either a separate wire or injected into the data wire. There are multiple ways a source-synchronous signaling scheme can be implemented. In this section, a few of these solutions will be explored.

Pilot Bit Signaling

With pilot bit signaling, the clocking signal is injected into the data stream. A famous example of a system utilizing pilot bit signaling is the Universal Asynchronous Receiver-Transmitter (UART) [23]. Figure 2.5 depicts how this is done. An important part of detecting the idle bit is that the line is high when idle, while the pilot bit is low. This means that detection of the pilot bit happens at the falling edge of the data line. This is then used to synchronize the clocks and signal the receiver that the data with a predetermined number of bits is coming.

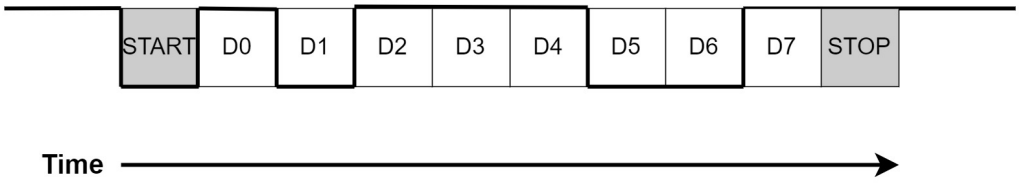


Figure 2.5: How the start and stop bits are injected in the UART protocol

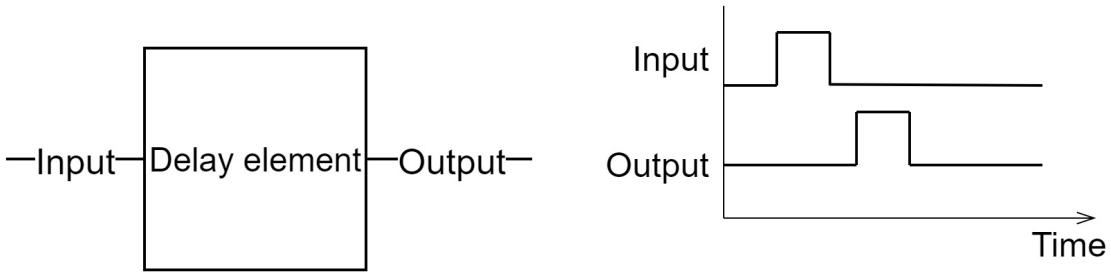
An advantage of the pilot bit signaling is that it enables high-performing data links. The solution is robust with respect to hard-to-control delays such as process variation and noise, as the pilot bit is sent on the same line as the data. This means that the delay for the data bits on the serial line is shared with the pilot bit. A disadvantage of this scheme is that the pilot bit takes up valuable space on the data line that could be used to send additional data instead. There must also be circuitry in the receiver that can trigger based on this signal, and this might be hard when the length of the data pulse is short at higher clock frequencies. This will be revisited in Chapter 4.

Separate Synchronization Line

This scheme is similar to the pilot bit signaling scheme. The biggest difference is that the clock signal is sent on a separate wire on the bus [24]. While it adds an extra wire, it also enables additional data bits to be sent in a single frame. Another positive effect of sending the clock signal on a separate wire is that it enables finer tuning for the timing of when the bit arrives in the receiver compared to the data bits. This effect makes it easier to design circuitry on the receiver end that reacts to the clock signal, which lowers the complexity of the design. There is also a point to consider; when the clock signal should be received in the deserializer compared to the data bits. If it is received at the same time as the first data bit, there would be a longer period to receive the clock bit and react to it, as the rest of the data bits arrive in the deserializer. For this to be possible, it is important that the delay for the separate synchronization line is equal to the delay of the serial line for the data bits. However, if it is received at the same time as the last data bit, no processing is necessary, and there would only need to be a reaction when the clock signal arrives. This sets a limit on possible methods that can be used to process the clock signal, as there is less time to process it.

2.5 Delay Elements

One of the most important modules in a wave-pipeline system is the delay elements. The delay elements are used to distance signals in time on the data line by delaying the signals using propagation delay [25]. Figure 2.6a shows the inputs and outputs of a delay element, and Figure 2.6b shows the timing for a delay element. The simplest form that delay elements can be implemented is by chaining an even number of inverters. How many inverters the delay element consists of decides how long the generated delay is. Delay elements do not have any functionality apart from adding the extra propagation delay, meaning that the logical functionality is the same on the input as the output, with the only difference being the added delay. In this section, different ways delay elements can be implemented are presented. Additionally, important delay properties that affect how the delay elements work are presented.



(a) The input/output of a delay element (b) The timing for a delay element

Figure 2.6: The input/output and the timing of a delay element

2.5.1 Digitally Controlled Delay Lines (DCDL)

Digitally Controlled Delay Lines (DCDL) is a type of delay circuit that utilize a select signal to determine the delay path, and with that, making the delay programmable [26]. What

makes the DCDL special is that the delay can be adjusted in run-time or even while actively used. Multiple DCDL designs exist [27]. This section will introduce some of these DCDL solutions.

Inverter Based Digitally Controlled Delay Lines

The simplest of these solutions is inverter-based and can be seen in Figure 2.7 [27]. This solution is made using pairs of inverters, chained, that output into multiplexers, where for each pair of inverters, the delay increases by $2 \cdot D_{inverter}$. The select signal for the multiplexers is used to determine how much delay is generated in the DCDL. This solution works best for situations where high resolution is desired, meaning the delta delay between the different steps should be small. The reason for this is that the inverters typically have a lower propagation delay when compared to other logic gates that are used in other DCDL designs. This difference also means that if a high delay is necessary, there would have to be many pairs of inverters to do so. The inverter-based DCDL also has a potential glitching problem. Glitching can occur when the select signal for the multiplexer's switches. This might be a problem if it is important to be able to switch the delay while the signal is high. Glitching is here defined as an unwanted pulse at the output of the DCDL. Note that this is not the only solution that will be presented with this potential problem. There are also some problems with this solution related to jittering, which will be further presented in section 2.5.2.

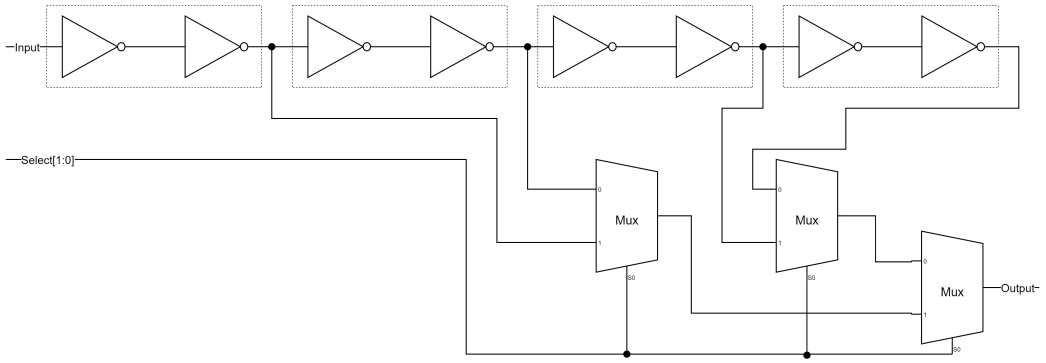


Figure 2.7: Inverter based DCDL. The figure is adapted from D. Preethi [27]

NAND-Based Digitally Controlled Delay Lines

The NAND-based DCDL can be seen in Figure 2.8 [27]. The initial step consists of two NAND-gates connected in serial and a load balancing NAND-gate. Each step after the first one consists of three NAND-gates in serial in addition to a NAND-gate used for load balancing. The last step called the end step in Figure 2.8, is set up to make sure that the signal that leads to the output is correct for all select signals. This includes the case where the select signals, *S0-S2*, is zero.

$$D_{total} = (2 \cdot D_{nand}) + ((n-1) \cdot 2 \cdot D_{nand}) \tag{2.9}$$

NAND-gates typically have a higher propagation delay than an inverter which gives lower resolution than inverters. This means that the DCDL design is more suited in situations where the delay needed is higher. The first step gives a propagation delay of $2 \cdot D_{nand}$, while each incremental step gives an added delay of $2 \cdot D_{nand}$, where D_{nand} is the propagation

delay for a NAND-gate. The total delay for each step can be seen in Equation 2.9, where n is the step number. An example is for step 2, where n would be two, and the total delay would be $4 \cdot D_{nand}$.

Select signals are used to decide the active delay path. However, these select signals can cause glitching. If the select signal is changed while the input is active, there is a risk that the output can be driven by multiple delay paths, which causes glitching. The next solution solves this problem.

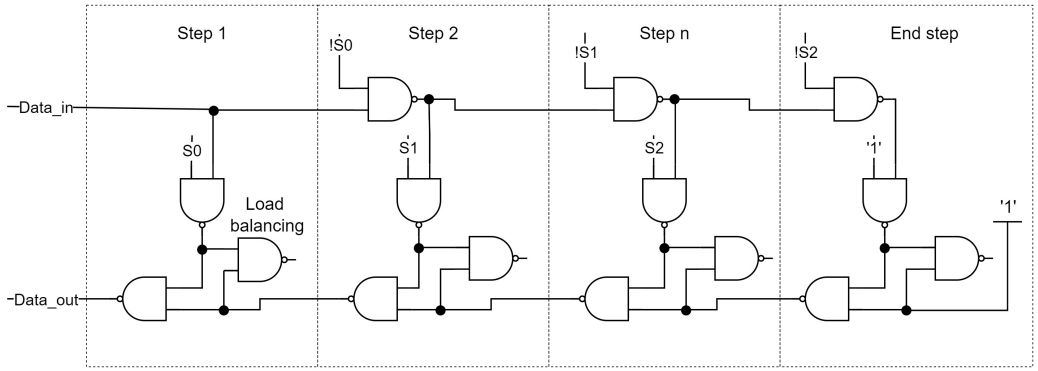


Figure 2.8: NAND-based glitchy DCDL. The figure is adapted from D. Preethi [27]

Glitchless NAND-Based Solution

The glitchless NAND-based solution was developed by De Caro [28] to remove the glitching problem. The glitching problem is removed by adding a NAND-gate with an extra control bit T_i, as can be seen in Figure 2.9. This also comes with the extra condition of $S_i = 0$ for $i < c$ and $S_i = 1$ for $i \geq c$, $T_i = 1$ for $i \neq c+1$ and $T_{c+1} = 0$, where c is the total control input value. These rules dictate that the S signal should be low for all stages before the active stage, high for the active stage, and the rest of the stages. Additionally, the T signal should be high for all the stages, except for the next step after the step currently in use. Control logic is needed to make a delay between the two sets of control signals to make sure glitching can not occur in any situation.

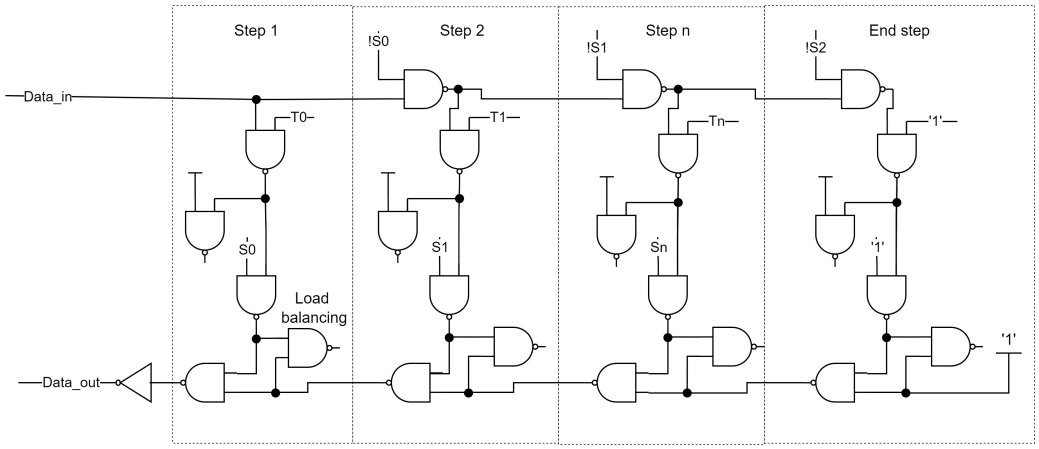


Figure 2.9: Glitchless NAND-based DCDL. The figure is adapted from De Caro [28].

$$D_{total} = (3 \cdot D_{nand}) + ((n-1) \cdot 2 \cdot D_{nand}) \tag{2.10}$$

Note that there is an odd number of NAND-gates for all the delay paths. The odd number of NAND-gates causes the output to be inverted compared to the input. This inverting issue is fixed by inserting an inverter at the output, as can be seen in Figure 2.9. Unlike the glitchy NAND-based DCDL, the first step of the glitchless solution gives a propagation delay of $3 \cdot D_{nand}$, with the total delay as can be seen in Equation 2.10. Like for the glitchy NAND-based DCDL, n is the step number. The increased propagation delay gives a higher minimum delay but the same delay resolution as the glitchy solution. Additionally, the glitchless solution adds an extra load balancing NAND-gate, increasing the NAND-gate count by two for each step compared to a glitchy solution.

2.5.2 Delay Uncertainty in DCDLs

All the DCDLs introduced above has uncertainties or uncontrollable delay variations. First off, it should be mentioned that there is a reason why all the DCDLs above contain inverting gates such as inverters and NAND-gates. The reason for this is that there is an inherent difference between the rise-time and fall-time for the different cell elements. This difference means that if the DCDLs contained non-inverting gates such as an AND-gate, there would be an accumulating delay difference between a high and a low bit, which would be worse the higher the number of steps in the DCDL is. Instead, with an inverting gate, the logic value for the bit would invert for each element in the DCDL averaging the rise-time and fall-time.

Another uncontrollable delay variation comes from the difference between the two most extreme process corners, slow-slow and fast-fast. These two corners are marked SS and FF in Figure 2.10. A difference of 48% in propagation delay in a NAND-gate between these two corners is not unrealistic. Note that this value is based on a standard cell library the author of this thesis had access to and is not representative of every standard cell library for all processes.

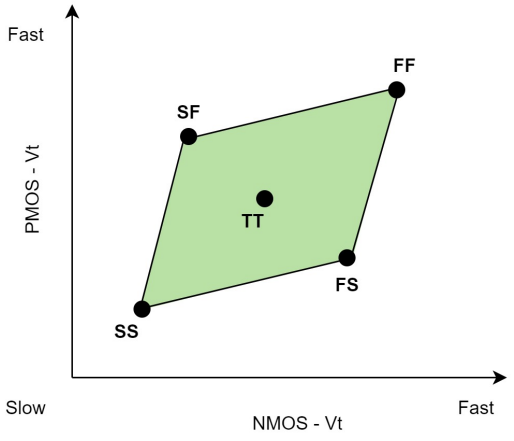


Figure 2.10: Process corner model

2.6 Wave-pipeline SerDes

The wave-pipeline SerDes combines the principles of wave-pipelining and the SerDes. The general idea is to release several bits on the serial line with a delay between them and, by that, transmit each bit as a wave on the serial bus [29]. This delay is controlled and generated using delay elements, such as DCDLs, as described in section 2.5.1. Wave-pipeline SerDes can be implemented in multiple ways. This section will explore two of these solutions and present some of the positive and negative aspects of these solutions.

2.6.1 XOR Based Solution

This solution is based on an earlier design by the author [12], but instead of multi-phase clocking, the design utilizes DCDLs. The serializer for the design can be seen in Figure 2.11. Note that the figure only shows the data path. At the start of a transmission, a pilot bit is transmitted on the serial line. After the first bit, which is the output of Reg A, has propagated through DCDL1, it is transmitted on the serial line. This same principle continues with the rest of the bits until the line is pulled down through the PullDown register. The propagation delay of the DCDLs decides how long each bit is transmitted, as all the registers are clocked by the same clock. This also means that the DCDL decides the width for each of the data bits. This is done by having a linear increase in the propagation delay through the different DCDLs, meaning DCDL2 should have about double the propagation delay compared to DCDL1, etc. Note that each bit has an extra XOR-gate that the signal must propagate through, which decreases the delay path needed in the DCDL element for that specific bit.

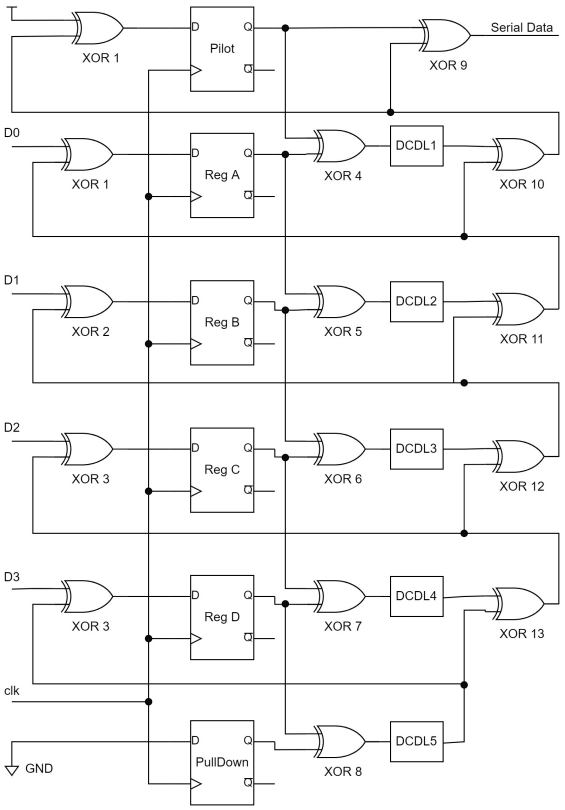


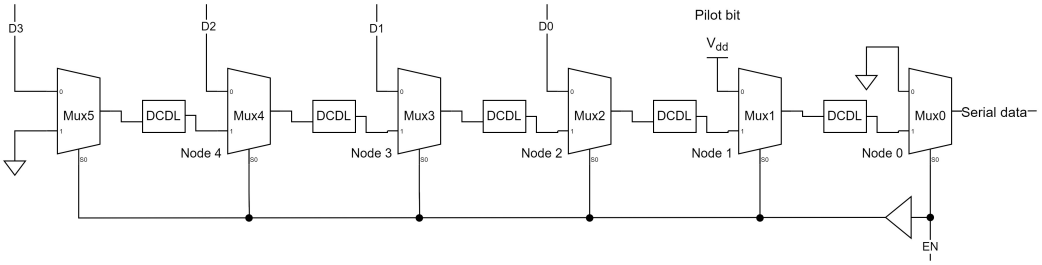
Figure 2.11: The serializer of the XOR based Wave-pipeline SerDes

The advantage of this solution is that it can be implemented by only using standard cell logic libraries, which is a requirement for the solution of this thesis. A disadvantage of this solution is that the XOR-gates have internal switching, which leads to higher power consumption. The design is also hard to modify. An example could be if an implementation of the XOR-based solution is made without a pilot bit, with another clock synchronization scheme. Note that this design as shown in Figure 2.11 has not been tested.

