Table of contents

List of Figures

List of Tables

1. Introduction

An integrated circuit or device used in high-speed communications that transforms between serial data and parallel interfaces in either direction is called a SerDes (Serializer/Deserializer). Additionally, a SerDes is used in many different applications and technologies with the main goal of reducing the number of input/output pins and connections while still enabling data transfer over a single line or differential.

Functionality-wise, a SerDes chip reduces the number of data routes needed for data transfer by enabling transmission between two places using parallel data via serial streams. Additionally, by doing this, fewer connection pins are required, keeping the wires and connectors thin and compact. Moreover, the receiver side does the exact opposite task from the transmitter side, which handles the conversion of parallel data to serial data.

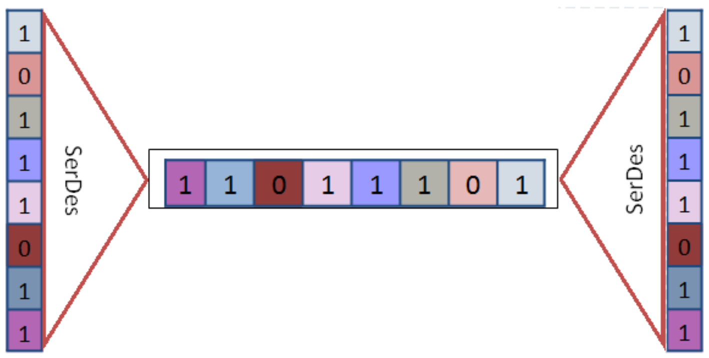


Fig 1.1 SerDES Working

In order to transfer over medium that doesn't typically handle parallel data, it transforms parallel data into serial data. Additionally, a SerDes can be used in situations where bandwidth preservation is required.

Serializer/Deserializer (SerDes) is already emerging as the leading solution in chips where there is a need for high-speed data movement and a limitation in the available I/O. However, like virtually all things, there are side-effects. In the case of SerDes, these side-effects take the form of extreme challenges in terms of design. Furthermore, these challenges are not going away or getting any easier, especially with the steady increase in demand for higher speeds in conjunction with the enormous increase in data requirements.

Also, in regards to the benefits, a SerDes affords the conversion of parallel data into serial data, which allows designers to increase the speed of data communication without the need to increase the pin count. However, with the increasing volumes of data, the number of devices accessing the internet, and the onset of growing cloud access, the design parameters for a SerDes are also increasing in complexity.

Nevertheless, SerDes is the key to helping designers and engineers meet this ever-increasing demand for speed and volumes of data. To summarize the totality of what a SerDes represents, it is the perfect convergence of analog precision and analog circuitry.

SerDes verification is a critical phase in integrated circuit design, ensuring the reliable transmission of serialized data over high-speed communication links. This verification process involves several key steps, including functional verification to confirm proper operation, timing verification to assess adherence to timing constraints, jitter analysis to mitigate timing deviations, eye diagram analysis for signal quality assessment, BER testing to evaluate error rates, power integrity analysis to manage power consumption and noise, compliance testing to ensure adherence to communication standards, interoperability testing to verify compatibility with other devices, simulation and emulation for comprehensive testing, and hardware validation for real-world performance assessment. Through a combination of rigorous analysis, testing, and validation techniques, SerDes verification aims to guarantee the robustness and reliability of communication interfaces in modern IC designs.

SerDes physical design is the meticulous process of laying out the components of Serializer/Deserializer circuits on a computer chip. It involves carefully arranging transistors, capacitors, and other elements to optimize performance and minimize interference. Engineers pay close attention to factors like signal integrity, power distribution, and routing paths to ensure that data can be transmitted quickly and reliably between devices. This process is essential because high-speed communication links, such as those used in networking equipment or high-performance computing systems, demand precise and efficient circuitry to maintain data integrity and minimize latency. Without thorough physical design, SerDes circuits may suffer from signal degradation, timing issues, or other problems that can impact the overall performance of electronic devices.