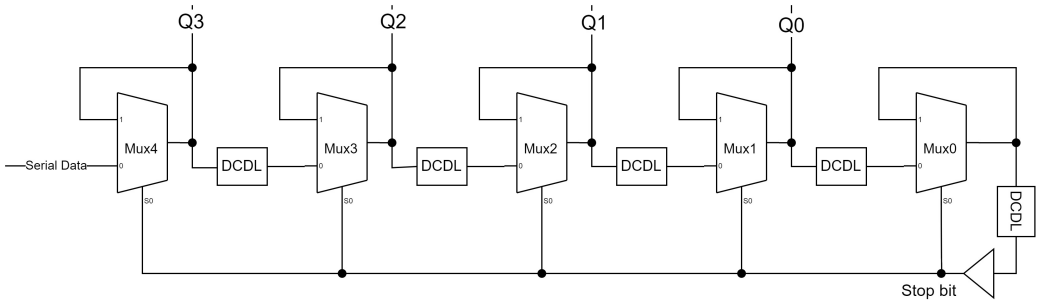
2.6.2 Multiplexer Based Solution

This solution, called Wave-Front Train (WAFT), as proposed by S. Lee et al., utilize multiplexers to implement a wave-pipeline SerDes [30]. WAFT is a four-to-one SerDes, capable

of 2 Gb/s data transfer rate, according to the inventors. The serializer of the solution can be seen in Figure 2.12a, and the deserializer can be seen in Figure 2.12b. The four parallel bits of data D3-D0 are connected to one of the inputs of the multiplexer. The multiplexer closest to the serial line in the serializer is used to set the serial line to ground when the serializer is disabled. The multiplexer that inputs D3 sets the serial line to ground at the end of the transfer. When the EN signal is low, the parallel data is loaded into the circuit through their respective multiplexer and further through the DCDLs before it stops at the input of the next multiplexer. An example of this is $D3$ propagating through *Mux5* and stopping at input one of *Mux4* at *Node 4*, which can be seen in Figure 2.12a. A pilot signal is sent at the head of the data packet. This pilot signal is used to stop the propagation of the data bits at the end of the path in the deserializer. When the EN signal is asserted, the data bits start to propagate through the circuit as waves with a distance of $D_{DCDL} + D_{mux}$ delay between them. The signal keeps propagating until it reaches the deserializer and the pilot signal reaches the end of the deserializer. When this happens, the multiplexer for the individual bits in the deserializer loops back to itself, making it a latch that holds the value at the output, marked with *Q3-Q0*. Note that the DCDL at the end of the deserializer should have a significantly lower delay path than the rest, as it is there to calibrate the pilot bit compared to the data bits.



(a) Serializer for the Wave-Front Train.



(b) Deserializer for the Wave-Front Train.

Figure 2.12: The Wave-Front Train solution. The figure is Adapted from S. Lee et al. [30].

There is a difference between rise-time and fall-time for multiplexers, like with DCDLs as mentioned in 2.5.2. To compensate for this delay difference, the DCDLs are implemented with an even number of inverters. This leads to the polarity of the propagating signal to invert at every DCDL, averaging the rise and fall times. This is a much-used technique in phase interpolation. An important aspect that the WAFT depends on is that the delay for the serializer and deserializer is identical. If they are not, there is a high risk that the difference leads to jittering at the deserializer, causing a degradation of the performance. WAFT is also sensitive to supply voltage variations. With big variations in the supply voltage, there is a

risk of severe performance degradation in the deserializer. The solution for this degradation is to adaptively control the supply voltage based on what is needed for the given bandwidth.

2.7 Timing for a Wave-pipeline Serializer-Deserializer

Transmission of data is completed in a single clock period, as defined in section 2.2.2. This means that each bit of the serial data must operate at a much higher frequency than the clock. This is affected by how many bits the transmission includes. More bits lead to shorter time per bit, while fewer bits give a higher time per bit. Figure 2.13 shows the timing for the four data bits and the pilot bit for the WAFT solution. At the rising edge of the clock, when the enable signal is asserted, the respective nodes already hold the value of the data bits. It is not before the signal from the last node has propagated that the value changes. Each bit can be followed through the serializer, from the first node following the respective data bits parallel input until it reaches the serial line. Note that the figure does not show the delay through the multiplexers, which is the reason that *Node 0* is the same as the serial line. In reality, there is propagation delay through the multiplexers and not just the DCDLs. This will be further discussed in Chapter 4.

In Figure 2.13, it can be seen that with any given clock frequency, the time for any additional data bits on the serial line must be taken from the preexisting bits. By taking time from the preexisting bits, the pulse width for each bit is shorter, given the assumption that the transmission should be finished in a single clock cycle. On the other hand, the maximum bandwidth for a wave-pipeline SerDes is dictated by the minimum pulse width that can be transmitted. While in theory, this is only dictated by the rise- and fall time, in reality, there are multiple constraints categorized under timing uncertainties.

The three important variations of process-, voltage-, and temperature (PVT) can have an impact on any electrical circuit [31]. This especially impacts a circuit that is timing sensitive such as a wave-pipeline SerDes. These variations are extensively tested for in any circuit, even down to corner cases. This testing is done to make sure that the circuit works as expected with high variations in these three variables.

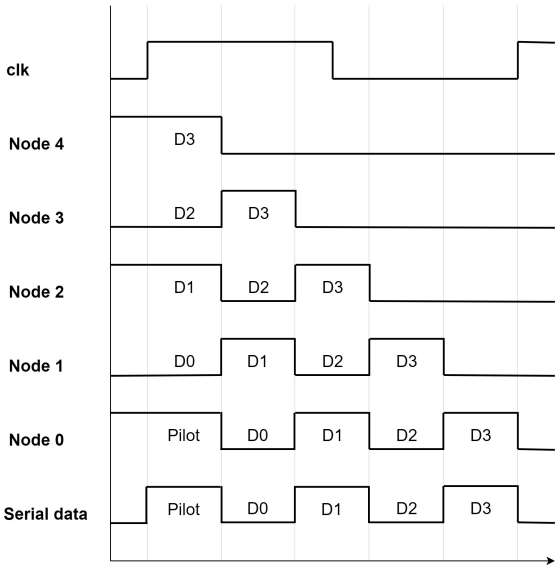


Figure 2.13: The timing for the pilot and data bits in the WAFT solution

Process variations, mentioned in section 2.5.2, are caused by unintentional differences in physical size and other properties of otherwise identical cells. The cause of these variations is minor changes under fabrication, such as line edge roughness and gate thickness fluctuation [32]. As a chip typically consists of many transistors, the likelihood of process variations is high. Process variations impact the propagation delay of any cell. This is the reason process variation is considered in this thesis.

The second variation, called voltage variations, is the variation of the source voltage of the chip. The delay of a cell is dependent on the saturation current. In turn, the saturation current is dependent on the supply voltage. This dependency means that when the supply voltage varies, the delay of the cell is affected.

The last of the variations, temperature variation, generally consists of two temperature variations. The first of the two temperature variations are due to the local density on the chip [33]. As the density of components, and the switching activity of these components, varies throughout the chip, the local temperature naturally also changes. In higher-density areas and high switching areas, the temperature is higher than in lower-density areas and low switching areas. The second temperature variation comes from the fluctuation of the temperature due to switching in each cycle. Both of these temperature variations directly impact the delay of the cell.

Timing uncertainties can generally be split into two categories, Static- and dynamic timing uncertainty. Static timing uncertainty refers to uncertainty due to variations in mean timing caused by slowly varying or one-time effects. An example of static timing uncertainty is the uncertainty caused by process variation. Dynamic timing uncertainty is cycle-to-cycle variations or variations that last for a short period of time [34]. An example of dynamic timing uncertainty is delay changes due to temperature or voltage variations. Timing uncertainties can be classified in one of two ways, skew, or jitter [35]. Skew refers to uncertainty in the relative timing between two signals, often defined as the timing difference between a data signal and its corresponding clock edge, called clock skew. Jitter refers to unwanted variations within a periodic signal from cycle to cycle.

One of the biggest consequences timing uncertainties can have on a wave-pipeline SerDes is related to the functionality and the solution's reliability. If the design is not robust, high static uncertainty can cause the wave-pipeline SerDes to never output the correct data, rendering it useless. While if there is high dynamic uncertainty, the output of the wave-pipeline SerDes might, in the worst case, be correct a fraction of the time.

2.8 Multiple Buffering

As wave-pipeline SerDes heavily relies on propagation for its functionality, there is a fundamental issue that needs to be solved; how can both the serializer and deserializer be reset and ready for a new transmission without reducing the throughput? A solution to this is by using buffers [36]. In this thesis, multiple buffering refers to multiple instances of specific components in both the serializer and deserializer that can be used in parallel for transmissions. Note that only one of these can be used at a time, as there is still only one serial line. An example of how double buffering could be implemented for the WAFT

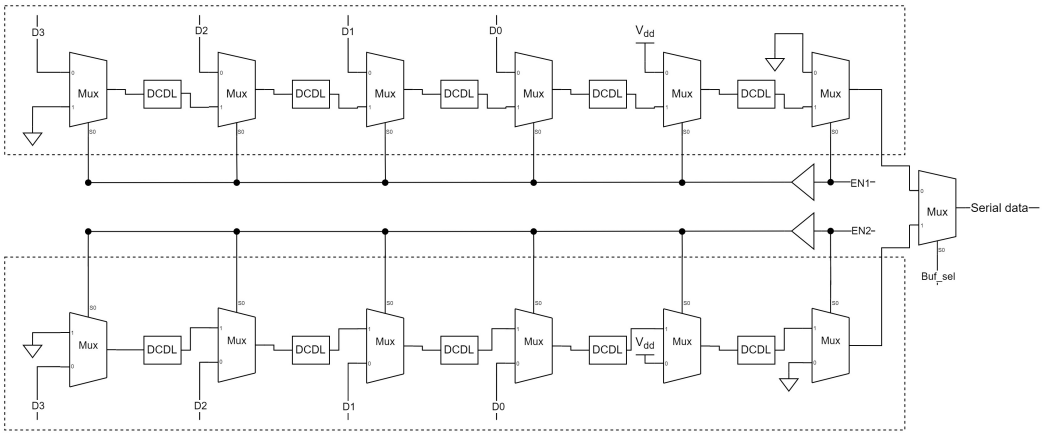


Figure 2.14: Example of how a double buffer could be implemented for the WAFT SerDes solution

solution can be seen in 2.14. There are two sets of multiplexers that both input the serial data. A multiplexer is added to the output of the serializer, which connects the two buffers to the serial line. This multiplexer is used to control which of the two buffers outputs the serial data onto the serial line. Additionally, there are two *EN* signals, one for each, so that the buffer not transmitting does not start a transmission when the other buffer is transmitting. This solution is not a perfect solution, as the two buffers have a set of DCDLs each, increasing the timing uncertainty for the solution. For a wave-pipeline deserializer with double buffering, it can be seen as a cycle where one of the buffers receives the data while the other one resets.

2.9 Timing Calibration

With the high number of delay variations caused by the PVT variations, it is important to consider delay calibration for a delay-sensitive solution such as the wave-pipeline SerDes for higher reliability for the proposed solution [37]. Delay calibration, in this sense, means the possibility of calibrating the solution so that it works as intended for all corner cases, with both static and dynamic variations. In a worst-case scenario, a wave-pipeline solution might not give the correct output at all if the delay is too high or too low in the DCDLs, as a wave-pipeline SerDes is highly dependent on precise delay to function as intended. A delay calibration scheme could be utilized both for initial calibration with the main purpose of rectifying static variations and periodic calibration against dynamic variations to increase the reliability of the solution. Additionally, another possible way of calibrating is by focusing on the variations in source voltage. Both methods will be further elaborated on in this section.

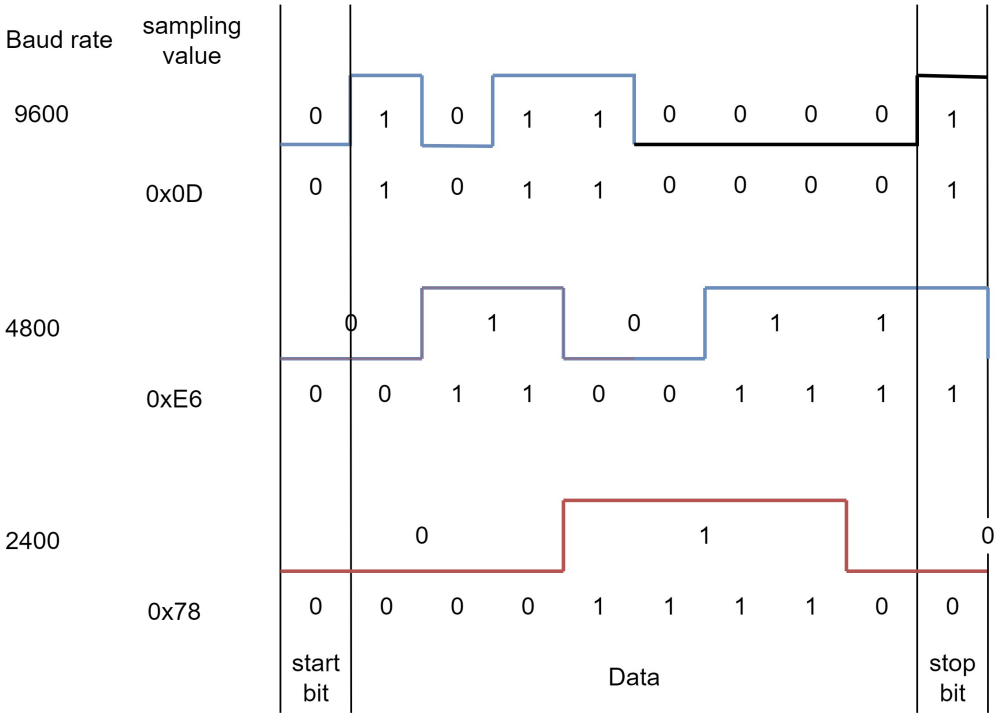


Figure 2.15: Example of autobaud when the transmitter has a lower baud rate than the receiver

The first of the calibration schemes introduced here is the timing calibration scheme used for baud rate solutions, such as the UART solution, named Automatic baud rate detection

(autobaud) [38]. autobaud is a process by which a receiver device can be calibrated by determining the speed of transmission. This is achieved by having a prearranged value that is received using a predetermined baud rate at the receiver, with an unknown baud rate at the transmitter. An example of this can be seen in Figure 2.15. Based on the exact sampled value, it can be determined which baud rate the receiver is using. In this example, the value $0x0D$ is sent from the transmitter, with a sampling baud rate of 9600 in the receiver. If the receiver samples the same value, it is known that the receiver and the transmitter are already using the same baud rate. The same system can be used to determine if the transmitter is using a lower baud rate. The reason for this is that the width of the pulse for each bit is wide enough that they can be sampled multiple times, and as the baud rate of the transmission is constant relative to the baud rate of the receiver, the sampled value is the same each time, for each baud rate. In this example, if the received value is $0xE6$, it can be determined that the baud rate of the transmitter is 4800, and if the received value is $0x78$ the baud rate of the transmitter is 2400. Note that in Figure 2.15 the colors indicate how much of the original data is sampled for the different baud rates. E.g., the blue part of the example with a baud rate of 9600 matches the sampled data for the example with a baud rate of 4800, and the red part in the example with a baud rate of 4800 matches the sampled value of the example with a baud rate of 2400.

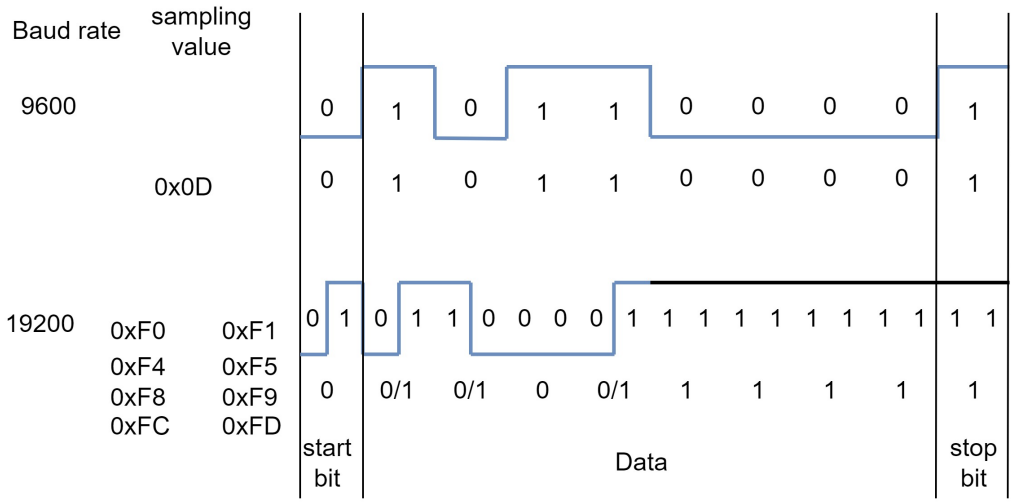


Figure 2.16: Example of autobaud when the transmitter has a higher baud rate than the receiver

Of course, the example above only works for baud rates lower than 9600. But how can autobaud detect baud rates that are higher than 9600? For the example above, the solution depends on the fact that the incoming data has a wider pulse width and that each bit can be sampled multiple times. But turned the other way, if the pulse width of the incoming data is narrower than what is possible to sample, scenarios where the sampling happens just as the serial line switches from one to zero, or vice versa, can occur. With autobaud this is solved by looking at which potential values the different cases for a given baud rate can have. This means that if the baud rate of the transmitter is 19200, there are three points at which the sampling happens while the serial line switches, which gives eight potential values that indicate the baud rate. In this example these eight values are: $0xF0$, $0xF1$, $0xF4$, $0xF5$, $0xF8$, $0xF9$, $0xFC$, and $0xFD$. Figure 2.16 shows a visualization of this.

The autobaud scheme can not be used the exact way described above for the proposed

solution in this thesis. The reason for this is that the proposed solution does not utilize baud rates as predetermined steps but should be possible to use at all clock frequencies. But the autobaud scheme can still be useful regarding the proposed solution. It can be used as a baseline or inspiration for a calibration scheme for the proposed solution. One of the most useful takeaways from the autobaud scheme is that changes are only done in the receiver, while the transmitter remains the same. This will be further elaborated on in Chapter 3.4.

While a timing calibration scheme does cover the three variables in PVT, it is not the only way to perform calibration. As the source voltage variations have the biggest impact of the three variations, it makes sense to focus on these variations [30]. This is what S. Lee et al., the creators of the WAFT focus on. According to them, the voltage variations have a big impact, with a 10% variation of the source voltage in either direction can cause up to 30% jitter for the WAFT deserializer. This is solved by introducing adaptive voltage generation. The voltage generator limits the variations in the supply voltage and smooths it out. Voltage calibration will be further discussed in Chapter 4.5.

2.10 Design for Testability

Testing the functionality of a design is important. If testing of the design is not considered in the design phase, faulty chips might make it to the market. The most common approach to testing digital circuits is to toggle every internal node of the circuit and observe the effect this has. The difficulty of toggling any internal node in the design through the circuit's inputs is called controllability. In the same manner, the difficulty of observing any internal node is called observability [39].

Design for Testability (DFT) is a way for engineers to test a design [40]. By considering testing earlier in the design flow, it is possible to make the design more testable while spending less time doing so. To fully test a design, one would have to be able to control and observe the internal logic of the design. In this thesis, the focus will be on scan chains, which can be utilized to achieve high controllability and observability for the design [41]. Scan chains work by adding a boundary-scan cell that includes a multiplexer and latches to each pin of the design. This collection of boundary-scan cells is then configured into a parallel-in and parallel-out shift register that can be observed. The input part of this shift register can then be loaded with values individually or shifted in through a dedicated input called *test data in*. Similarly, for the output part of the shift register, the data could either be sampled parallel on separate pins or through a serial shift output pin, called *test data out*. It is important to note that the boundary-scan cells do not contribute to the normal functionality of the circuit. Scan-chains will be talked about further in Chapter 3.6.

Chapter 3

Implementation

In this chapter, the proposed solution is presented. The presentation starts by introducing the overview of the proposed solution and continues by introducing the different modules of the design in detail.

3.1 Proposed Solution

The proposed solution is a SerDes that utilizes wave-pipelining principles for the serialization process. The solution is capable of a five-to-one reduction of data wires on the bus and is scalable. This scalability is possible because the solution is split into a configurable number of serializer and deserializer units capable of transmitting five bits per clock cycle. Figure 3.1 shows an overview of the proposed solution. The serializer input consists of five bits of parallel data in addition to a Transmitter Valid (TxValid) and a Transmitter Ready (TxReady) signal used for handshaking. Additionally, the system clock and active-low reset inputs to the serializer. The outputs of the serializer consist of the serial line, a synchronization line named *xck*, and a set of ready and valid signals. The deserializer inputs the serial line and *xck* signal from the serializer and outputs the five bits of parallel data in addition to a Receiver Valid (RxValid) signal and a ready signal that is sent to the serializer. The deserializer inputs a Receiver Ready (RxReady) signal, a valid signal from the serializer, a clocking signal, and an active-low reset. Note that the handshaking protocol used for the proposed solution is based on the AMBA AXI handshaking protocol [42]. The handshaking signals will be more thoroughly introduced in section 3.3.3.