The need for SerDes physical design arises from the growing demand for high-speed data transmission in modern electronic devices. As technology advances, devices like smartphones, servers, and networking equipment require increasingly faster communication interfaces to handle large amounts of data efficiently. Serializer/Deserializer circuits play a crucial role in enabling this high-speed communication by converting parallel data streams into serial data streams and vice versa. However, achieving reliable data transmission at such high speeds requires careful physical design to address challenges like signal degradation, crosstalk, and power distribution. By optimizing the layout and routing of SerDes components, engineers can ensure that data can be transmitted quickly and accurately, enabling seamless communication between devices and enhancing overall system performance.

2. Literature Review

[1] A 100 Gb/s Quad-Lane SerDes Receiver with a PI-Based Quarter-Rate All-Digital CDR

The paper introduces a 100 Gb/s quad-lane Serializer/Deserializer (SerDes) receiver featuring a phase-interpolator (PI)-based quarter-rate all-digital Clock and Data Recovery (CDR) system. The CDR design leverages a multi-phase multiplying delay-locked loop (MDLL) to generate eight-phase reference clocks, achieving multi-phase frequency multiplication with minimal area and power consumption. A shared MDLL is employed to generate and distribute eight-phase clocks to each CDR. The proposed CDR incorporates a novel initial phase tracker utilizing a preamble for rapid lock time of approximately 12 ns and ensuring a constant output data sequence. Utilizing quarter-rate 2x-oversampling architecture, the CDR employs a custom-designed PI controller to minimize loop latency. Additionally, the decimation factor of the CDR can be adjusted to enhance the dithering jitter performance of the recovered clock.
Moreover, a new continuous-time linear equalizer (CTLE) receiver is adopted to reduce power consumption while achieving a data rate of 25 Gb/s per lane. Implemented in 40 nm CMOS technology, the 100 Gb/s four-channel SerDes receiver, comprising 4 CTLEs, 4 CDRs, and MDLL, occupies a compact active area of only 0.351 mm^2 and consumes 241.8 mW, demonstrating high energy efficiency of 2.418 pJ/bit.

[2] Implementation of 10bit SerDes for Gigabit Ethernet PHY

The paper explores the optimization of high-speed and low-power Serializer/Deserializer (SerDes) design for wideband communication, particularly in Ethernet applications. It emphasizes the significance of SerDes in facilitating high-speed communication in electronic devices. The SerDes functionality comprises two key blocks: Parallel In Serial Out (PISO) and Serial In Parallel Out (SIPO). By focusing on efficiency, the study aims to achieve high-speed data transfer rates while minimizing power consumption. The optimized SerDes design proposed in the paper demonstrates a power consumption of 737 mW and a data transfer rate of 25 Gb/s, showcasing its potential for enhancing performance in wideband communication systems.

[3] A 10 Gbps SerDes For Wireless Chip-to-chip Communication

The paper introduces a 10 Gbps serializer/deserializer (SerDes) featuring a phase interpolator (PI) based clock and data recovery (CDR) circuit tailored for high-speed and short-range wireless chip-to-chip communication. Operating with 4:1 muxing and 1:4 demuxing functionalities, the SerDes incorporates an 8-phase delay-locked loop (DLL) within the PI-based CDR to generate evenly spaced reference clock phases. Utilizing the phase vernier, the system transforms these 8-phases into sampling clocks for the sampler, which executes 2× oversampling to recover data from the input signal.
Fabricated using a 65 nm CMOS process, the proposed SerDes attains a data rate of 10 Gbps with a recovered peak-to-peak clock jitter of 36.25 ps. In terms of physical specifications, the 10 Gbps SerDes occupies an active area of 0.095mm^2 while consuming 88 mW of power.

[4] Use of SerDes to reduce the cost of packaging of VLSIs

The research paper delves into addressing the rising expenses associated with VLSI packaging, primarily driven by the increasing number of input and output (I/O) pins alongside the growing intricacy of designs. To combat these challenges cost-effectively, the study advocates for the utilization of Serializer/Deserializer (SerDes) technology. The primary objective is to curtail the number of I/O pins within internal chips, particularly those that do not necessitate external signal connectivity, by integrating open-source SerDes solutions.