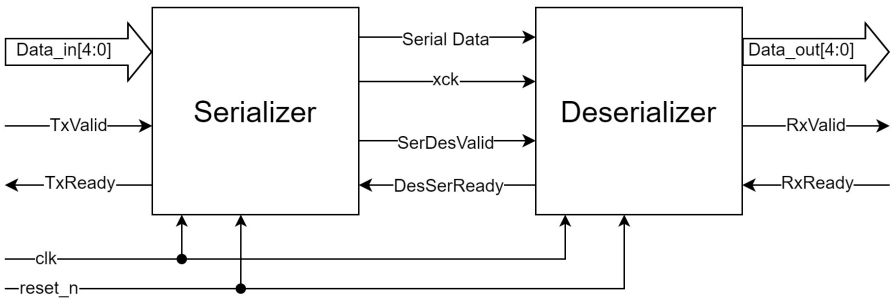


Figure 3.1: Overview of the proposed solution

The design is capable of backpressure, meaning that when the transmitter has data to send that the receiver is not ready to receive, the proposed solution can hold said data for

as long as necessary until the receiver is ready. Additionally, the proposed solution has low latency and is capable of transmitting five bits of data per unit every clock cycle.

As the solution is made up of units that can input up to five bits of data each, a custom parallel width input and output is possible. An example of this is 64 bits. For 64 bits of parallel data, there would be 12 serializer units of five bits each, in addition to a unit number 13 that inputs 4 bits. The deserializer would mirror the serializer.

3.1.1 The Connection Between the Serializer and Deserializer

The connection between the serializer and deserializer can be seen in Figure 3.2. Each of the serializer and deserializer units contains three buffers. Control logic decides which buffers are assigned to transmit the serial data. This works the same for both the serializer and deserializer, though the control logic is separate and not dependent on each other for the two parts. The reason three buffers are used is that this enables the buffers to inhabit different states that cycles for each clock cycle. These three states are: Transmit, offload, and reset. For the transmit state, a new transmission of data between the serializer and deserializer is done. The offload state is the offloading of the data from the deserializer to the output of the proposed solution. And finally, in the reset state, the buffer module resets in preparation for the transmit state. Note that these are not states in a Finite State Machine (FSM), but rather names used to describe the functionality of the different buffers at any given clock cycle.

The source code for the buffering system can be seen in Appendix B.2 for the serializer, and in Appendix C.2 for the deserializer.

Note that this is a simplification of the connection and that the figure is only meant to show a simplified visualization of the connection. What is contained in the buffers, and the control logic for them will be explained in more detail later in this chapter.

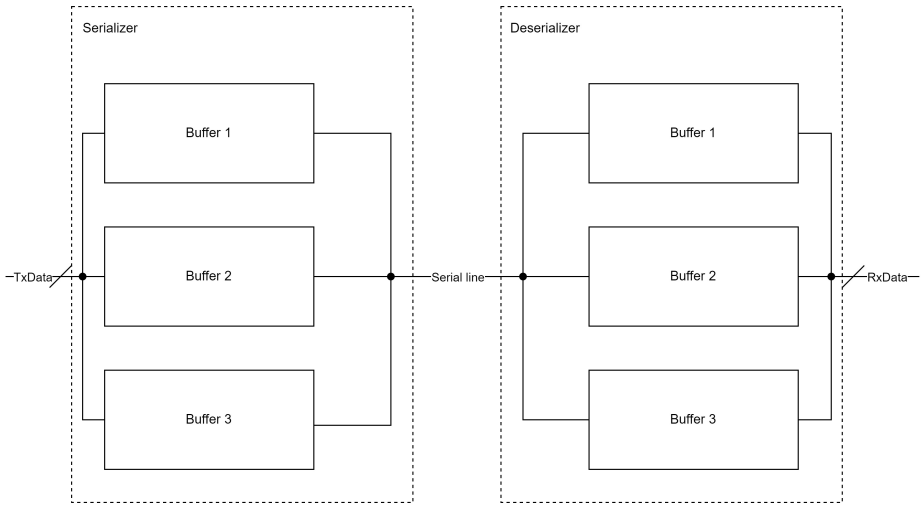


Figure 3.2: A simplified view of the serializer- and deserializer units

3.2 Data Path

The section will describe the datapath for the proposed solution. This is done module by module, starting with the serializer, then the deserializer, and finally the DCDL. The data path for the proposed solution is based on the WAFT solution made by S. Lee Et al., as

mentioned in 2.6.2. This choice is discussed in Chapter 4. Additionally, the source code for the serializer data path can be found in Appendix B.3, and the source code for the deserializer data path can be found in Appendix C.3.

3.2.1 Serializer

The data path of the serializer consists of five registers with an enable input, six two-input multiplexers, and five DCDLs. The proposed serializer can be seen in Figure 3.3. The five bits of parallel input are marked as *D4* to *D0* and are the input of the five registers, *Reg 4* to *Reg 0*. The output of the five registers is then connected to input zero of their respective multiplexer. This makes it so that when the select signal, named *sel_in*, is low, and the serializer is not transmitting data, the five data bits propagates through the first multiplexer but stop at the next multiplexer, as the select signal for the multiplexers is zero. The serial line is connected to ground when the serializer is idle through *Mux0*. When the *sel_in* signal is set high, the multiplexer closest to the serial line switches, and the first bit is transmitted. The *sel_in* signal also propagates through the first DCDL and switches the select signal of *Mux1*, which causes the second data bit to get transferred on the serial line. This pattern continues for each of the data bits until the last bit has been transmitted, and the serial line is set to ground through input one of *Mux5*. Additionally, the *sel[4]* signal is transmitted as the clock synchronization signal *xck* to the deserializer. This makes it so that the clock synchronization signal is sent at the same time as the last bit on the serial line.

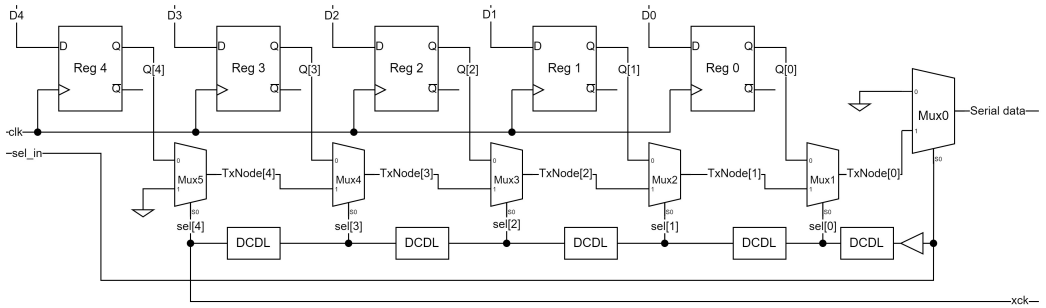


Figure 3.3: The data path for the serializer in the proposed solution

The biggest change, when compared to the WAFT solution, is that the DCDLs have been moved to the select line for the multiplexers instead of on the data line between the multiplexers. The reason for this is that the clock signal *xck* can be timed so that it leaves the serializer at the same time as the last bit, which is D4. The *xck* signal is also the second change compared to the WAFT solution, which instead utilizes a pilot bit, as described in Chapter 2.6.2. These changes will be further discussed in Chapter 4. Note that, depending on how many parallel bits the input of the serializer consists of the number of serial data modules that can be found in the design varies. Each of the modules is capable of serializing five bits of data, as mentioned in the overview above.

3.2.2 Deserializer

The deserializer of the proposed solution, which can be seen in Figure 3.4, consists of five registers with enable, five active high latches, six DCDLs, and an SR AND-OR latch. Note that the SR-latch module and the circuitry around it are simplified in this figure and are explained in more detail in Section 3.2.2. Here, the DCDLs are placed on the serial line as

a separation between the nodes that connect the serial line to the latches, which are called *RxNode4* to *RxNode0* in the figure. An extra DCDL is added to the input of both the serial line and the *xck* input, which are used to calibrate the *xck* to the last bit of the transmitted data. The DCDLs are needed on both the lines so that the calibration can be done both ways. This means two scenarios. The first scenario where the *xck* signal is early compared to the last bit of the transmission, or the second scenario where the last bit of the transmitted data is early compared to the *xck* signal. Note that this only calibrates the data bits with respect to *xck* and that it does not calibrate the timing between the data bits.

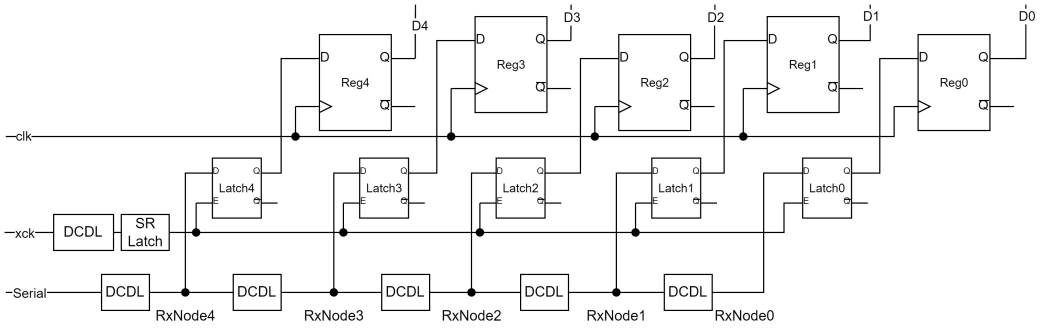


Figure 3.4: The data path for the deserializer in the proposed solution

In addition to the DCDL for calibration, the serial line is connected in serial through four DCDLs. These DCDLs separate the nodes where the output latches are connected. When the last bit of the transmission reaches *RxNode4*, the *xck* signal reaches the SR AND-OR latch, which sets the enable signal for the output latches high, sampling and holding the input value at the rising edge of the enable signal. At the same time, the enable for the five output registers is set high, and the value from the output latches is sampled by the output registers.

One of the biggest differences between the proposed solution and the WAFT solution is that the latching method of feedback signals on multiplexers from the WAFT solution is replaced with active high latches. This will be further discussed in Chapter 4.

SR-latch

Figure 3.5 shows the SR-latch and the circuitry surrounding it. While the correct term for this circuitry is SR AND-OR latch, for simplicity it will be called SR-latch in this thesis. The set signal, named *S* here, is connected to the *xck* signal that comes from the serializer through a DCDL, as mentioned in the section above. This means that, with a small propagation delay from the logic gates in the SR-latch, the output is set high as soon as the *xck* signal arrives at the *S* input of the latch. This causes an enable signal, named *EN* to be set high, which is used to enable the latches that can be seen in Figure 3.4, named *Latch0* to *Latch4*. This locks the output of the latches with the value on the input at the time the *EN* signal was set. The *EN* signal is also connected to the enable signal for the parallel output registers in the deserializer, causing them to start sampling on the positive clock edge of *clk*. Additionally, the output of the SR-latch. called *sel*, is the input of a delayed reset block, which is also clocked by *clk*. The delayed reset block is used to delay the signal by two clock pulses. This block contains a counter that is set to zero when idle, and starts incrementally counting up when the *sel* signal is set high. When this counter reaches a value of two, the output of the block is set high. This output is connected to the reset input, named *R*, of the SR-latch. This

causes the output of the SR-latch to reset back to zero, which, in turn, causes the counter to reset to zero.

Figure 3.1 shows the truth table for the SR-latch. When both inputs S and R are low, the output of the SR-latch does not change. When the set signal S is set high, the output of the SR-latch, named Q, is set high, and when the reset signal R is set high, Q is set low. Note that Q is set low regardless of what S is when R is set high, as the R signal has priority

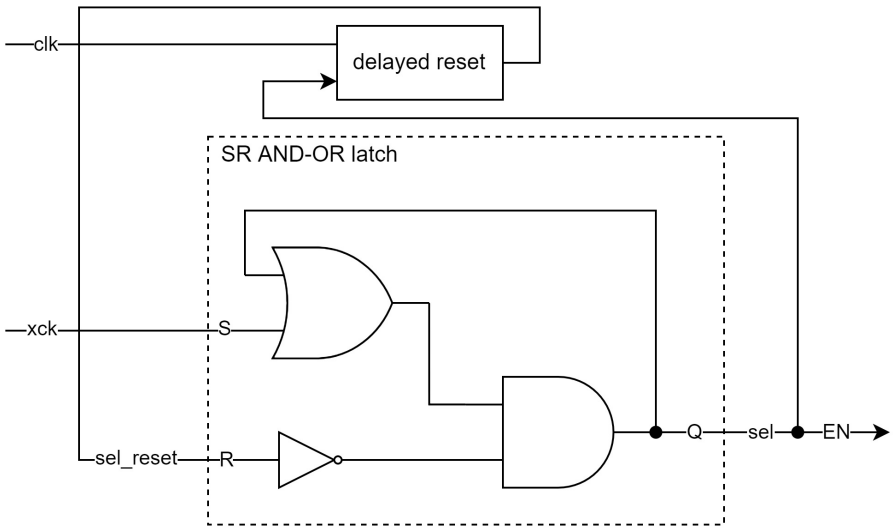


Figure 3.5: The SR AND-OR latch used in the deserializer and the circuitry surrounding it

the S signal for this type of SR-latch. The fact that the Q signal does not change when both over the S and R signal is low is important for the functionality of the SR latch with respect to the deserializer. As the SR latch only needs the positive edge of the *xck* signal to trigger, the length of the pulse is not important, within reason, for the functionality of the deserializer. On the other hand, as the R signal has priority over the S signal, if the *xck* signal for any reason is high for a long period, the functionality of the deserializer is not compromised by this either, as the Q signal is set low regardless of what the S signal is.

Table 3.1: The truth table for the SR AND-OR latch

S	R	Q
0	0	NO CHANGE
1	0	1
x	1	0

How the clock signal is received is one of the biggest issues with the proposed solution. As mentioned in Chapter 2.4.3, the width of the clock signal must be wide enough to be reliably detected. At the same time, the timing of the clock signal in relation to the data is sensitive and needs to be precise. There exist certain scenarios where the input register for the sel signal for the counter in the delayed reset block, shown in Figure 3.5, can enter a metastable state. This meta stable state would then, in the worst case, cause the *sel* signal to remain high for one clock cycle longer than intended. The situation where the metastable state can be an issue is if the *xck* signal enters the SR-latch and propagates through it inside the setup time of the register, causing the sampling value to be an unknown value between zero and one. If this value is closer to zero, the *sel_reset* signal remains low for an additional clock cycle. This lasts until the input register can input the right value. This, as stated above, can cause the *sel* signal, and with that, the *EN* signal, to remain high for a clock cycle longer than intended. This issue can be caused by certain clock frequencies of the system clock, in

addition to the length of the wire between the serializer and the deserializer, as this dictates the propagation delay for the signal before it reaches the SR-latch. This might be a challenge for certain high-speed systems, which is worth keeping in mind. The source code for the SR-latch can be found in Appendix C.5, and the control signals in the delayed reset block can be found in Appendix C.3.

3.2.3 Digitally Controlled Delay Lines (DCDL)

The DCDL configuration chosen for the proposed solution is identical to the design introduced in Chapter 2.5.1, and can be seen in Figure 2.8. The DCDLs used in the serializer and deserializer are identical, but unlike for the WAFT solution, for the proposed solution, the DCDLs are not mirrored. This means that they are not placed at the same spot in the serializer and deserializer but rather on the select signal for the multiplexers in the serializer and the serial data line in the deserializer. This is not a problem here, as the DCDLs in the serializer affect the delay of the serial data as if they were on the serial line. The DCDL solution was implemented to be customizable, meaning that the number of steps can easily be changed for how many are needed for any given system clock and library. The number of select signals dynamically changes with the need as more steps are added or taken away. As mentioned in Chapter 2.5.1, the chosen DCDL scheme has a potential for glitching, this will be further discussed in Chapter 4. The source code for the DCDL module can be seen in Appendix D.

3.3 Control Signals

In this section, the control signals for both the serializer and deserializer are presented. This includes select signals for the buffers, the handshake signals, and the calibration scheme. Source code for the serializer control signals can be found in Appendix B.4 and the source code for the deserializer control signals can be found in Appendix C.4.

3.3.1 Control Signals for the Serializer

The control circuitry for the serializer is significantly larger than for the deserializer. One of the biggest contributors to this is the FSM which does most of the controlling of the transmissions. The reason for this is that the serializer is the module that controls whether a transmission should happen based on handshake signals from the deserializer and the transmitter. The FSM for the proposed solution is to a high degree inspired by the FSM in the xor-based solution mentioned in Chapter 2.6.1.

There are three states in the state machine: Idle, wait, and transfer. The idle state functions as a state in which the design waits for any incoming parallel data from the transmitter. The second state, called wait, is entered when the receiver initiates a wait state. This state enables backpressure. The last state, transfer, is the state in which the design transfers data. If the FSM stays in the transfer state, a new transmission is started each clock cycle of the system clock. The state diagram for the FSM can be seen in Figure 3.6. The state of the FSM switches based on the two inputs TxValid signal, which is the valid signal from the transmitter, and the *DesSer_Ready* signal, which is the ready signal sent from the deserializer to the serializer. If TxValid is set low, and the FSM is not in the idle state, the state is changed to idle on the next clock cycle, regardless of which of the two other states it is currently in. This is because TxValid being set low indicates that the transmitter does not have data to send. If the *DesSer_Ready* signal is low while the

TxValid signal is changed to high, it means that the transmitter is ready to send data while the receiver is not ready to receive the data. This causes the FSM to change state to wait, where it stays until one of two scenarios. The first of these scenarios is that the receiver is ready and the *DesSer_Ready* signal is set high. This causes the FSM to change state to transfer. In the second scenario the transmission is stopped by the transmitter setting the TxValid signal low. This causes the FSM to change state to idle. Similarly, if the FSM is in the transfer state, and the *DesSer_Ready* signal is set low, the FSM changes the state to the wait state. The outputs of the FSM include the enable signal for the parallel input of the serializer, the TxReady signal, and a *SerDes_Valid* signal which is sent to the deserializer for the handshake.

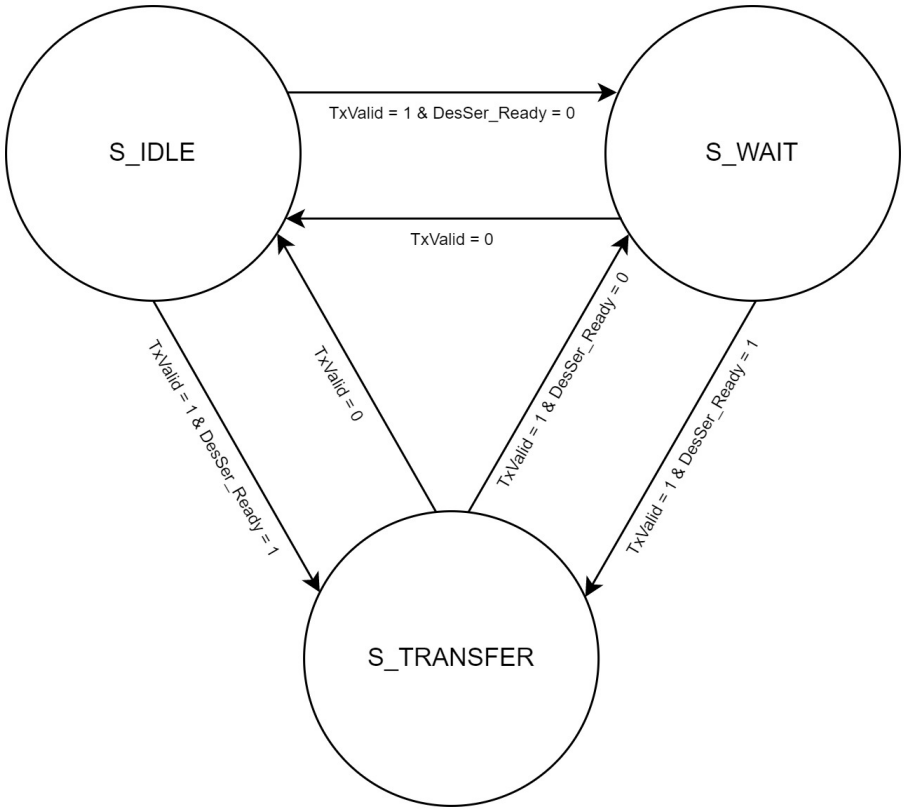


Figure 3.6: The Finite State Machine for the control signals of the serializer

The buffer selection is done using a counter that count from zero to two and is reset zero. As there is minimal switching when the serializer is idle, the counter keeps cycling through these three values. This is also true while the FSM is in the transfer state, as this means that data is continuously transmitted. When the FSM is in the wait state, the counter is paused. The reason for this is that the data which is waiting to be transmitted is held in the parallel input registers of one of the three buffers, causing a need to track which of the three buffers holds the data. When the state switches from wait to transfer, the counting resumes. This counter value is used for two things. Firstly, to set one of three select signals, called *sel*, high. This signal is the select signal that is sent to one of the three buffers for the multiplexers and is the signal that propagates through DCDLs, as shown in Figure 3.3. Secondly, the counter value is used directly as a select signal for an output multiplexer that decides which of the three buffers gets to transmit data on the serial line and the clocking synchronization line.