The proposed approach outlines an integrated methodology to seamlessly integrate SerDes functionalities into VLSI design processes. This integration strategy aims to optimize chip layouts, reduce I/O pin counts, and ultimately mitigate the overall packaging costs associated with VLSI implementations.

[5] An 8b/10b Encoding Serializer/Deserializer (SerDes) Circuit for High Speed Communication Applications Using a DC Balanced, Partitioned-Block, 8b/10b Transmission Code

This paper presents an 8b/10b encoding Serializer/Deserializer (SerDes) circuit utilizing a DC-balanced, partitioned block, 8b/10b transmission code. The transmission code format consists of variable-length packets suitable for high-speed applications. The design includes separate blocks for serializer and deserializer functionalities.
The serializer circuit receives 8-bit data in parallel mode and outputs 10-bit coded-serialized data to the deserializer. Conversely, the deserializer decodes the received information and outputs the original 8-bit parallel data. While the serializer circuit is designed using Cadence in 0.6 μm CMOS technology, all other blocks are designed in Verilog and VerilogXL in XILINX. Finally, all individual blocks are integrated to form a cohesive system.

[6] High Speed SerDes Design Verification

This paper presents the crucial importance of accurately predicting Serial Data (SerDes) channel insertion loss and return loss for determining SerDes link performance, considering its widespread utilization across various applications such as gigabit rate links, storage, telecommunications, and data communications. To showcase a systematic modeling technique, the paper introduces two test vehicles: one featuring a bare package sample and the other incorporating a package mounted on a printed circuit board (PCB). The design of SerDes signals on both test vehicles is outlined, and their characterization through model-to-measurement correlation is discussed in Section I. Section II details a step-by-step modeling procedure, validated up to 19 GHz, to establish a correlation between the SerDes model and measurements on the bare package sample. Section III provides intricate insights into the measurement technique to ensure the accuracy of the collected data. Moreover, in Section IV, the paper presents SerDes design references for operation at 5 GHz and 12.5 GHz, based on the differential insertion loss and near-end coupling performance of the bare package and the package on the PCB as studied. The paper concludes with final remarks encapsulating the significance of the findings and potential future research directions.

[7] Research on High-speed SerDes Interface Testing Technology

The abstract outlines a study focused on testing high-speed serial interface SerDes chips, crucial for meeting the escalating demand for rapid data transmission in contemporary technology. With increasing clock frequencies, serial interfaces are preferred over parallel ones due to their reduced wire count and interference.

The study presents a method for testing high-speed SerDes serial interfaces utilizing the V93000 test system. An FPGA product, specifically the XCKU040 model, serves as the target chip for testing. The article details the testing of the SerDes interface chip with a transmission rate of 16Gbps, encompassing functions such as sending/receiving tests. A hardware test PCB platform centered around an FPGA is constructed to facilitate various testing modes, including inner loop, outer loop, and built-in self-test modes.

Furthermore, the study establishes a hardware test PCB platform based on FPGA, primarily focusing on testing the functions of transmission (TX) and reception (Rx).

[8] Verification of SerDes Design Using UVM Methodology:

In the paper, the authors propose the design and verification of a Serializer/Deserializer (SerDes) system. They highlight the longstanding necessity for efficient data transmission since the inception of computers, emphasizing the challenges associated with parallel transmission such as skew, crosstalk, cost, and board space constraints in System on Chip (SoC) designs. To address these issues, SerDes technology is advocated as a solution for moving large volumes of data between points within a system, across different systems, or even between geographically separated systems.

The authors emphasize the advantages of SerDes technology, including its ability to transmit data at higher rates while being cost-effective. They discuss the use of Verilog Hardware Description Language (HDL) in the design of SerDes and the verification process employing Universal Verification Methodology (UVM). By leveraging UVM, the authors highlight the creation of a reusable test bench, which significantly reduces time to market, a crucial factor in today's fast-paced technology landscape.

[9] A 10 Gb/s SerDes Transceiver

The paper presents a Serial Data Link Transceiver designed to support data rate transfer up to 10 Gbit/s, implemented using 65-nm CMOS technology. The Transmitter is modeled using Verilog-A and features a 3-taps Finite Impulse Response Filter (FIR) along with Current Mode Logic (CML) drivers. On the other hand, the Receiver, which is DC-coupled, is driven by Power Management Units (PMU), Band Gap Reference (BGR), and Low Drop Out regulator (LDO). Equalization in the receiver is achieved through a Continuous Time Linear Equalizer (CTLE) with channel loss compensation of up to 7.5 dB, a Variable Gain Amplifier (VGA), and a programmable 3-taps Decision Feedback Equalizer (DFE). The sampling clock is obtained using a Clock and Data Recovery block (CDR). Both the transmitter and the receiver operate with a supply voltage of 1.2V, derived from a 1.8V voltage supply using PMU.
The average power consumption of the receiver is reported as 9.57 mW at a 10 Gbit/s rate. The transceiver's equalization performance is evaluated over a 30-inch FR4 channel, demonstrating compensation capabilities of up to 27 dB loss at the Nyquist frequency (5 GHz).

[10] A Low-Power SerDes for High-Speed On-Chip Networks

The paper introduces a 32:1 muxing and 1:32 demuxing Serializer/Deserializer (SerDes) designed for low-power on-chip networks. The deserializer component of the system integrates digital clock and data recovery (CDR) mechanisms, utilizing a multiplying delay-locked loop (MDLL) based frequency multiplier to generate a reference clock for CDR operations. Implemented in a 65 nm CMOS process, the proposed SerDes and MDLL achieve a measured data rate of 3.52 Gbps while facilitating 32:1 parallel-to-serial multiplexing and 1:32 serial-to-parallel demultiplexing conversion.
The SerDes occupies an active area of 0.19 mm^2 and consumes 14 mW of power, showcasing its efficiency and suitability for low-power on-chip network applications.

[11] A 40-Gb/s Quarter-Rate SerDes Transmitter and Receiver Chipset in 65-nm CMOS

The abstract presents a 40-Gb/s transmitter (TX) and receiver (RX) chipset tailored for chip-to-chip communications, fabricated using a 65-nm CMOS process. The TX module incorporates a quarter-rate multi-multiplexer (MUX)-based four-tap feed-forward equalizer (FFE). Notably, a charge-sharing-effect elimination technique is introduced into the 4:1 MUX to enhance its jitter performance and power efficiency.
On the other hand, the RX module employs a two-stage continuous-time linear equalizer as the analog front end and integrates a low-cost sign-based zero-forcing engine for automatic tap weights adjustment of the TX-FFE. The clock data recovery mechanism integrates low-pass filters with adaptively adjusting bandwidths into the data-sampling path, alongside high-linearity compensating phase interpolators, ensuring high jitter tolerance and low jitter generation.
In practical performance, the fabricated TX and RX chipset demonstrate the ability to deliver 40-Gb/s PRBS data at BER < 10^(-12) over channels with >16-dB loss at half-baud frequency, while maintaining a total power consumption of 370 mW.

[12] Flip Chip Package for 28GSerDes Interface

The paper discusses the successful design of a flip-chip package tailored for 28G-capable Serializer/Deserializer (SerDes) interfaces. Design optimization of the multi-layer 3D vertical Ball Grid Array (BGA) area is achieved through the utilization of an Electromagnetic (EM) solver to ascertain optimal insertion and return losses within manufacturing constraints. Various stripline and microstrip pair-to-pair spacings, each exhibiting different levels of coupling, were assessed in terms of eye opening and crosstalk-induced jitter.
Following complete die-to-package-to-board assembly, the performance of the system was evaluated at data rates up to 28G. It was found that both stripline and microstrip pairs demonstrated adequate performance, indicating the viability of the flip-chip package design for high-speed SerDes interfaces.

[13] A Low-power 3.52 Gbps SerDes with a MDLL Frequency Multiplier for High-speed On-chip Networks

The paper introduces a low-power 32:1-to-1:32 Serializer/Deserializer (SerDes) capable of operating at 3.52 Gbps, designed for on-chip serial-link networks. Key to its design is a multiplying delay-locked loop (MDLL) based frequency multiplier for efficient power consumption. The deserializer component incorporates a phase-interpolator (PI)-based 2x-overdamping digital Clock and Data Recovery (CDR) mechanism, crucial for recovering clock and data signals accurately.