Lastly, the *xck* signal is derived from the select signal after it has propagated through all the DCDLs. This makes it so that the signal is in time with the last bit of the data transmission before it leaves the serializer.

3.3.2 Control Signals for the Deserializer

The control signals for the deserializer are generally significantly simpler than the control signal for the serializer. This is due to the deserializer mostly being passive, which means it, for the most part, only reacts to incoming signals from either the serializer or the receiver.

The RxReady signal, which is the ready signal from the receiver to the deserializer, is directly connected to the *DesSer_Ready* signal, which is the ready signal from the deserializer to the serializer. There are two reasons for this. The first reason for the direct connection is that the deserializer should not receive data if the receiver is not ready to offload it. The second reason is to reduce as much delay between the receiver and the serializer as possible. The RxValid signal is derived from the *SerDes_Valid* signal, which is the valid signal from the serializer to the deserializer. This is done through a register that is clocked by the system clock. This is done so that the transmitting data, which takes a clock cycle, has time to reach the deserializer before the RxValid signal is set high.

There is also a counter in the deserializer, like in the serializer. The counter works the same way as the counter in the serializer; the counter counts from zero to two before it resets to zero. There is a signal that stops the counter if either the RxReady signal or the *SerDes_Valid* signal is low. This is used to stop unnecessary switching when the deserializer is idle, reducing the power consumption. Note that the reduction of power consumption from this is minor. But compared to the serializer, the control logic for the stop signal is limited to a single NAND-gate, which makes it worth it. The counter controls a one-hot signal, named *ser_sel*. The *ser_sel* signal controls the active buffer for each clock cycle. This is done by deciding which of the three buffers gets the serial data input line and the *xck* signal using AND-gates as shown on the left side in Figure 3.7. Two sets consisting of three and-gates, one for the serial data input and one for the *xck* input, has the *ser_sel* one hot signal as the second input with one of the three inputs for each of three and-gates. The output of these two sets of three AND-gates is sent into one deserializer module each. This leads to the buffer that inputs the *SerialDatan* signal and *xckn* signal for the currently active one-hot signal, *ser_sel*, being the only buffer receiving these two signals, with the select signal being one-hot. This means that only one of the three buffers receives the *SerialDatan* signal and the *xckn* signal at a time. Another positive effect is that the two non-active buffers receive a logical zero as input on both these inputs. For the serial data input, this means that there is no need for a reset signal in the deserializer module, as the logical zero output from the AND-gate can propagate through the deserializer and reset it when the deserializer module should be reset.

Additionally, the *ser_sel* signal is used to determine which of the parallel data outputs are to be connected to the output of the design. On the output, the *ser_sel* signal is shifted compared to the input so that the buffer containing the data from the last transfer is connected to the output. If this shift was not there, the wrong data would be outputted. This is done using multiplexers as shown on the right side in figure 3.7. A set of five parallel data lines is outputted from each of the three buffers and sent through a set of five multiplexers that use the same *ser_sel* signals as the select signal. The output of the multiplexers is connected directly to the designated spot in the *RxData* vector, meaning the output for bit zero is connected to *RxData[0]*, which is the output for the proposed solution.

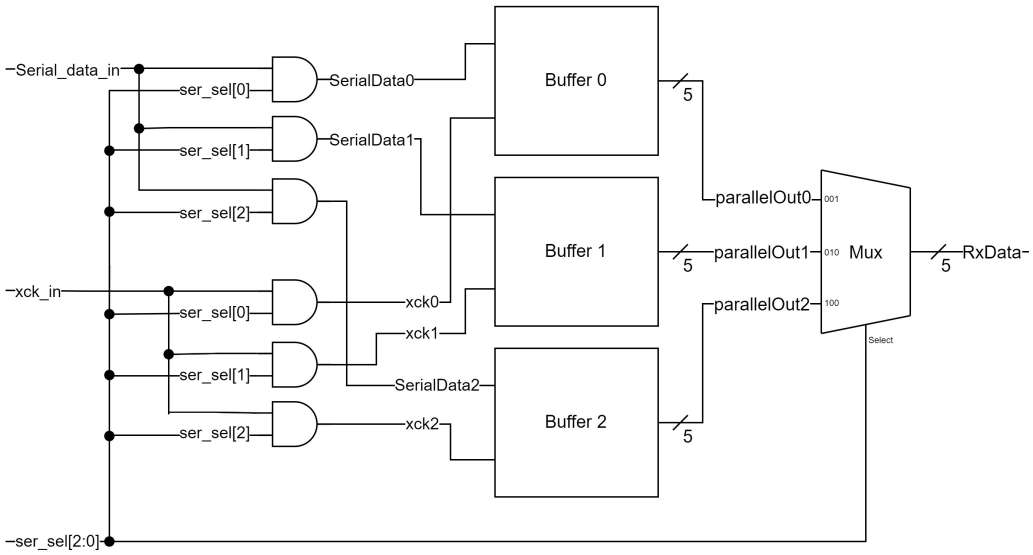


Figure 3.7: How the deserializer buffer input and output are selected

3.3.3 Handshake

As stated at the start of this chapter, the handshake protocol is based on the AMBA AXI protocol [42]. This protocol is the same as the one used in the project preceding this thesis [12].

The protocol consists of a valid signal and a ready signal. The valid signal indicates that valid data is ready to be sent, while the ready signal indicates that the data is ready to be received. This means that the valid signal is always sent from a transmitter to a receiver, and the ready signal is always sent from a receiver to a transmitter. When both the ready and the valid signals are high, a transmission starts. This scenario is called a handshake. Similarly, a beat defines a clock cycle where a handshake occurs. In the proposed solution, handshake signals are used between the transmitter and the serializer, the serializer and deserializer, and the deserializer and receiver.

A transmission is defined as a single transfer of data. If a transmission is started, it must be finished. Additionally, if the valid signal remains high, transmissions continue for as long as the ready signal is high or until the valid signal is set low. Both the valid signal and the ready signal can be set high regardless of what the other signal is at any time. If the ready signal is set low, backpressure occurs, and the data for the next transmission must be held in the serializer until the ready signal is set high again. Lastly, the ending of transmission is indicated by the valid signal being set low.

3.4 Calibration

As mentioned in Chapter 2.9, calibration of the deserializer in a wave-pipeline SerDes is important for reliability of the solution. The main focus in this chapter will be on how a timing calibration scheme can be implemented.

The proposed solution does not rely on a baud rate system with increments to determine the speed of the system, like the autobaud system. This means that the autobaud scheme can not be directly used for the proposed solution. Instead, a solution inspired by the autobaud scheme will be presented. As the autobaud system is made to be used to synchronize the

baud rate of a serializer to the deserializer, there is a fundamental difference between the proposed solution and a solution that would employ autobaud. For the proposed solution of this thesis, the frequency is known when the system is synthesized. A calibration scheme for the proposed solution would rather be used for small-scale calibration rather than for big changes like what was explained with the examples in Chapter 2.9. This difference should be evident in how the calibration scheme is implemented for the proposed solution in contrast with how it would be implemented for a system utilizing baud rate with bigger incremental changes.

Firstly, keeping the serializer unchanged while changing the delay values for the deserializer is a principle that can be used for the proposed solution, like with the autobaud system. This principle makes it so that the delay of the serializer can be used as a constant reference while the deserializer is adjusted to match it. Secondly, transmitting a given value that can be used as an identifier in the deserializer could also be used for the proposed solution.

Figure 3.8 shows a visualization of a calibration scheme for the proposed solution with these two ideas combined. The figure shows the received data for three scenarios. The five-bit value sent from the serializer in this example is 11011. In scenario one, the data received in the deserializer matches the value sent from the serializer, and no adjustments are needed. In scenario two, the value received is shifted by one bit to the left. This shifting means that the delay in the deserializer is too low, and the DCDL delay path for the deserializer should be longer to make up for it. In scenario three, the data received in the deserializer is shifted one bit to the right compared to the data sent from the serializer. In this scenario, the delay in the deserializer is too high, and the DCDL delay path for the deserializer should be shorter. Note that this is only one of many ways the calibration scheme could be implemented, and it is not without shortcomings. The biggest shortcoming is that the calibration scheme only works for small changes. If the delay in the deserializer is much higher or lower than the serializer, the received data could consist of five zeros. This happens because the serial line is grounded through *Mux0* while idle, and grounded through *Mux5* after the transmission is completed, as shown in Figure 3.3. Note that if there is only a single 1 in the sampled value in the deserializer, the course of action can be determined. But these shortcomings do not mean that this calibration solution is not usable with the complete proposed solution. As mentioned earlier in this section, the frequency of the system clock is known at the time of synthesis. This means that the calibration solution should only ever be needed for small increments of calibration.

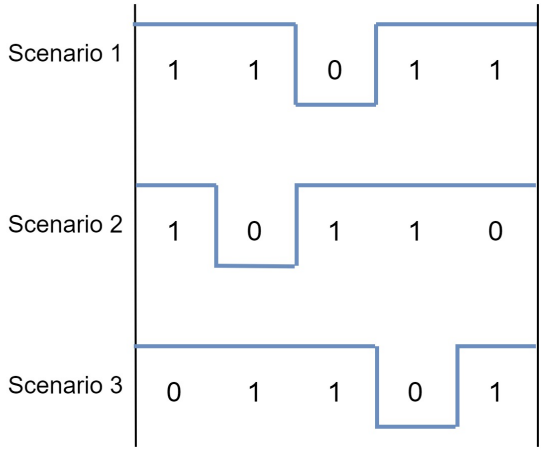


Figure 3.8: A visualization of an example of how a calibration scheme could be implemented for the proposed solution

There are some additional questions that need to be answered about the calibration scheme. One of which is: When should the calibration be done? The first and maybe most obvious answer to this question is initialization. At start-up, the variations due to PVT are unknown. This is mainly due to process variation, which can have a major impact on the total delay variation. A timing calibration done at initialization would negate the impact of the static variations on the proposed solution, only leaving the dynamic variations. As for

the dynamic variations, there are two ways calibration can be done. First of which is at reset. If the reset signal is set, a timing calibration is triggered. The problem with a reset-triggered timing calibration is that it would require something or someone to set the reset signal, which would be avoided with the second solution. The second solution is to trigger a timing calibration at a given interval. An example of this is that a timing calibration is triggered after every n transmission, where n is the number of transmissions between each timing calibration. This would require a control system that keeps track of how many transmissions have occurred since the last calibration, which takes up area, and increases the power consumption, but it would be working with no outside influence. How many transmissions there are between the calibrations can be decided based on the frequency on the system clock, which process node the proposed solution is running on, or similar metrics to optimize the calibration for the specific design of the proposed solution.

The second question is how to communicate between the serializer and deserializer when the calibration is starting. As a calibration transmission should not be outputted from the deserializer, the RxReady signal is irrelevant for calibration. This means that a calibration transmission can be initiated at any time, as long as a normal transmission is not currently transmitted. Additionally, the deserializer does not have to transmit anything to the serializer in regards to the calibration, but the deserializer does have to be notified that the calibration transmission is not a normal transmission. The simplest way to do this is to start the transmission without setting the *SerDes_Valid* signal. This idea works because the latches in the deserializer are controlled by the *xck* signal that is sent from the serializer, in addition to the fact that the deserializer should always be ready because there is no need to output anything from it to the receiver.

3.5 Timing for the Proposed Solution

In this section, the timing for the proposed solution is presented. This includes timing for the five individual bits and how and why they are affected by different aspects such as frequency and timing uncertainty. Additionally, an example using an imaginary standard cell library is conducted to find the number of DCDL steps needed for this example.

The delay path for *D0* between the input register and output register can be seen in Figure 3.9. *D0* is the bit closest to the serial line, which means that the delay path for *D0* in the serializer, is the shortest. In turn, the delay path of *D0* in the deserializer is the longest, as it is the only bit that has a delay path through all five DCDLs in the deserializer. In total, the propagation of *D0* consists of one multiplexer, five DCDLs, and a latch. While *D0* does propagate through *Mux1*, this happens before the transmission starts, which is the reason why it is not included. Equation 3.1 shows the total propagation delay for a data bit in the proposed solution. n is the bit number of the individual bit, e.g., n is zero for *D0*, n is one for *D1*, etc. It should be noted that the propagation through the multiplexers is slightly different as the signal propagates from input 0 to the output and input 1 to the output of the multiplexers. But for this equation, the propagation delay is simplified to be equal for both. There is a slight difference in the propagation delay for the 5 data bits. This is due to the multiplexers in the serializer that does not have a mirrored counterpart in the deserializer. This means that the DCDLs in the deserializer need to account for this by having added delay equal to D_{mux}, when compared to the DCDLs in the serializer. Equation 3.1 does not show this. Additionally, each bit must propagate through four DCDLs in addition to a small DCDLs used for calibration, named $D_{DCDLcal}$. The $D_{DCDLcal}$ can be found in the deserializer between the serial line and *RxNode4*, and can be seen in Figure 3.4. D_{total} is also

the minimum time before the latch can be locked after a transmission start. This means if it is locked before this time, the data will not be sampled.

$$D_{total} = (n+1) \cdot D_{mux} + 5 \cdot D_{DCDL} + D_{DCDLcal} \tag{3.1}$$

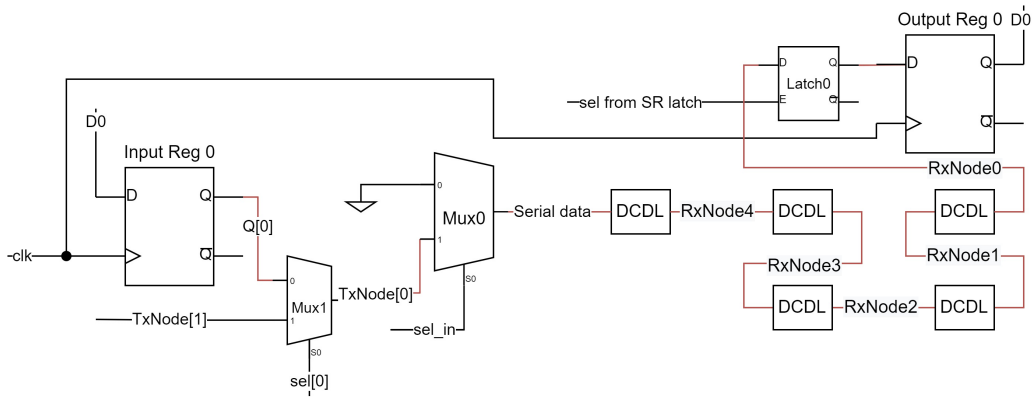


Figure 3.9: The delay path for *D0* from the input register to the output register with the delay path marked in red

With an example system clock frequency of 1 GHz, equations for the total delay for each of the five bits can be made. It is, however, important to note that these equations are dependent on the standard cell library used for the implementation. Here in this example, an imaginary library will be used. Firstly, a clock frequency of 1 GHz gives a period of 1 ns. This means that with a requirement that the transmission should finish in a clock cycle, the total propagation delay for the five data bits should not exceed 1 ns. This means that the total pulse width for each bit can not be higher than 200 ps. This works out as each of the five bits has to propagate through the system with a near-constant delay separating them. Additionally, the different bits need to propagate through multiplexers. *D4* must propagate through six multiplexers, *D3* through five multiplexers, etc. *D0* has the fewest multiplexers to propagate through, with only two multiplexers. The difference that this causes must be added to the DCDLs in the deserializer as mentioned above. With this added difference, it is assumed in the equation that each bit propagates through six multiplexers. Additionally, each bit must propagate through a latch to reach the input of their respective register. But note that this is not important in regards to the transmission as the latch samples the value and is seen as the end of the propagation path, while the register samples the value on the next clock cycle. This assumption can be seen in Equation 3.2.

$$D_{total} = 6 \cdot D_{mux} + 5 \cdot D_{DCDL} + D_{DCDLcal} \tag{3.2}$$

In this example, the propagation delay for a multiplexer, D_{mux} is 9.5 ps, and the propagation delay of a NAND-gate is 3.8 ps. The DCDLs used for calibration is assumed to be small, with only a single step. Note that this is just a choice made for this example, and that this calibration DCDL can be adjusted to any value just as the other DCDLs. D_{DCDL} can be found using Equation 2.9. The total propagation delay for the multiplexers in this example is 57 ps, and the propagation delay for the calibration DCDLs is 7.6. The results of this inserted in Equation 3.2 can be seen in Equation 3.3. Note that $D_{total} = 1000$ ps is the max propagation delay, mentioned above. This leads to a total delay per DCDL of 187.08ps, which can be seen in Equation 3.4. This total DCDL delay is also the bit width for each bit,

as this is the propagation delay of the DCDLs on the select line in the serializer. Inserting this value as the total propagation delay for a DCDL in Equation 2.9 and solving for n gives a result of $n = 23.6$, which can be seen in Equation 3.5. This result gives a total steps per DCDL of 24 steps, as steps in the DCDL can only be full steps.

$$1000ps = 57ps + 5 \cdot D_{DCDL} + 7.6ps \tag{3.3}$$

$$D_{DCDL} = \frac{1000ps - (57ps + 7.6ps)}{5} = 187.08ps \tag{3.4}$$

$$n = \frac{187.08ps - 7.6ps}{7.6ps} + 1 = 24.6 \tag{3.5}$$

3.6 Design for Testability in the Proposed Solution

The proposed solution is also made with DFT in mind. More precisely, it is made with scan chains in mind. For the proposed solution, there are clear input points for boundary-scan cells, which can give a high testing coverage, and with that, high observability. An example of such a point is the serial data and the *xck* output line for the serializer. Connecting the scan chain at this point gives access to observe the complete transmission and the *xck* signal. In the deserializer, the output of the latches can be similarly observed using boundary-scan cells.

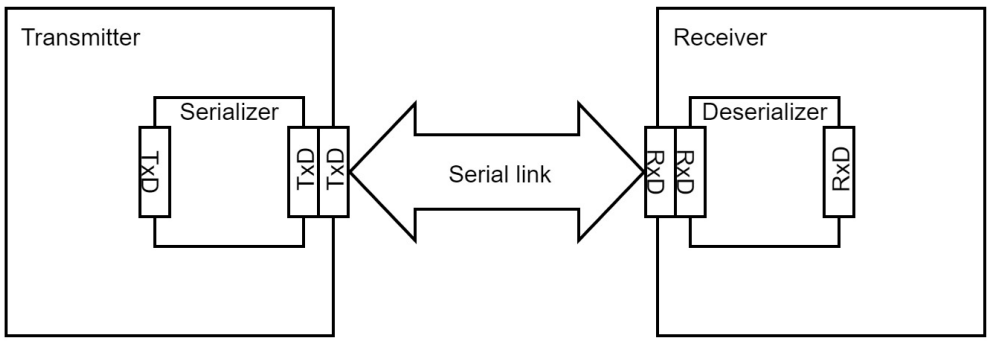
Chapter 4

Results and Discussion

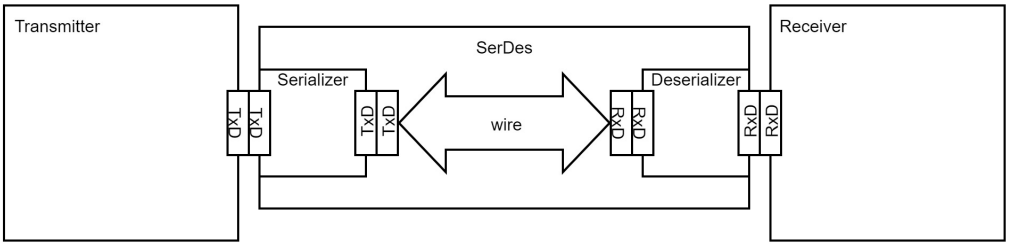
This chapter starts by presenting the methodology for the proposed solution. The results from both simulation, synthesis and routing follow. Additionally, discussions regarding both design choices, the results, and general discussion about the proposed solution are found in this chapter.