Utilizing a 65 nm CMOS process, the proposed SerDes and MDLL system facilitate 32:1 parallel-to-serial multiplexing and 1:32 serial-to-parallel demultiplexing conversion. Notably, the MDLL frequency multiplier achieves a multiplication factor of N = 16, converting an input frequency of 110 MHz into an output frequency of 1.76 GHz.

Measured performance indicates the system's ability to achieve the desired data rate while occupying a minimal active area of 0.19 mm^2 and consuming only 14 mW of power, underscoring its suitability for low-power on-chip applications.

[14] Design of 56 Gb/s NRZ and PAM4 SerDes Transceivers in CMOS Technologies

The paper introduces two ultra-high-speed Serializer/Deserializer (SerDes) designs tailored for PAM4 and NRZ data transmission. The PAM4 transmitter (TX) incorporates an output driver with a 3-tap Feed-Forward Equalizer (FFE) and adjustable weighting to ensure clean outputs at 4 levels. Conversely, the PAM4 receiver (RX) features a purely linear full-rate Clock and Data Recovery (CDR) alongside Continuous Time Linear Equalization (CTLE) and a 1-tap Decision Feedback Equalizer (DFE) combination for data recovery and demultiplexing. For NRZ data transmission, the transmitter utilizes a tree-structure Multiplexer (MUX) with an integrated Phase-Locked Loop (PLL) and phase aligner. The NRZ receiver employs a linear Phase Detector (PD) with a special vernier technique to manage incoming 56 Gb/s data streams. All chips have undergone silicon verification with satisfactory performance. These designs serve as prospective examples for next-generation 400 Gigabit Ethernet (GbE) applications, showcasing advancements in SerDes technology.

3. Objectives

The main objective of this project is to design a serializer-deserializer (SerDes) which can serialize 32 bit parallel data onto 4 lanes and deserialize the data from 4 lanes back into 32 bit data output. This project also is aimed toward creating a reusable UVM environment to extensively test the functioning of the SerDes. Finally the RTL to GDS flow is followed to obtain the GDSII file. The objectives are further discussed in detail in the following section.

1. **Design Serializer:** The Serializer converts parallel data from a data source, such as a microprocessor or memory, into a serial data stream for transmission over a serial link. It typically consists of a shift register that receives parallel data in parallel-in serial-out (PISO) fashion and outputs it serially. The Serializer is controlled by a clock signal that determines the rate at which data is serialized.
    a. **Encoder:** The Encoder in the Serializer converts parallel data into a coded format for transmission. This coding can include error detection and correction codes, to ensure data integrity during transmission.
    b. **Parallel-to-Serial Converter:** This component converts parallel data from the Serializer into a serial format for transmission. It takes parallel data inputs and shifts them out sequentially, synchronized with the clock signal.