4.1 Methodology

This section focuses on the methodology for the proposed solution. This includes some decisions made in regards to the methodology, in addition to important information needed to recreate the proposed solution.



(a) Serializer as a sub-module of the transmitter, and deserializer as a sub-module of the receiver, connecting the two using a serial link.



(b) Placing the proposed design in a sub-module separate from the transmitter and receiver.

Figure 4.1: Alternative ways to connect the modules in the proposed solution.

One of the first considerations to make in this thesis was the module setup for the proposed solution in regards to the preexisting transmitter and receiver. Two alternative solutions will be discussed here. Both solutions can be seen in Figure 4.1. The first of which, as can be seen in 4.1a, splits the SerDes into two sub-modules: Serializer and deserializer. These two modules are then inserted in the transmitter and receiver in their respective places, with a serial link connecting the serializer in the transmitter to the deserializer in the receiver. For the second solution, which can be seen in 4.1b, the proposed solution is packaged as a separate sub-module of the system with an interface to the transmitter and the receiver. For the solution of this thesis, version b was selected. The reason for this choice is to make it easier to package and implement the solution in the system. It is also easier to make changes to and testing of the proposed solution when it is contained in the system as a separate entity. It is also possibly a more intuitive packaging option that is easier to navigate. Another positive side effect of this choice is the possibility of making the solution into an IP-block. There are some routing concerns related to the packaging choice, but that is outside the scope of this thesis.

The module hierarchy of the proposed solution can be seen in Figure 4.2. In relation to Figure 4.1, serdes.sv is the serdes top module, and the two modules serializer_wavepipe.sv and deserializer_wavepipe.sv are the Serializer and Deserializer modules respectively. The serializer_wavepipe module and the deserializer_wavepipe module both contain two sub-modules: A control module and a data module. The data modules are named Serializer_multi_buf for the serializer and Deserializer_multi_buf for the deserializer. These two modules are the modules that act as entities that can be multiplied for any number of parallel inputs, as explained in Chapter 3. These modules also contain the three datapath buffers, which in turn contain the DCDL sub-modules for the serializer and the DCDL and SR_latch sub-modules for the deserializer. A note here is that the control module for both the serializer and deserializer are shared between the three data path buffers, which is done for lower power consumption and area usage, and works for this solution as the three buffers never transmit data simultaneously.

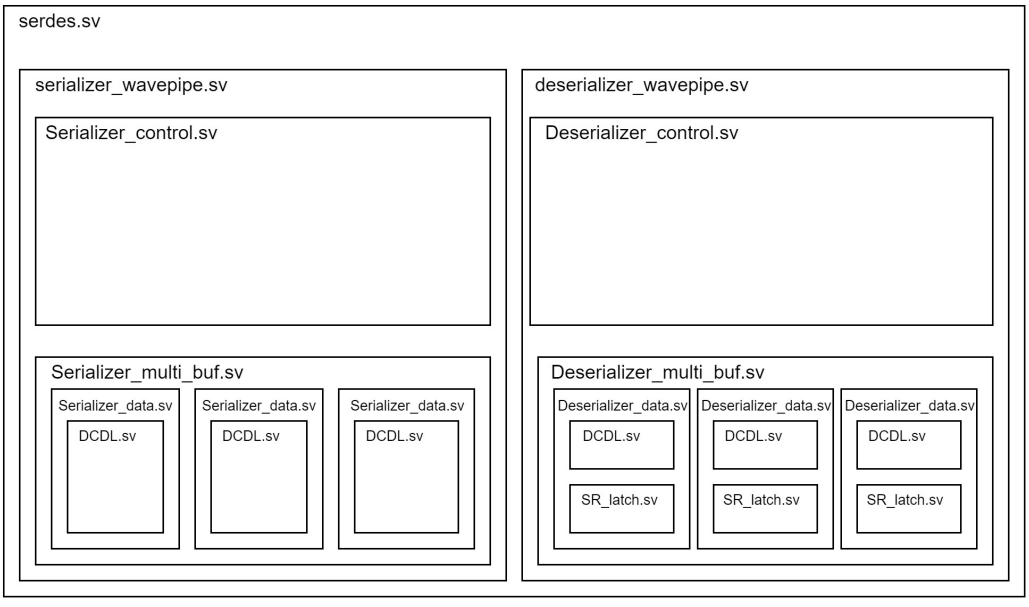


Figure 4.2: The modules of the proposed solution

The proposed solution was implemented using the HDL SystemVerilog, as this was a requirement for the thesis. The solution was simulated using Xilinx Vivado's integrated simulation tools, while Cadence Genus was used for synthesis. The standard cell library used for the synthesis is a 28 *nm* process. Due to confidentiality, no more information about the library can be provided in this thesis. Tool Command Language (tcl) scripts were set up for the synthesis of the design, which, for the most part, simplified the synthesis. tcl scripts, in regards to synthesis, are simple scripts containing the different console commands used to run the synthesis. The synthesis of the proposed solution was a bigger part of the thesis than first anticipated. The main reason for this is the DCDLs. Any synthesis tools will try to optimize the design while maintaining the logic function of the circuit. This means that any delay element, as described in Chapter 2.5, that does not have any logical functionality, will be optimized to a wire, making it so that delay elements are not synthesizable out of the box. In turn, when the delay elements are optimized away, some of the logic connected to the delay elements might not drive any primary outputs, which also causes the synthesis tools to optimize the circuit removing these parts too. Figure 4.3 shows a console message about this optimization. In this case, the DCDLs in the deserializer has been removed, which causes the output registers to be removed because there is nothing connected to the *D* input of the registers.

```
Info    : Deleting instances not driving any primary outputs. [GLO-34]
        : Deleting 317 hierarchical instances.
Following instances are deleted as they do not drive any primary output:
'u_deserializer_wavepipe/genblk1_G_0___genblk1__U_des_multi/genblk1_G_0___U_Ser_data/mux_Q_74_16',
'u_deserializer_wavepipe/genblk1_G_0___genblk1__U_des_multi/genblk1_G_0___U_Ser_data/mux_Q_74_17',
'u_deserializer_wavepipe/genblk1_G_0___genblk1__U_des_multi/genblk1_G_0___U_Ser_data/mux_Q_74_18',
'u_deserializer_wavepipe/genblk1_G_0___genblk1__U_des_multi/genblk1_G_0___U_Ser_data/mux_Q_74_21',
'u_deserializer_wavepipe/genblk1_G_0___genblk1__U_des_multi/genblk1_G_0___U_Ser_data/mux_Q_74_24',
'u_deserializer_wavepipe/genblk1_G_0___genblk1__U_des_multi/genblk1_G_0___U_Ser_data/mux_count_44_17',
'u_deserializer_wavepipe/genblk1_G_0___genblk1__U_des_multi/genblk1_G_0___U_Ser_data/mux_sel_reset_44_17',
```

Figure 4.3: Synthesis console message about optimization of logic due to optimized delay elements

To solve these problems, the synthesis tool, in this case, Cadence Genus, needs to be notified that the delay elements should be preserved or that they should not be touched. The synthesis tools provide multiple ways this can potentially be done, which will be discussed here. In Figure 4.4 different design objects can be seen. Design objects are identifiable objects in the design that can be manipulated using attributes during synthesis. However, the focus of this thesis will be on attributes related to preserving design objects. Preserving design objects prevents the synthesis tool from replacing or modifying the design object during optimization. First off is an attribute named *set_dont_touch_network*, which is mainly used to set the don't touch attribute on clock networks and the buffers on the clock network. The attribute can only be set on clocks, ports, and pins. While the attribute was extensively researched for this thesis, no way of using it to preserve DCDLs was found. The problem is that the attribute can only be set on design objects that have been mapped by the synthesis tool in the generic synthesis

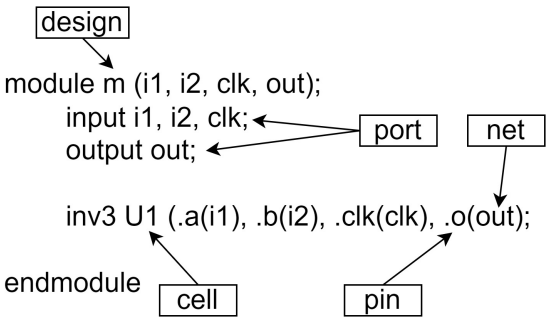


Figure 4.4: The different design objects, as seen from the synthesis tool with an example Verilog code

step, which comes after the optimization step, in which the DCDLs are removed. The second solution named *set_dont_touch* works similarly with the exception that it is more general, as it can set the don't touch attribute on cells, nets, designs, and library cells. While the *set_dont_touch* attribute shares the same problem as the *set_dont_touch_network* attribute for cells, nets, and designs, it does not share the problems for library cells. This means that it is possible to use this attribute to preserve library cells. But to be able to preserve a library cell, one must first instantiate the library cell. And to be able to do so before optimization, the instantiation must be done by hand. Hand instantiation means inserting the library cell into the design by hand. Hand instantiation can be done by instantiating a cell in the design that is connected to a specific cell in the standard cell library, which in the case of the proposed solution is a NAND-gate cell. The source code for this module can be found in Appendix D.2. The *set_dont_touch* can then be set on the library cell, preventing the optimization. Note that for the attribute to affect the optimization, the command must be inserted between the elaboration step, and the generic synthesis step.

The proposed solution also contains combinatorial loops, which can cause trouble in relation to the synthesis tools. The reason for this is that for the timing analysis, the start of the loop can not be determined, as it would normally be defined as the output of a register or latch. For the case of Cadence Genus, the combinatorial loops are disabled using loop breakers, which break the circuit and interfere with the functionality of the solution. The loop breakers have no impact on the optimization of the circuit, and with a single command, the loop breakers can be disabled. For the proposed solution, this was done at the end of the synthesis.

4.2 Important Design Choices

In this section, important design choices that were made for the proposed solution are presented and discussed. This includes clock signaling, the choice of DCDL scheme, how the buffering system was implemented, the calibration scheme, and other miscellaneous choices that were made for the proposed solution.

4.2.1 Choice of Mesochronous Clock Scheme

The choice of mesochronous clock scheme was between the two types described in Chapter 2.4. The first scheme, called receiver clock generation, described in section 2.4.2, can be a robust solution. This solution was not chosen because the clock generation circuitry would add to the area of the proposed solution while also giving a significant increase in power consumption. This leaves the source-synchronous clock scheme, as described in section 2.4.3. There are two ways of implementing a source-synchronous scheme. The first is the pilot bit signaling solution, which looks the best at first glance as it requires no additional wires on the bus while working similarly compared to the scheme that utilize a separate clock line. However, if the pilot bit signaling scheme were to be chosen, there is a hurdle that needs to be overcome in regards to the proposed solution. As the pilot bit is inserted in the data transmission, and the total width of transmission is dictated by the clock frequency, as described in Chapter 3.5, the width of the pilot bit pulse can become so narrow that there is no way of consistently detecting in the deserializer, without added significant complexity, area, and power consumption for the proposed solution. Additionally, the pilot bit is inserted on the serial lines, which means that it is taking up time from the serial line that could be used for an additional data bit. This means that with the reduction of units needed, as each unit can transmit an addition bit, the total number of wires is limited with a separate

clock line scheme. Note that does not mean that there are fewer total lines than for a pilot bit scheme, but fewer than if the number of data bits stay the same. Because of these two reasons, the separate clock line scheme is favored for the proposed solution, and both reasons have a major impact on the proposed solution. This is also the reason the separate clock line scheme was chosen for the proposed solution.

4.2.2 Choice of DCDL

There are many ways of implementing DCDLs. Three of these solutions were introduced in Chapter 2.5. In this section, the choice between these three solutions is explained.

As stated in Chapter 2.5.1, the propagation delay through an inverter is typically low. This means that the number of inverter pairs needed for the total delay is high. Additionally, the leakage power of an inverter is high compared to a NAND-gate, based on the library the author of this thesis had access to. The reason for this is that the propagation delay to power consumption ratio is higher for an inverter, compared to a NAND-gate. As there is a higher total number of inverters in an inverter-based solution, than NAND-gates in the NAND-based solution, the total power consumption from leakage power is higher for the inverter-based solution. Similarly, while the area of a single inverter is lower than a NAND-gate, the increase in the total number of inverters needed compared to a NAND-based solution leads to a higher area for the inverter-based solution. When these points are added, a NAND-based solution is the better choice for the DCDLs in the proposed solution.

It should also be noted that the reason that the solutions presented in Chapter 2.5 is either based on inverters or NAND-gates is that they both invert the signal and take up a smaller area when compared to gates such as AND-gates. As described in Chapter 2.5.2, the fact that the elements invert the signal between the stages is important as there is a difference in rise-time and fall-time for the gates. When the signal is inverted between each stage, this difference between rise-time and fall-time is averaged, and it does not matter if the input is a one or a zero.

As it has been established why the choice between inverters and NAND-gates ended with NAND-gates, the last choice made was between the glitchy NAND-based solution, and the glitchless type, as presented in Chapter 2.5.1. The first difference is that the glitchless solution does not have the potential issue of having multiple delay paths connected to the output at the same time, which can introduce glitching. To stop this potential glitching problem, additional NAND-gates, with an extra set of select signals, are inserted to block the signal from propagating through the wrong step when switching occurs. The extra NAND-gates that are inserted add to the delay of the DCDL, while also adding to the total area for the DCDLs when compared to the glitchy solution. Additionally, the power consumption is increased as there are extra NAND-gates with leakage power and internal power.

The glitching issue for the glitchy NAND-gate is only a problem if the DCDL is actively in use when the select signal changes. This means that there is no danger of glitching for the proposed solution, as the select signal never changes while there is an active signal. This reason makes the glitchless solution obsolete if the reason it was chosen is that it is glitchless. However, as the propagation delay of cells is dependent on the standard cell library it comes from or the process technology of the library, the glitchless solution might be a good solution if the desired total delay of the DCDL is high or if the resolution of the DCDL steps is desired to be higher. The glitchy DCDL was chosen based on the propagation delay of the NAND-gate in the library the author had access to, in addition to the fact that glitching is not an issue for the proposed solution. But do note that the glitchless solution might be a good solution for some standard cell libraries.

4.2.3 Choice of Buffer Scheme

In this section, choices related to the buffer scheme are discussed. This includes control signal choices and choices related to what the buffers consist of.

There were initial concerns regarding the looping counters for selecting the active buffer. This concern was related to the counters continuously counting. The reason for this concern was the potential high switching activity that could unnecessarily increase the power consumption of the solution. Using AND-gates for the input, as shown in Figure 3.7, removed any such concerns. The reason is that it meant that, outside of the switching related to the counter itself, no extra switching activity happens. Outside of enabling functionality, such as backpressure mentioned in Chapter 3.3.1, there is no need to be able to stop the counters. This fact simplifies the control signals. With the simplified control logic, the power consumption is reduced compared to what would have been with a more complex counter-system. This reduction negates the increased power consumption from the counter to some degree compared to a more complex counter.

As stated at the start of this section, the second choice regarding the buffer scheme is related to what the buffers consist of. Some of the parts, as introduced in Chapter 3.2, can be implemented in different ways as either shared between the three buffers or individually for each buffer. An example of a part of the buffer that could be shared but is not in the proposed solution is the input register in the serializer. There are multiple such examples for both the serializer and the deserializer. As the proposed solution is not a fully optimized design, these examples could be used to improve the solution in regards to both power consumption and area utilization. Do note, however, that the DCDLs can not be shared between the buffers for either the serializer or the deserializer. The reason they can not be shared is that the signal going through the DCDLs needs to be able to propagate back when reset, as described in Chapter 2.8.

4.2.4 Other Design Choices

In chapter 2.6, two examples of how a wave-pipeline SerDes can be implemented was explained: The XOR-based solution and the WAFT solution, where the XOR-based solution is a modified version of a solution made by the author of this thesis. A choice was made in regards to which of these two solutions could provide the best baseline for the proposed solution in this thesis. In the end, the WAFT solution was chosen. The reason for this is the 15 XOR-gates in the XOR-based design, which typically has a higher internal power consumption from internal switching. Additionally, using the standard cell library the author had access to, it was estimated that the area required for the XOR-based solution is significantly higher, due to the size of the XOR-gates. Additionally, as stated in Chapter 2.6.1, The XOR-based solution is hard to modify. An example of this is switching from pilot bit signaling to a separate synchronization line, which would cause the whole data path for the XOR-based solution needing to be redesigned. These reasons combined made it clear that the WAFT solution is a better baseline for the proposed solution and is the reason that it was chosen. The WAFT solution utilizes multiplexers in the deserializer as latches. This design choice was not made for the proposed solution as multiplexers connected with a feedback line causes issues in synthesis due to the loopbreakers mentioned in section 4.1.

4.3 Simulation Results

In this section, the simulation results are presented. This includes examples of different transmission scenarios. Simulations, as stated in section 4.1, were done using the integrated

simulation tools of Xilinx Vivado. Smaller testbenches were made for some of the modules. This includes the *Serializer_wavepipe*, and *Deserializer_wavepipe* modules, in addition to the DCDL module. These testbenches were made to prove the behavioral functionality of the sub-modules, so that the debugging would be easier for the fully assembled design. A bigger testbench were made for the complete implementation. This testbench inputs random 64-bit values, at the parallel input, and is set up with different scenarios for RxReady and TxValid. The three most notable combinations are:

- The start of a transmission, where both RxReady and TxValid are low.
- The start of a transmission, where TxValid is low, but RxReady is high.
- The pausing of a transmission, where TxValid is high, but RxReady is low.

Additionally, an assertion was made that checks the input from the previous clock cycle against the current output, as these two values should match. If there is not a match, there is an error counter accumulating once each time the values do not match. The DCDLs can not be modeled with the desired delay in the behavioral simulations. The delay was simulated using the time operator #, which is not synthesizable but can be used to model time in simulations. The code snippet below shows an example of how this can be done. This code works by assigning *sel[1]* the value of *sel[0]* $1ns$ after the value of *sel[0]* changes.

```
1 assign #1ns sel[1] = sel[0];
```

In Figure 4.5, a transmission for the proposed solution can be seen. The transmission starts at the first clock cycle that the four handshaking signals, TxValid, TxReady, RxValid, RxReady, are high simultaneously. This point is marked with the yellow line in the figure. After the transmission has started, a new value is ready on the output each clock cycle, with any given value having a latency of one clock cycle. Note that while it is not possible to see the complete input and output number on the figure, due to the width of the data, the error count is visible. There would have been a value of one added to it for each situation were the input and output did not match.

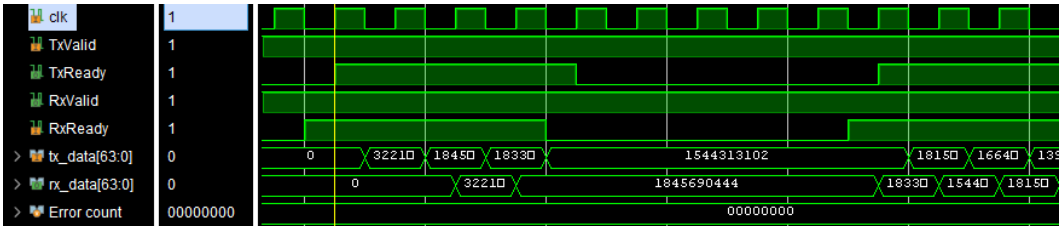


Figure 4.5: Simulation of a transmission for the proposed solution. The figure is a screen capture from Vivado.

In Figure 4.6, a scenario of backpressure is shown. In this situation, the RxReady signal is set low while TxValid is still high, and the transmission is paused. The transmission is held in a pause state in two scenarios. The first is until the RxReady signal is set high again and the transmission resumes, which is what happens in Figure 4.6. The second scenario is until the TxValid signal is set low, and the transmission is aborted, this scenario is not shown in any of the figures. Note that the RxReady signal is set low on a falling edge of the clock signal, *clk*, and the TxReady signal is kept high until the next rising edge of the clock. The

reason for this, is that the check whether the RxReady signal is still high is done at the rising edge of the clock. This means that the transmission currently happening as the RxReady signal is set low must finish, for the data not to be lost. When the RxReady signal is set high again after an undefined amount of time, the data held in the serializer is then transmitted, and outputted by the deserializer. In the case of Figure 4.6, this is the value of 1322846814. This can be seen at *RxData* after the transmission resumes.

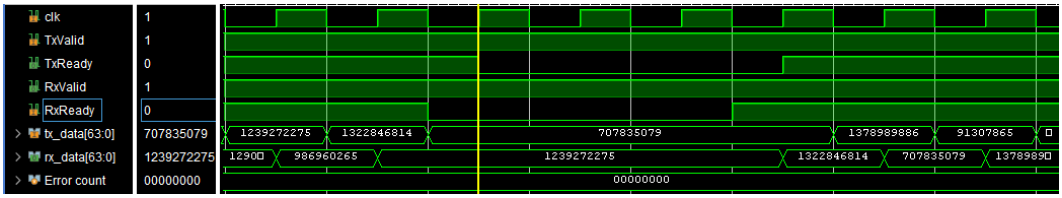


Figure 4.6: Simulation of a transmission were RxReady is set low. The figure is a screen capture from Vivado.

4.4 Results

In this section, the results for the proposed solution are presented. This includes performance, power consumption, and area, as these results are the most important in regard to the work of this thesis. The power consumption results come from the power estimation tools in Cadence Genus, and the area results are gathered by inserting the proposed solution in ARM's Mail GPU, at a 5 nm process [43]. Which is a representative example of current GPU technology.

4.4.1 Performance

In this section, the performance of the proposed solution is presented and discussed. This includes throughput and latency for the proposed solution, compared to a parallel solution.

The proposed solution makes it possible to complete transmission in a single clock cycle, as presented in section 4.3. This means that the throughput is high, with new data available on the output each clock cycle. The high throughput is made possible by the multiple buffering scheme, as each buffer has the designated cycling states of Transmit, offload, and reset, mentioned in Chapter 3.1.1. This means that every clock cycle a new buffer is ready to offload data to the output of the proposed solution.

The latency for the proposed solution is dictated by the total propagation time for transmission. As stated in Chapter 2.2.2, no transmission should last for longer than a clock cycle. For a parallel solution, both the input side and output side of the bus contain a register. This means that any single transmission can not take less time than a clock cycle, as the output register must sample the bits at the rising edge of the clock. Combined, this means that the latency for the proposed solution matches a parallel solution at one clock cycle.

The throughput and latency for the proposed solution match a parallel solution, which is one of the requirements listed in Chapter 1.5.

4.4.2 Power Consumption

In this section, the power consumption for the proposed solution is presented and discussed.

The power consumption for the proposed solution can be broken down into two parts. Power consumption from the transmitter and receiver named P_{ps}, and power consumption due to wire channel power, named P_{wire}. The equation for this is shown in Equation 4.1.

With a reduction of lines on the bus, it is expected that P_{wire} is lower for the proposed solution than a fully parallel solution. On the other hand, it is expected that P_{ps} is higher, as this part for a fully parallel solution only consists of two sets of registers connected, compared to the serialization and deserialization logic of the proposed solution. The author of this thesis did not have access to tools that could estimate P_{wire}. Because of this, this section will focus on P_{ps}.

$$P_{total} = P_{ps} + P_{wire} \tag{4.1}$$

The power consumption was estimated using the cadence genus power estimation tool, with 28 nm process. The total width of the parallel input used for the estimations are 64 bits, which gives a total of 13 serialization, and deserialization units. The total power consumption, as well as the power consumption of different subcategories, can be seen in Table 4.1. The highest power consumption comes from the registers. This is due to the combination of the number of registers, and that the registers have a higher power consumption than the smaller cells, such as the NAND-gates used in the DCDLs. The second highest source of the power consumption is the logic, which for the most part comes from the NAND-gates and the multiplexers. The total power consumption for the proposed solution is 260 μW. This includes the control logic, and logic needed for the data path.

Table 4.1: Power consumption for the proposed solution in μW

	Leakage [μW]	Internal [μW]	Switching [μW]	Total [μW]	Total [%]
Register [μW]	12.638	108.76	6.7412	128.14	49.15%
Latch [μW]	2.2216	1.7315	0.5053	4.4585	1.72%
Logic [μW]	45.386	14.496	35.336	95.219	36.52%
Clock [μW]	1.0797	6.5624	25.242	32.885	12.61%
Subtotal [μW]	61.325	131.55	67.825	260.70	100%
Total [%]	23.52%	50.46%	26.02%	100%	100%

The power consumption for a simplified parallel solution can be seen in Table 4.2. The solution consists of two sets of registers, one on the transmitter side and one on the receiver side, in addition to some combinatorial logic for the reset signal. As with the power estimation for the proposed solution, the total width of the parallel data is 64 bits. The total power consumption for the parallel solution is 66.001 μW.

Table 4.2: Power consumption for a parallel solution in μW

	Leakage [μW]	Internal [μW]	Switching [μW]	Total [μW]	Total [%]
Register [μW]	2.8781	46.494	1.6329	51.005	77.28%
Logic [μW]	0.6542	1.7312	4.3156	6.7011	10.15%
Clock [μW]	0	0	8.2944	8.2944	12.57%
Subtotal [μW]	3.5323	48.226	14.243	66.001	100%
Total [%]	5.35%	73.07%	21.58%	100%	100%

When comparing the power estimations for the proposed solution with the parallel solution, the power consumption for the proposed solution is 3.95 times higher than for the parallel solution. This can be seen in the graph of Figure 4.7. It is important to reiterate that the power consumption for the proposed solution is expected to be higher for the

proposed solution since the power consumption due to wires and buffers are not considered. Additionally, it is important to note that the main objective is to reduce the number of wires on the bus, which is a trade-off when compared to the power consumption. With that being said, it is a significant increase in power consumption, but it is important to set it in context of a big chip. The proposed solution would be a small part of a system, and the power consumption a small part of the total power consumption of this system. It is therefore not an unreasonable increase in the power consumption considering the area-power trade-off.

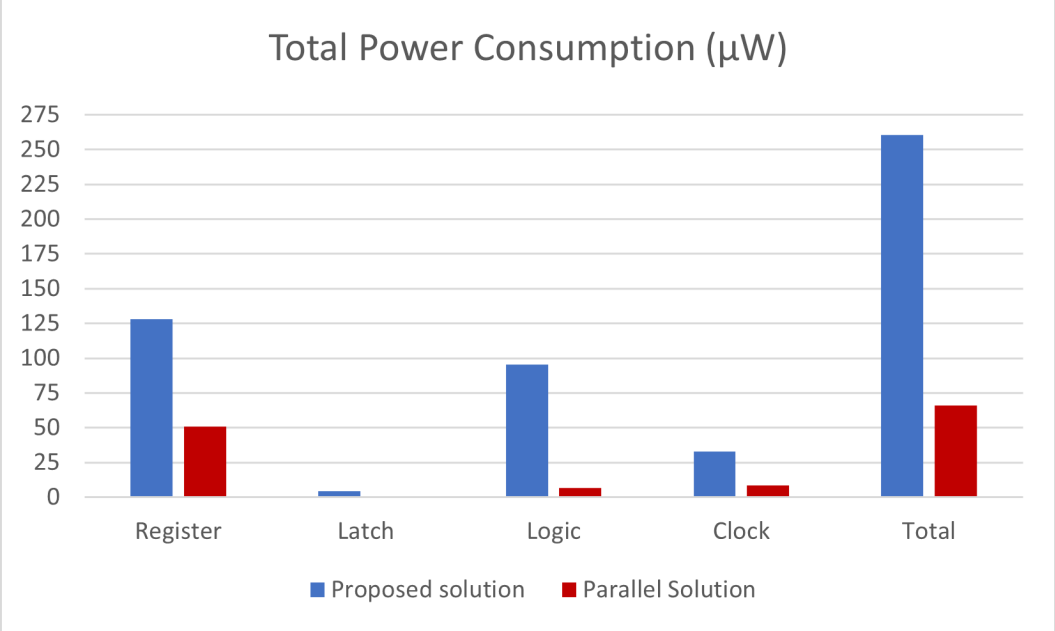


Figure 4.7: Comparison of the power consumption between a parallel solution and the proposed solution

Note that the power estimation results are not perfect. This is mainly due to estimations not including the power consumption from the bus lines, P_{wire}, which is expected to be significantly higher for the parallel solution compared to the proposed solution. This will skew the results in favor of the parallel solution, as the parallel solution will have a seemingly lower power consumption than in reality. The reason the power consumption for the bus lines is not included is that it was not possible to measure with the tools available to the author of this thesis. Additionally, the parallel solution is simplified, with less functionality, such as backpressure, and clock gating. Note that the power estimations are tied to the standard cell library that was used and that the results will differ if another library or process is used.

4.4.3 Routing

In this section, the area of the proposed solution is presented, compared to a parallel solution, and discussed.

The proposed solution was implemented using routing tools by ARM Norway in a 5 nm process, with a 64-bit input and output. This was done by inserting the proposed solution as the connection of a shader core and comparing it to a parallel solution. This is shown in Figure 4.8. The red and grey part of the figure indicates two different voltage domains, and the connection is indicated by the yellow triangles at the bottom of the gray area. Figure 4.8a shows the connection using a parallel solution, while Figure 4.8b shows the connection

using the proposed solution. The triangle in the proposed solution is noticeably narrower than for a parallel solution in the figures.

The number of lines in the proposed solution is reduced by about half compared to the parallel solution. Note that this does not include any information about the area of the solution itself but rather the changes to routing caused by the proposed solution. Additionally, the routing length was reduced by 17.94%. The routing length and reduction of lines are related. With a lower number of lines, there is both a shorter total routing length, and it can also enable the routing to be done more optimally, further shortening the routing length. Overall, the results show a decrease in routing, which is in line with what was expected with a reduction of five-to-one in lines that comes with the proposed solution.

(a) Connections using a parallel solution

(b) Connection using the proposed solution

Figure 4.8: Connections to a shader core using a parallel solution and the proposed solution. the connection is indicated using yellow rectangles at the bottom. Courtesy of ARM.

Routing for the proposed solution was done by ARM. The reason for this is that the tools for doing so were not accessible to the author. Additionally, if routing would have to be done by the author of this thesis, it would mean that he would have less time to focus on other equally important things within the somewhat short time frame of the thesis. Due to confidentiality, some details surrounding the routing and synthesis done by ARM can not be shared in this thesis. A summary, written by Morten W. Lund, of everything that was done by arm can be found in Appendix A. Here, he states that due to ARM's conservative synthesis flow, both the SR-latch and D-latches introduced several timing issues that they were only partially able to solve. This is one of many details that need to be fixed before a wave-pipeline SerDes can be production-ready. These issues made it hard for ARM to extract either power consumption data or static timing analysis. Further, he writes that the trial appears to be successful in regards to the area and that the added area caused by the added logic seems to be offset by the reduction of lines and routing length. This is in line with what can be seen in the numbers presented above.

4.5 Further Discussion

There are multiple multiplexers in the data path for the serializer which can cause an issue. The issue is related to the rise- and fall-time of signal propagating through a multiplexer.

For *D4*, as seen in Figure 3.3, this can cause a difference in propagation delay in relation to *D0*, which has fewer multiplexers to propagate through. The solution to this issue is to insert inverters on the data line between each multiplexer, which, as mentioned earlier, averages the rise-time and the fall-time for the total propagation path. It is important, however, that each signal is inverted an even number of times, so that the initial value for the input is equal to the value at the output of the proposed solution. In the example presented in Chapter 2.7 it can be seen that the propagation delay of the multiplexers is a small part compared to the propagation delay for the DCDLs. This means that the difference in rise- and fall-time might be negligible, but it is still worth keeping in mind.

It is stated in Chapter 2.2.2 that for this thesis, it was decided that any signal should only propagate on the serial line of a maximum of one clock cycle. The reason this choice was made is to both simplify the wave-pipeline SerDes and to make sure that the reliability of the proposed solution is not reduced by a high number of components that the signal must propagate through. For the second reason, the main concern for a long propagation path would be in regards to rise-time and fall-time differences in delay, which would have an amplified effect in such cases. This is also one of the reasons inverters used for averaging the rise-time and fall-time is recommended.

There is also a second talking-point regarding the multiplexers in the proposed solution. There are, in total, six multiplexers on the data line in the serializer, while in the deserializer there are zero. This means the serializer and the deserializer in the proposed solution are not mirrored, which causes the propagation delay in the two parts to be different because the multiplexers in the serializer have a propagation delay that is not in the deserializer. The solution to this issue is to add a delay equal to the propagation delay for a multiplexer to each of the DCDLs in the deserializer, as stated in Chapter 3.5. This can be better explained with *D4* as an example. At the start of the transmission, the *D4* signal propagates through five multiplexers in the serializer. If the propagation delay of a multiplexer is added in all four DCDLs between *RxNode4* and *RxNode0* in the deserializer, the total propagation delay for the serializer and the deserializer becomes equal, and the issue is solved for all the bits. Since the *D4* signal propagates through all the multiplexers in the serializer, it has the highest delay in the serializer, which means it should have the lowest propagation delay in the deserializer. On the other end, the *D0* signal only propagates through a single shared multiplexer in the serializer. However, with added delay on all four of the DCDLs in the deserializer, the total delay is equal to the total propagation delay of the *D4* signal.

The DCDL module for the proposed solution is made as modular as possible, with the possibility of deciding the number of steps using a parameter. The reason for this is that the number of steps needed for the proposed solution is dependent on the propagation delay of a NAND-gate cell in the standard cell library used. Additionally, the number of steps is also dependent on frequency of the system clock used in the design the proposed solution is placed in. This makes it so that the number of steps for correct functionality can only be determined based on the propagation delay and the system clock frequency for the design the proposed solution is used in. If a wave-pipeline SerDes is implemented as an IP solution, a DCDL step could be implemented as a library cell. This would make it easier to design the cell with small differences in propagation delay for corner cases, and might reduce process variations.

The buffering solution used in this thesis could possibly be further optimized for area than what has been done in this thesis. This could possibly be done, by move certain components in each of the buffers, so that the component is shared between the three component instead of there being three of the same components. An example of this is the input and output registers. It might be possible that the three buffers share a single set of registers, and by

that the number of components is reduced, and the total area is reduced.

Few calibration schemes could be found while researching for this thesis. This led to the author looking at calibration schemes used for other communication circuits. The autobaud scheme was the only one of these that the author deemed to have potential as inspiration for the calibration scheme presented in Chapter 3.4. Because of this, the calibration scheme, an essential part of the proposed solution, is a potential point for further work.

As mentioned in Chapter 2.9, timing calibration is not the only way of calibrating a wave-pipeline SerDes. The voltage calibration method is the second way of calibration mentioned in this thesis. While voltage variations are, according to S. Lee et al., the variable with the biggest impact on a wave-pipeline SerDes of the three variations, it was found to be too comprehensive for this thesis. With that being said, the claims made by S. Lee et al. are worth further research and should be considered as future work for this thesis.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The routing congestion on the increasingly parallel on-chip interconnects in SoCs leads to higher power consumption and area usage. In this thesis, a wave-pipeline SerDes was implemented, and it was tested if it could be the solution to the routing congestion. The proposed solution is capable of five-to-one reduction of data lines on the bus, and one transmission per clock cycle. The proposed solution is also capable of backpressure. Without considering the power consumption from the lines and buffers on the bus, the power consumption of the proposed solution showed an increase of 3.95 times compared to a parallel solution. However, it is expected that with the reduction of lines, the increase in power consumption is limited. Additionally, the proposed solution showed a reduction of lines on the bus by approximately 50%, with a reduction of 17.94% in routing length. This is a significant decrease, which can be expected to cause the area for the proposed solution to be lower in total. The proposed solution fulfills all the design targets and requirements listed in Chapter 1.5.

In the introduction of the thesis, three questions were asked. The first of these questions is: Can a wave-pipeline SerDes be implemented so that it can be reliably used across multiple technology nodes? The author of this thesis believes it has been proved that it is possible, provided considerations are made for PVT variations and propagation delay for the relevant cells in the library used for the implementation. Can the proposed solution be ported to a next-generation technology node? Or is such a solution too reliant on the characteristics of the standard cell library used for the implementation to do so? The proposed solution proves that it is possible, with small adjustments, to make a wave-pipeline SerDes solution that can be ported across technology nodes, while still maintaining the reliability, provided a sufficient calibration scheme is implemented. The solution is not reliant on any characteristics of the standard cell library used. The exception for this is the propagation delay which, for the DCDLs, the modularity of the DCDL module simplifies in the case of porting the design. However, preparation is needed if the design is to be ported to a new process, in regards to the propagation delay, which is worth keeping in mind.

The results presented in this thesis are promising. The author believes it is worth researching to make it an IP solution for widespread use. This statement is backed by ARM in the summary, which can be found in Appendix A. Here, they say that they think that the wave-pipeline SerDes is a viable option to be considered in their future GPU designs. Further, they say that they believe that the wave-pipeline SerDes solution has potential and that they will use it as a reference in their continuous effort to improve their GPU's performance, power consumption and to reduce the area.

5.2 Future Work

Several areas for future work have been identified throughout this thesis. These areas are mentioned in this chapter. The first is to implement a calibration scheme. The proposed calibration scheme in this thesis could not be implemented due to the time limitation of the thesis. Additionally, based on the claims of S. Lee et al., power supply variations have the biggest impact of the PVT variations [30]. It is worth looking further into voltage calibration schemes.

Additionally, a more extensive power analysis should be done. This includes the power consumption from the lines and buffers on the bus. While it is not the main focus of this thesis, it is still important in regards to low-power or battery-powered chips, if the proposed solution is ever going to be useful as an IP block.

It is also important that the proposed solution is further optimized for both area and power consumption as it is not a perfect solution as it is now. This could be done by finding better ways of implementation or by optimizing the buffering solution, as mentioned in Chapter 4.5.

Lastly, how to receive the clock signal in the deserializer from the serializer was identified as the most challenging to figure out. The SR-latch solution, while it works as intended now, might not be the best solution as to how this can be done. A point for the future is to further research this, to try to find an even better way of doing this than what is presented here.

Bibliography

[1] S. Borkar, "Design challenges of technology scaling," *IEEE Micro*, vol. 19, no. 4, pp. 23–29, Jul. 1999, conference Name: IEEE Micro.

[2] 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: ACM Press, 2008, p. 43. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1353610.1353620

[3] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl, "Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. Portland OR USA: ACM, Jun. 2020, pp. 1633–1649. [Online]. Available: https://dl.acm.org/doi/10.1145/3318464.3389705

[4] 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)*, Sep. 2006, pp. 9–13.

[5] B. Venu, "Multi-core processors - An overview," *University of Liverpool*, 2011, publisher: arXiv Version Number: 1. [Online]. Available: https://arxiv.org/abs/1110.3535

[6] C. Zhang, M. Li, Z. Wang, K. Yin, Q. Deng, Y. Guo, Z. Cao, and L. Liu, "Multi-channel 5Gb/s/ch SERDES with Emphasis on Integrated Novel Clocking Strategies," *JSTS:Journal of Semiconductor Technology and Science*, vol. 13, no. 4, pp. 303–317, Aug. 2013. [Online]. Available: http://www.dbpia.co.kr/Journal/ArticleDetail/NODE02241803

[7] Y. Luo, "A high speed serializer/deserializer design," *Dissertation at the university of New Hampshire*, 2011.

[8] 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*, Jun. 2014, pp. 1–2, iSSN: 2158-9682.

[9] J. Postman and P. Chiang, "A Survey Addressing On-Chip Interconnect: Energy and Reliability Considerations," *ISRN Electronics*, vol. 2012, pp. 1–9, Mar. 2012. [Online]. Available: https://www.hindawi.com/journals/isrn/2012/916259/

[10] 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.

[11] D. R. Stauffer, Ed., *High speed serdes devices and applications*. New York: Springer, 2008, oCLC: ocn226974729.

[12] M. Pedersen, "Compacting on-chip ultra-wide global interconnect buses," *For the NTNU course TFE4590 - Specialization Project*, 2021.

[13] H. Kaeslin, *Digital integrated circuit design: from VLSI architectures to CMOS fabrication*. Cambridge: Cambridge University Press, 2008, oCLC: 558886553. [Online]. Available: http://www.books24x7.com/marc.asp?bookid=29297

[14] J. P. Shen and M. H. Lipasti, *Modern processor design: fundamentals of superscalar processors*. Long Grove, Illinois: Waveland Press, 2013.

[15] C. T. Gray, W. Liu, and R. K. Cavin, *Wave pipelining: theory and CMOS implementation*, ser. The Kluwer International series in engineering and computer science. Boston: Kluwer Academic Publishers, 1994, no. 248.

[16] L. W. Cotten, "Maximum-rate pipeline systems," in *Proceedings of the May 14-16, 1969, spring joint computer conference on XX - AFIPS '69 (Spring)*. Boston, Massachusetts: ACM Press, 1969, p. 581. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1476793.1476883

[17] W. Burleson, M. Ciesielski, F. Klass, and W. Liu, "Wave-pipelining: a tutorial and research survey," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 3, pp. 464–474, Sep. 1998. [Online]. Available: http://ieeexplore.ieee.org/document/711317/

[18] D. Doman, *Engineering the CMOS library: enhancing digital design kits for competitive silicon*. Hoboken, N.J: John Wiley & Sons, 2012, oCLC: ocn755700209.

[19] D. Jansen, Ed., *The Electronic Design Automation Handbook*. Boston, MA: Springer US, 2003. [Online]. Available: http://link.springer.com/10.1007/978-0-387-73543-6

[20] J. M. Rabaey, A. P. Chandrakasan, and B. Nikolic, *Digital integrated circuits: a design perspective*, 2nd ed., ser. Prentice Hall electronics and VLSI series. Upper Saddle River, N.J: Pearson Education, 2003.

[21] D. Wiklund, "Mesochronous clocking and communication in on-chip networks," *Linkoping University*, 2003.

[22] J. Sparsø and S. B. Furber, Eds., *Principles of asynchronous circuit design: a systems perspective*, ser. European low power initiative for electronic system design. Boston: Kluwer Academic Publishers, 2001.

[23] "Keystone architecture universal asynchronous receiver/transmitter (UART) user guide," [ONLINE]. Available: https://www.ti.com/lit/ug/sprugp1/sprugp1.pdf?ts=1654675616284, Last accessed on 2022-06-08.

[24] D. Verbitsky, R. R. Dobkin, R. Ginosar, and S. Beer, "StarSync: An extendable standard-cell mesochronous synchronizer," *Integration*, vol. 47, no. 2, pp. 250–260, Mar. 2014. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167926013000497

[25] G. S. Jovanovic and M. K. Stojcev, "Current starved delay element with symmetric load," *International Journal of Electronics*, vol. 93, no. 3, pp. 167–175, Mar. 2006. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/00207210600560078

[26] R. Giordano, F. Ameli, P. Bifulco, V. Bocci, S. Cadeddu, V. Izzo, A. Lai, S. Mastroianni, and A. Aloisio, "High-Resolution Synthesizable Digitally-Controlled Delay Lines," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 3163–3171, Dec. 2015, conference Name: IEEE Transactions on Nuclear Science.

[27] D. Preethi and R. S. Valarmathi, "Efficient Implementations of Delay Elements for Digitally Controlled Delay Lines," in *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, Jul. 2019, pp. 1–7.

[28] D. De Caro, "Glitch-Free NAND-Based Digitally Controlled Delay-Lines," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 1, pp. 55–66, Jan. 2013, conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

[29] 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)*, Jul. 2012, pp. 265–268.

[30] S.-J. Lee, K. Kim, H. Kim, N. Cho, and H.-J. Yoo, "Adaptive network-on-chip with wave-front train serialization scheme," in *Digest of Technical Papers. 2005 Symposium on VLSI Circuits, 2005.*, Jun. 2005, pp. 104–107, iSSN: 2158-5636.

[31] "PVT (Process, Voltage, Temperature)," [ONLINE]. Available: https://www.physicaldesign4u.com/2020/07/pvt-process-voltage-temperature.html, Last accessed on 2022-05-25.

[32] M. A. Scarpato, "Digital circuit performance estimation under PVT and aging effects," *Université Grenoble Alpes NNT : 2017GREAT093. tel-01773745f*, p. 157, 2017.

[33] T. McConaghy, A. Gupta, J. P. Hogan, K. Breen, and J. Dyck, *Variation-aware design of custom integrated circuits: a hands-on field guide.* New York Heidelberg: Springer, 2013.

[34] P. L. Teehan, "Reliable high-throughput FPGA interconnect using source-synchronous surfing and wave pipelining," *The University Of British Columbia*, p. 120, 2008.

[35] M.-B. Lin, *Introduction to VLSI Systems: a Logic, Circuit, and System Perspective.* Hoboken: CRC Press, 2011, oCLC: 908077561. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=1763326

[36] X. Tan, X.-W. Shen, X.-C. Ye, D. Wang, D.-R. Fan, L. Zhang, W.-M. Li, Z.-M. Zhang, and Z.-M. Tang, "A Non-Stop Double Buffering Mechanism for Dataflow Architecture," *Journal of Computer Science and Technology*, vol. 33, no. 1, pp. 145–157, Jan. 2018. [Online]. Available: http://link.springer.com/10.1007/s11390-017-1747-6

[37] A. H. Balabanyan and A. A. Durgaryan, "Fully integrated PVT detection and impedance self-calibration system design," in *2016 XXV International Scientific Conference Electronics (ET).* Sozopol, Bulgaria: IEEE, Sep. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7753460/

[38] R. Electronics, "Auto Baud Rate Detection (AutoBaud)," *Application note*, pp. 3–5, 2003, [ONLINE]. Available: https://www.renesas.com/us/en/document/apn/auto-baud-rate-detection-autobaud, Last accessed on 2022-05-27.

[39] M. L. Bushnell and V. D. Agrawal, *Essentials of electronic testing for digital, memory, and mixed-signal VLSI circuits*, ser. Frontiers in electronic testing. Boston: Kluwer Academic, 2000, no. 17.

[40] K. L. Kishore and V. S. V. Prabhakar, *VLSI Design*. I.K International Publishing House, 2009, oCLC: 850766903.

[41] T. Wheeler, P. Graham, B. Nelson, and B. Hutchings, "Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification," in *Field-Programmable Logic and Applications*, G. Goos, J. Hartmanis, J. van Leeuwen, G. Brebner, and R. Woods, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, vol. 2147, pp. 483–492, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/3-540-44687-7_50

[42] "An introduction to AMBA AXI," [ONLINE]. Available: https://developer.arm.com/documentation/102202/0200/Channel-transfers-and-transactions, Last accessed on 2022-06-08.

[43] "Arm mali GPUs," [ONLINE]. Available: https://www.arm.com/product-filter, Last accessed on 2022-06-27.

Appendix A

Feedback from ARM

In this appendix, feedback on the proposed solution from ARM is presented. This feedback regards both synthesis and routing of the proposed solution. The feedback is written by Morten W. Lund M.Sc., which is also an advisor for the thesis.

A.1 Background

An implementation trial and an evaluation of the Wave-Pipeline (WP) SERDES solution presented by Morten Pedersen in his master thesis have been conducted by ARM Norway. The trial includes synthesis and physical layout of the design in conjunction with a part of an upcoming Mali GPU by ARM. A full layout of the entire GPU was not possible due to time constraints. However, the WP-SERDES was successfully implemented as the main interface for one of the GPU's "shader-core" computation blocks.

A.2 Evaluation

As expected, the WP-SERDES module(s) were not friendly to ARM's conservative synthesis flow. Both D-latches and SR-latches, found in the WP-SERDES, introduced several timing issues that we were only able to partially solve for the trial. This was not a showstopper but is one of the many small things that have to be solved before the design is "production-ready." Unfortunately, this did hamper our effort to extract power consumption data and do static timing analysis. For the area of the design, the trial appears to be successful. The cost of the added logic seems to be offset by the reduction in routing, even at a compression ratio of 4 to 1. The total routing length is, as expected, reduced according to the reduction of lines. This reduces the total line capacitance and, therefore, also power consumption. Further investigation is needed to confirm that this reduction of line capacitance is enough to compensate for the higher frequency. However, the numbers are good in our opinion.

A.3 Conclusion

Though we only managed to do a limited trial of the design, ARM still thinks the WP-SERDES is a viable option for us to consider in future GPU designs. We still have some design methodology challenges we need to overcome before we can do a large-scale test of the WP-SERDES design. Regardless, we strongly believe that this solution has potential and will use it as a reference in our continuous effort to improve our GPU's performance, power consumption, and reduce area.

Your efforts are greatly appreciated, and I wish you all the best in your future endeavor.

Appendix B

Source Code for the Serializer

Here, the source code for certain serializer modules can be found. This includes the most important modules for the serializer, such as the data module, control module, and buffering module. In the *serializer_data* module, note the "only for simulation" comments at the bottom. This is used in place of the DCDLs for simulation of the proposed solution. Here, arrows are used to indicate line breaks for lines that are too long. In section B.1, note that there are two possible *Serializer_multi_buf* modules that can be generated on line 37. This is where the serializer units are generated. The two possibilities include a unit of five bits or a unit of a custom number of bits for the last unit. The last unit is only generated if there is a remainder after all the full units have been generated.

B.1 Serializer Wavepipe

```
1  module serializer_wavepipe
2     #(
3      parameter type TXDATA_T     = logic unsigned [63:0],
4      parameter  N_DCDLSteps      = 20,
5      parameter int  N_Ser        = $size(TXDATA_T)/5,
6      // - Derived Data Types -
7      parameter int  TXGROUP_SIZE = ( $size(TXDATA_T) != $bits(TXDATA_T) ) ?
       ↪  $size(TXDATA_T) : 1,
8      parameter int  TXGROUP_MSB  = TXGROUP_SIZE - 1,
9      parameter type TXVALID_T    = logic unsigned [TXGROUP_MSB:0],
10     parameter logic  remainder  = (($size(TXDATA_T) % 5) != 0) ? 1 : 0
11     )
12          (
13    // **** Interfaces ****
14
15    // **** Outputs ****
16    output logic                   tx_ready,
17    output logic                   SerDes_Valid,
18    output logic [N_Ser:0]        ser_data,
19    output logic [N_Ser:0]        xck,
20
21    // **** Inputs ****
22    input logic[N_DCDLSteps-1:0] DCDL_sel,
23    input logic                            DesSer_Ready,
```

```
24      input logic                                 tx_valid,
25      input TXDATA_T                          tx_data,
26      // - System (Clock and Reset) -
27      input logic                                 clk,
28      input logic                                 reset_n
29      );
30
31      wire [2:0]sel_in;
32      wire [2:0]enable;
33      wire [1:0]ser_sel;
34
35          for (genvar g_i = 0; g_i <= N_Ser; g_i++)
36       begin
37       if (remainder == 1 && (g_i == (N_Ser))) begin
38                  Serializer_multi_buf
39                  #(
40                          .N_DCDLSteps                (N_DCDLSteps)
41                  )
42                  U_Ser_multi_rem
43                  (
44                          .tx_data
                            ↪  (tx_data[$size(TXDATA_T)-1:$size(TXDATA_T) -
                            ↪  $size(TXDATA_T)%5]),
45                          .clk                        (clk),
46                          .reset_n                    (reset_n),
47                          // Outputs
48                          .ser_data                   (ser_data[g_i]),
49                          .xck                        (xck[g_i]),
50                          // Inputs
51                          .enable                     (enable),
52                          .sel_in                     (sel_in),
53                          .ser_sel                    (ser_sel),
54                          .DCDL_sel                   (DCDL_sel)
55                  );
56          end
57          else if(g_i < (N_Ser)) begin
58          Serializer_multi_buf
59                  #(
60                          .N_DCDLSteps                (N_DCDLSteps)
61                  )
62                  U_Ser_multi
63                  (
64                          .tx_data                    (tx_data[5*g_i+:5]),
65                          .clk                        (clk),
66                          .reset_n                    (reset_n),
67                          // Outputs
68                          .ser_data                   (ser_data[g_i]),
69                          .xck                        (xck[g_i]),
70                          // Inputs
```

```
71                          .enable                     (enable),
72                          .sel_in                     (sel_in),
73                          .ser_sel                    (ser_sel),
74                          .DCDL_sel                   (DCDL_sel)
75                  );
76          end // end if
77      end // end for
78
79          Serializer_control
80          #(
81                  .TXDATA_T                       (TXDATA_T),
82                  .N_Ser                              (N_Ser)
83          )
84          U_Serializer_Control
85          (
86                  .tx_ready                       (tx_ready),
87      .SerDes_Valid                       (SerDes_Valid),
88      // **** Inputs ****
89      .enable                                      (enable),
90      .sel                                              (sel_in),
91      .DesSer_Ready                       (DesSer_Ready),
92      .tx_valid                               (tx_valid),
93      .ser_sel                                     (ser_sel),
94      // - System (Clock and Reset) -
95      .clk                                             (clk),
96      .reset_n                                     (reset_n)
97      );
98
99  endmodule
```

B.2 Serializer Multi Buf

```
module Serializer_multi_buf
   #(
    parameter type TXDATA_T     = logic unsigned [63:0],
    parameter  N_DCDLSteps     = 20,
    // - Derived Data Types -
    parameter int  TXGROUP_SIZE = ( $size(TXDATA_T) != $bits(TXDATA_T) ) ?
     ↪  $size(TXDATA_T) : 1,
    parameter int  TXGROUP_MSB  = TXGROUP_SIZE - 1,
    parameter type TXVALID_T    = logic unsigned [TXGROUP_MSB:0],
    parameter int  N_Ser        = $size(TXDATA_T)/5
    )
        (
   // **** Interfaces ****

   // **** Outputs ****
   output logic           ser_data,
   output logic           xck,

   // **** Inputs ****
   input logic            [2:0]enable,
   input logic            [2:0]sel_in,
   input logic            [N_DCDLSteps-1:0] DCDL_sel,
   input logic            [1:0] ser_sel,
   input logic            [4:0] tx_data,
   // - System (Clock and Reset) -
   input logic            clk,
   input logic            reset_n
   );

   logic [2:0]ser_out;
   logic [2:0]xck_i;

   // OUTPUT MUX FOR BUFFERS AND XCK
   always@(ser_sel, ser_out, xck_i)
   begin
   case (ser_sel)
      2'b00 :
      begin
        ser_data = ser_out[0];
        xck = xck_i[0];
      end
      2'b01:
      begin
        ser_data = ser_out[1];
        xck = xck_i[1];
        end
      2'b10:
```

```
47        begin
48          ser_data = ser_out[2];
49          xck = xck_i[2];
50        end
51        default:
52        begin
53          ser_data = 0;
54          xck = 0;
55        end
56      endcase
57     end
58
59     for (genvar g_i = 0; g_i < 3; g_i++)
60      begin
61      serializer_data
62      #(
63                .N_DCDLSteps                (N_DCDLSteps)
64       )
65      U_Ser_data
66      (
67                .tx_data                    (tx_data),
68                .clk                        (clk),
69                // Outputs
70                .ser_data                   (ser_out[g_i]),
71                .xck                        (xck_i[g_i]),
72                // Inputs
73                .sel_in                     (sel_in[g_i]),
74                .enable                     (enable[g_i]),
75                .DCDL_sel                   (DCDL_sel),
76                .reset_n                    (reset_n));
77      end
78
79  endmodule
```

B.3 Serializer Data

```
// ***************************************************************************
// MODULE
//   serializer - Wavepipe Serial Data Link - Tx
// ***************************************************************************

module serializer_data
  #(
    parameter type TXDATA_T     = logic unsigned [63:0],
    parameter  N_DCDLSteps     = 20,
    // - Derived Data Types -
    parameter int  TXGROUP_SIZE = ( $size(TXDATA_T) != $bits(TXDATA_T) ) ?
     ↪  $size(TXDATA_T) : 1,
    parameter int  TXGROUP_MSB  = TXGROUP_SIZE - 1,
    parameter type TXVALID_T    = logic unsigned [TXGROUP_MSB:0]
    )
   (
    // **** Outputs ****
    output logic           ser_data,
    output logic           xck,

    // **** Inputs ****
    input logic enable,

    input logic  [N_DCDLSteps-1:0] DCDL_sel,
    input logic   [4:0]  tx_data,
    input logic          sel_in,
    // - System (Clock and Reset) -
    input logic            clk,
    input logic            reset_n
    );

  reg [4:0]Q;
  reg [4:0]TxNode;
  reg [4:0]sel;

  assign xck = sel[4];

always_ff @(posedge clk)
begin
        if (reset_n == 1'b0) begin
                Q = 5'b00000;
        end
        else begin
                if (enable == 1'b1)
                        Q = tx_data;
                else
                        Q = Q;
```

```
47          end
48  end
49
50  // Muxes
51  assign ser_data = (sel_in)?TxNode[0]:1'b0;
52  assign TxNode[0] = (sel[0])?TxNode[1]:Q[0];
53  assign TxNode[1] = (sel[1])?TxNode[2]:Q[1];
54  assign TxNode[2] = (sel[2])?TxNode[3]:Q[2];
55  assign TxNode[3] = (sel[3])?TxNode[4]:Q[3];
56  assign TxNode[4] = (sel[4])?1'b0:Q[4];
57
58  // Comment for simulation.
59  DCDL #(.Nsteps(N_DCDLSteps)) DCDL0(.data_in(sel_in), .select(DCDL_sel),
    ↪  .data_out(sel[0]));
60
61  for (genvar g_i = 0; g_i < 4; g_i++)
62  begin
63  DCDL #(.Nsteps(N_DCDLSteps)) DCDL1(.data_in(sel[g_i]), .select(DCDL_sel),
    ↪  .data_out(sel[g_i + 1]));
64  end
65
66  // ** ONLY FOR SIMULATION **
67  // Uncomment for simulation
68  // assign #1ns sel[0] = sel_in;
69  // assign #1ns sel[1] = sel[0];
70  // assign #1ns sel[2] = sel[1];
71  // assign #1ns sel[3] = sel[2];
72  // assign #1ns sel[4] = sel[3];
73
74  endmodule
```

B.4 Serializer Control

```
1  module Serializer_control#(
2      parameter type TXDATA_T     = logic unsigned [63:0],
3      parameter int  N_Ser        = (($size(TXDATA_T) % 5) != 0) ?
       ↪  $size(TXDATA_T)/5 : $size(TXDATA_T)/5 - 1,
4      parameter  N_DCDLSteps      = 20,
5      // - Derived Data Types -
6      parameter int  TXGROUP_SIZE = ( $size(TXDATA_T) != $bits(TXDATA_T) ) ?
       ↪  $size(TXDATA_T) : 1,
7      parameter int  TXGROUP_MSB  = TXGROUP_SIZE - 1
8      )
9      (
10
11     // **** Outputs ****
12     output logic tx_ready, // Ready to Transmitter <=> SerTxReady
13     output logic SerDes_Valid, // Valid to Deserializer <=> SerDesValid
14     output logic [2:0]enable, // Enable for input registers in serializer
15     output logic [2:0]sel, // select signal for the muxes in the serializer
       ↪  (the signal that is delayed) and also the clock signal sent to the
       ↪  deserializer.
16     // output logic  [N_DCDLSteps-1:0] DCDL_sel, // Select how many steps
       ↪  that should be used in the DCDL, effectively adjusting the delay.
17
18     // **** Inputs ****
19     input        logic tx_valid, // Valid from Transmitter <=> TxSerValid
20     input        logic     DesSer_Ready, // Ready from Deserializer <=>
       ↪  DesSerReady
21     output logic [1:0] ser_sel, // Select which serializer in the multibuffer
       ↪  that is connected to the serial line.
22     // - System (Clock and Reset) -
23     input logic            clk,
24     input logic            reset_n
25      );
26
27   // **** Local Variables  ****
28   parameter S_IDLE = 2'b00, S_WAIT = 2'b01, S_TRANSFER = 2'b11;
29   logic [1:0]   curr_state;
30   logic [1:0]   last_state;
31
32   logic unsigned [1:0] count;
33   logic unsigned [1:0] last_count;
34   logic unsigned [2:0] enable_select;
35   logic count_stop;
36   logic count_stop1;
37
38     // *** COUNTER ***
39     always @(posedge clk)
40     begin
```

```
41      count_stop = count_stop1;
42      end
43
44      always @(posedge clk)
45      begin
46      if (count_stop == 1'b0) begin
47                  last_count = count;
48                  count = count;
49      end
50      else begin
51                  if(count == 2'b00)begin
52                          last_count = count;
53                              count = 2'b01;
54                          enable_select = 3'b010;
55                  end
56                  else if (count == 2'b01)begin
57                          last_count = count;
58                              count = 2'b10;
59                          enable_select = 3'b100;
60                  end
61                  else begin
62                          last_count = count;
63                              count = 2'b00;
64                          enable_select = 3'b001;
65                  end
66          end
67          end
68
69      assign ser_sel = count;
70
71      always @ (posedge clk)
72      begin : Control
73          tx_ready = DesSer_Ready;
74      end
75
76      always @(posedge clk)
77          begin : FSM
78          if (reset_n == 0) begin
79                  count_stop1 = 1'b0;
80          SerDes_Valid = 1'b0;
81          sel = 3'b0;
82          end
83          else
84      begin
85          case (curr_state)
86              S_IDLE:
87                  begin
88                      sel = 3'b0;
89                      if (last_state != S_TRANSFER)
```

```
90                      count_stop1 = 1'b0;
91                  if (tx_valid == 1'b1 && DesSer_Ready == 1'b0)
92                  begin
93                      SerDes_Valid = 1'b1;
94                      count_stop1 = 1'b1;
95                      last_state = curr_state;
96                      curr_state = S_WAIT;
97                  end
98                  else if (tx_valid == 1'b1 && DesSer_Ready == 1'b1)
99                  begin
100                     SerDes_Valid = 1'b1;
101                     count_stop1 = 1'b0;
102                     sel[count] = 1'b1;
103                     last_state = curr_state;
104                     curr_state = S_TRANSFER;
105                 end
106                 else
107                     curr_state = S_IDLE;
108             end
109         S_WAIT:
110             begin
111                 sel[last_count] = 1'b0;
112                 sel[count] = 1'b0;
113                 if (DesSer_Ready == 1'b1)
114                 begin
115                     SerDes_Valid = 1'b1;
116                     count_stop1 = 1'b0;
117                     sel[count] = 1'b1;
118                     last_state = curr_state;
119                     curr_state = S_TRANSFER;
120                 end
121                 else
122                     curr_state = S_WAIT;
123             end
124         S_TRANSFER:
125             begin
126                 sel[count] = 1'b1;
127                 sel[last_count] = 1'b0;
128                 if (DesSer_Ready == 1'b0 && tx_valid == 1'b1)
129                 begin
130                     sel[count] = 1'b0;
131                     count_stop1 = 1'b1;
132                     last_state = curr_state;
133                     curr_state = S_WAIT;
134                 end
135                 else if (tx_valid == 1'b0)
136                 begin
137                     count_stop1 = 1'b1;
138                     SerDes_Valid = 1'b0;
```

```
139                         last_state = curr_state;
140                         curr_state = S_IDLE;
141                     end
142                     else
143                         curr_state = S_TRANSFER;
144                 end
145         endcase
146                 end // end reset
147     end //End FSM
148 
149 always @(posedge clk)
150 begin
151         if (reset_n == 0) begin
152                 enable = 3'b000;
153         end
154         else begin
155     case (curr_state)
156     S_IDLE:
157     begin
158            if (tx_valid == 1'b1 && DesSer_Ready == 1'b1) begin
159                 enable = 3'b111;
160            end
161            else begin
162                 enable = enable;
163            end
164     end
165     S_WAIT:
166     begin
167         if (last_state == S_IDLE && DesSer_Ready == 1'b0) begin
168                       enable = enable_select;
169         end
170         else if (last_state == S_TRANSFER && DesSer_Ready == 1'b0) begin
171                 enable = 3'b000;
172         end
173         else begin
174                 enable = 3'b111;
175         end
176     end
177     S_TRANSFER:
178     begin
179         if (DesSer_Ready == 1'b0 && tx_valid == 1'b1) begin
180                 enable = 3'b000;
181         end
182         else if (tx_valid == 1'b0) begin
183                    enable = enable_select;
184 
185         end
186         else begin
187                    enable = 3'b111;
```

```
188
189           end
190         end
191       endcase
192           end // End reset
193   end
194
195   endmodule
```

Appendix C

Source Code for the Deserializer

Here, the source code for certain deserializer modules can be found. This includes the most important modules for the deserializer, such as the data module, control module, and buffering module. As for the serializer data module, there is a "for simulation only" comment, which replaces the DCDLs in simulations of the complete design.

C.1 Deserializer Wavepipe

```
module deserializer_wavepipe
   #(
    parameter type RXDATA_T     = logic unsigned [63:0],
    parameter  N_DCDLSteps      = 20,
    parameter int  N_Ser        = $size(RXDATA_T)/5,
    // - Derived Data Types -
    parameter int  RXGROUP_SIZE = ( $size(RXDATA_T) != $bits(RXDATA_T) ) ?
     ↪  $size(RXDATA_T) : 1,
    parameter int  RXGROUP_MSB  = RXGROUP_SIZE - 1,
    parameter type RXVALID_T    = logic unsigned [RXGROUP_MSB:0],
    parameter logic  remainder  = (($size(RXDATA_T) % 5) != 0) ? 1 : 0
    )
        (
   // **** Interfaces ****

   // **** Outputs ****
   output logic           rx_valid,
   output logic           DesSer_Ready,
   output RXDATA_T             rx_data,

   // **** Inputs ****
   input logic[N_DCDLSteps-1:0] DCDL_sel,
   input logic                                SerDes_Valid,
   input logic                                rx_ready,
   input logic[N_Ser:0]                ser_data,
   input logic[N_Ser:0]                xck,

   // - System (Clock and Reset) -
   input logic                                clk,
```

```
29      input logic                                     reset_n
30      );
31
32      logic [2:0] ser_sel;
33
34       for (genvar g_i = 0; g_i <= N_Ser; g_i++)
35       begin
36       if (remainder == 1 && (g_i == (N_Ser))) begin
37       Deserializer_multi_buf
38       #(
39                   .N_DCDLSteps                                    (N_DCDLSteps)
40        )
41       U_des_multi_rem
42       (
43                   .rx_data                                            (rx_data[$size(RXDATA_T)-1
                     ↪  : ($size(RXDATA_T) - $size(RXDATA_T)%5)]),
44                   .clk                                                (clk),
45                   // **** Outputs ****
46                   .ser_data                                           (ser_data[N_Ser]),
47                   // **** Inputs ****
48                   .xck                                                (xck[g_i]),
49                   .ser_sel                                           (ser_sel),
50                   .DCDL_sel                                           (DCDL_sel)
51       );
52           end
53       else if(g_i < (N_Ser)) begin
54               Deserializer_multi_buf
55       #(
56           .N_DCDLSteps                                    (N_DCDLSteps)
57        )
58       U_des_multi
59       (
60                   .rx_data                                            (rx_data[5*g_i+:5]),
61                   .clk                                                (clk),
62                   // **** Outputs ****
63                   .ser_data                                           (ser_data[g_i]),
64                   // **** Inputs ****
65                   .xck                                                (xck[g_i]),
66                   .ser_sel                                           (ser_sel),
67                   .DCDL_sel                                           (DCDL_sel)
68           );
69           end // end if
70       end // end for
71
72      Deserializer_control
73      #(
74                   .RXDATA_T                                           (RXDATA_T),
75                   .N_Ser                                                   (N_Ser)
76      )
```

```
77      U_Deserializer_Control
78      (
79                      .rx_ready                                       (rx_ready),
80                      .SerDes_Valid                                (SerDes_Valid),
81                      // **** Inputs ****
82                      .DesSer_Ready                                (DesSer_Ready),
83                      .rx_valid                                       (rx_valid),
84                      .ser_sel                                       (ser_sel),
85                      // - System (Clock and Reset) -
86                      .clk                                               (clk),
87                      .reset_n                                       (reset_n)
88      );
89
90  endmodule
```

C.2 Deserializer Multi Buf

```
module Deserializer_multi_buf
   #(
    parameter type RXDATA_T     = logic unsigned [63:0],
    parameter  N_DCDLSteps             = 20,
    // - Derived Data Types -
    parameter int  TXGROUP_SIZE = ( $size(RXDATA_T) != $bits(RXDATA_T) ) ?
     ↪  $size(RXDATA_T) : 1,
    parameter int  TXGROUP_MSB  = TXGROUP_SIZE - 1,
    parameter type TXVALID_T    = logic unsigned [TXGROUP_MSB:0],
    parameter int  N_Ser        = $size(RXDATA_T)/5
    )
        (
        // **** Interfaces ****

   // **** Outputs ****
   input logic                                ser_data,
   output logic [4:0]                rx_data,
   // **** Inputs ****
   input logic                                xck,
   input logic[N_DCDLSteps-1:0] DCDL_sel,
   input logic[2:0]                        ser_sel,
   // - System (Clock and Reset) -
   input logic                                clk
   );

logic [0:2]ser_in;
logic [0:2]xck_in;
logic [14:0] rx_data_i;


// OUTPUT MUX FOR BUFFERS

assign ser_in[0] = ser_data && ser_sel[0];
assign ser_in[1] = ser_data && ser_sel[1];
assign ser_in[2] = ser_data && ser_sel[2];

assign xck_in[0] = xck && ser_sel[0];
assign xck_in[1] = xck && ser_sel[1];
assign xck_in[2] = xck && ser_sel[2];

        always@(ser_sel, rx_data_i)
        begin
        case (ser_sel)
                3'b001 :
                begin
                        rx_data = rx_data_i[14:10];
                end
```

```
47                  3'b010:
48                  begin
49                          rx_data = rx_data_i[4:0];
50                  end
51                  3'b100:
52                  begin
53                          rx_data = rx_data_i[9:5];
54                  end
55                  default:
56                  begin
57                          rx_data = rx_data_i[4:0];
58                  end
59          endcase
60          end
61
62          for (genvar g_i = 0; g_i < 3; g_i++)
63          begin
64          Deserializer_data
65          #(
66                          .N_DCDLSteps                (N_DCDLSteps)
67           )
68          U_Des_data
69          (
70                          .rx_data                    (rx_data_i[5*g_i+:5]),
71                          .clk                        (clk),
72                          // Outputs
73                          .ser_data                   (ser_in[g_i]),
74                          // Inputs
75                          .xck_in                     (xck_in[g_i]),
76                          .DCDL_sel                   (DCDL_sel));
77          end
78
79  endmodule
```

C.3 Deserializer Data

```
//
↪  ********************************************************************************
// MODULE
//   serializer - Generic Serial Data Link - Tx
//
↪  ********************************************************************************

module Deserializer_data
  #(
    parameter type RXDATA_T     = logic unsigned [63:0],
    parameter  N_DCDLSteps      = 20,
    // - Derived Data Types -
    parameter int  TXGROUP_SIZE = ( $size(RXDATA_T) != $bits(RXDATA_T) ) ?
    ↪  $size(RXDATA_T) : 1,
    parameter int  TXGROUP_MSB  = TXGROUP_SIZE - 1,
    parameter type TXVALID_T    = logic unsigned [TXGROUP_MSB:0]
    )
   (
    // **** Interfaces ****

    // **** Outputs ****
    output logic   [4:0]  rx_data,

    // **** Inputs ****
    input logic           xck_in,
    input logic           ser_data,
    input logic  [N_DCDLSteps-1:0] DCDL_sel,

    // - System (Clock and Reset) -
    input logic            clk,
    input logic             reset_n
    );

  reg enable;
  logic sel;
  logic sel_reset;
  int count;

  logic [4:0]Q;
  logic [4:0]RxNode;
  logic [4:0]RxNodeLatch;

  assign rx_data = Q;

  always_ff@(posedge clk)
  begin
    if (reset_n == 0)
```

```
45      begin
46          count = 0;
47          sel_reset = 1;
48      end
49      else begin
50      if (sel == 1)
51          count++;
52      else
53          count = 0;
54      if (count >= 2)
55          sel_reset = 1;
56      else
57          sel_reset = 0;
58    end
59    end
60
61   always_comb
62   begin
63      if (sel == 1)
64          enable = 1;
65      else
66          enable = 0;
67      end
68
69   always@(posedge clk)
70      if (sel == 1'b1) begin
71          Q = RxNodeLatch;
72      end
73      else begin
74          Q = Q;
75      end
76
77  for (genvar g_j = 0; g_j < 5; g_j++)
78  begin
79  custom_latch U_latch(.q(RxNodeLatch[g_j]), .d(RxNode[g_j]), .select(sel));
80  end
81
82  SR_latch U_latch(.S(xck_in), .R(sel_reset), .Q(sel));
83
84  // Comment for simulation
85  DCDL U_DCDL0 (.data_in(ser_data), .select(DCDL_sel), .data_out(RxNode[4]));
86
87  for (genvar g_i = 1; g_i < 5; g_i++)
88  begin
89  DCDL #(.Nsteps(N_DCDLSteps)) U_DCDL4(.data_in(RxNode[g_i]),
    ↪  .select(DCDL_sel), .data_out(RxNode[(g_i-1)]));
90  end
91  // End comment for simulation
92
```

```
93  // FOR SIMULATION ONLY
94  // ----------------------
95  //    assign #1ns RxNode[4] = ser_data;
96  //    assign #1ns RxNode[3] = RxNode[4];
97  //    assign #1ns RxNode[2] = RxNode[3];
98  //    assign #1ns RxNode[1] = RxNode[2];
99  //    assign #1ns RxNode[0] = RxNode[1];
100 //// ----------------------
101
102 endmodule
```

C.4 Deserializer Control

```
1  module Deserializer_control#(
2      parameter type RXDATA_T     = logic unsigned [63:0],
3      parameter int  N_Ser        = (($size(RXDATA_T) % 5) != 0) ?
       ↪  $size(RXDATA_T)/5 : $size(RXDATA_T)/5 - 1,
4      parameter  N_DCDLSteps      = 20,
5      // - Derived Data Types -
6      parameter int  RXGROUP_SIZE = ( $size(RXDATA_T) != $bits(RXDATA_T) ) ?
       ↪  $size(RXDATA_T) : 1,
7      parameter int  RXGROUP_MSB  = RXGROUP_SIZE - 1,
8      parameter type RXVALID_T    = logic unsigned [RXGROUP_MSB:0]
9      )
10     (
11
12         // **** Outputs ****
13         output logic                          rx_valid, // Valid to
           ↪  Transmitter <=> TxSerValid
14         // output logic [N_DCDLSteps-1:0] DCDL_sel, // Select how many steps
           ↪  that should be used in the DCDL, effectively adjusting the
           ↪  delay.
15         output logic[2:0]                ser_sel, // Select which serializer
           ↪  in the multibuffer that is connected to the serial line.
16         output logic                          DesSer_Ready, // Ready to
           ↪  Serializer <=> DesSerReady
17
18         // **** Inputs ****
19         input logic                                 SerDes_Valid, // Valid
           ↪  from Serializer <=> SerDesValid
20         input logic                           rx_ready, // Ready to
           ↪  Transmitter <=> SerTxReady
21
22         // - System (Clock and Reset)
           ↪  -
23         input logic                                 clk,
24         input logic                                 reset_n
25     );
26
27         int count;
28         logic count_stop;
29
30     assign count_stop = ~(rx_ready & SerDes_Valid);
31     assign DesSer_Ready = rx_ready;
32
33         always@(count)
34     begin
35         if (count == 0) begin
36             ser_sel = 3'b001;
37         end
```

```
38          else if (count == 1) begin
39              ser_sel = 3'b010;
40          end
41          else begin
42              ser_sel = 3'b100;
43          end
44      end
45
46      always @(posedge clk)
47      begin
48      if(count_stop == 1'b0) begin
49          if (count >= 2)
50              count = 0;
51          else
52              count++;
53      end else
54          count = count;
55      end
56
57          always @(posedge clk)
58      begin : Control
59          if (reset_n == 1'b0) begin
60                  rx_valid = 1'b0;
61          end
62          else begin
63                  rx_valid = SerDes_Valid;
64          end
65      end
66
67  endmodule
```

C.5 SR-Latch

```
module SR_latch(
    input S,
    input R,
    output reg Q
    );

wire S_i;

assign S_i = S | Q;
assign Q = (~R & S_i);

endmodule
```

Appendix D

Source Code for the Digitally Controlled Delay Lines

Here, the source code for the DCDL module can be found. All the intermediate steps are generated based on the *Nsteps* parameter. Note that the *custom_nand* reference, is a custom block containing the specific NAND-cell used in the design. This module can be seen in appendix D.2. Here, $NAND2\_CELL\_NAME$ is a temporary name, were the name of the NAND-cell, from a specific standard cell library is inserted. Additionally, the *custom_nand* module has an ifndef that that if is chosen in simulation, and else is chosen for synthesis. This can be used to simulate the delay in-place of the delay inserted in the *serializer_data* and *deserializer_data* modules.

D.1 DCDL Module

```
1  // Comments: Naming for the steps: step1_1 = step[0][0]
2  // step2_2 = step[1][1]
3
4  module DCDL
5          #(
6          parameter Nsteps = 7
7      )
8      (
9      input logic data_in,
10     input logic [Nsteps-1:0]select,
11     output logic data_out
12     );
13
14     logic [Nsteps-1:0][1:0]step;
15     logic [Nsteps-1:0] data_outi;
16     logic [Nsteps-1:0] LB;
17
18         assign step[0][0] = data_in;
19     // First step
20         custom_nand step0(.y(step[0][1]), .a(select[0]), .b(step[0][0]));
21         custom_nand lb0(.y(LB[0]), .a(data_outi[0]), .b(step[0][1]));
22
```

```
23      // Ending inverter
24      assign data_outi[6] = !step[6][0];
25
26      // Output
27      custom_nand nand_out(.y(data_out), .a(data_outi[0]), .b(step[0][1]));
28
29          for (genvar g_i = 1; g_i < Nsteps; g_i++)
30      begin
31          custom_nand step1_0(.y(step[g_i][0]), .a(!select[g_i-1]),
            ↪  .b(step[g_i-1][0]));
32          custom_nand step1_1(.y(step[g_i][1]), .a(!select[g_i]),
            ↪  .b(step[g_i][0]));
33          custom_nand lb1(.y(LB[g_i]), .a(data_outi[g_i]), .b(step[g_i][1]));
34          custom_nand nand_out1(.y(data_outi[g_i-1]), .a(data_outi[g_i]),
            ↪  .b(step[g_i][1]));
35          end
36  endmodule
```

D.2 Custom NAND Module

```
module custom_nand
  (
   // Outputs
   output logic y,
   // Inputs
   input wire   a,
   input wire   b
   );

  // NAND2
`ifndef SYNTHESIS
`define DLY #1ns
  always_comb y <= `DLY !(a & b);
`else // Insert NAND2 cell
  NAND2_CELL_NAME nand2( .Z(y), .A(a), .B(b) );
`endif

endmodule
```

Morten Pedersen

On-Chip Ultra-Wide Global Interconnect Buses using Wave-Pipeline SERDES

NTNU
Kunnskap for en bedre verden