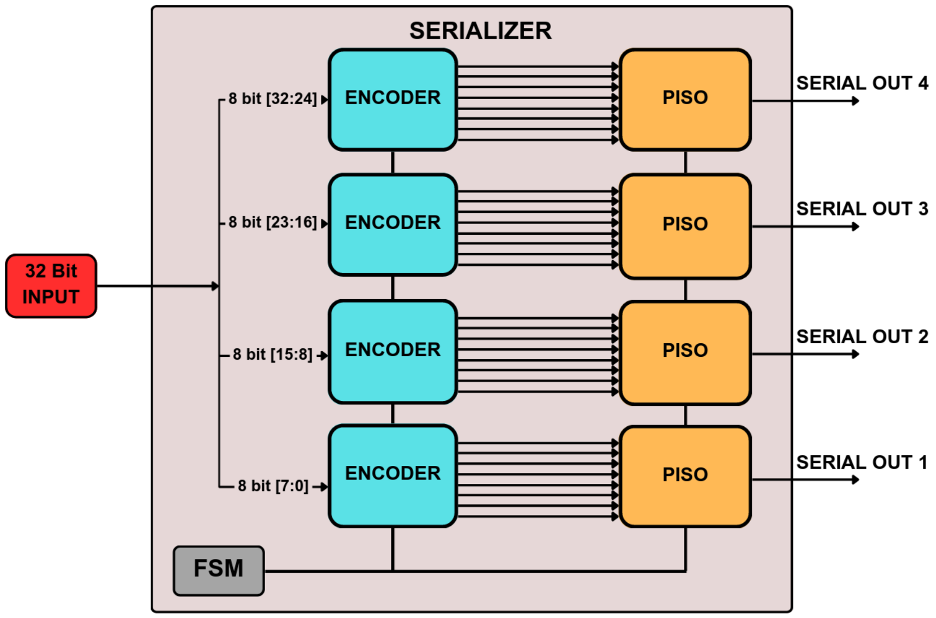


Fig 3.1 Serializer block diagram

2. **Design Deserializer:** The Deserializer receives the serial data stream from the transmission medium and converts it back into parallel data. It typically consists of a shift register that receives serial data in serial-in parallel-out (SIPO) fashion and aligns it into parallel data. The Deserializer also includes synchronization logic to ensure that the received data is correctly aligned with the clock signal.

   a. **Serial-to-Parallel Converter:** The Serial-to-Parallel Converter in the Deserializer converts the incoming serial data stream back into parallel data. It collects the serial bits into a shift register and then outputs the parallel data at the appropriate rate.
   b. **Decoder:** The Decoder in the Deserializer performs the inverse operation of the Encoder. It decodes the received data back into its original format, removing any error detection and correction codes that were added during encoding.

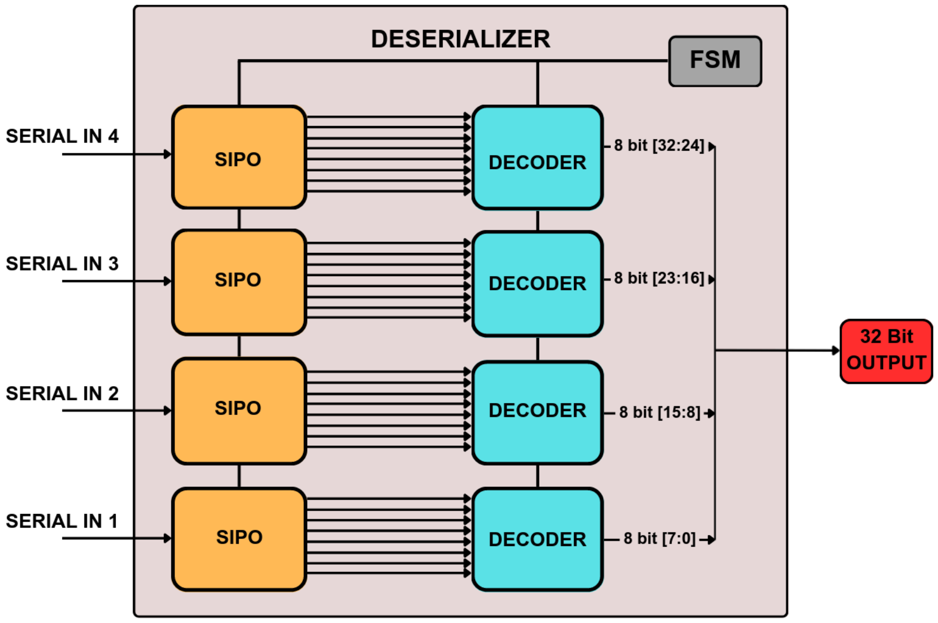


Fig 3.2 Deserializer block diagram

3. **Verify Each Component:** Perform functional verification by simulating each component to ensure that it behaves as expected and meets its specifications.

4. **Control Logic:** The control logic in the SerDes manages the operation of the Serializer and Deserializer. It controls the timing of data transmission and reception, handles data alignment and synchronization, and manages any error detection and correction mechanisms.

5. **Integrate All Components:** Connect the individual components (PISO, SIPO, encoder, and decoder) in a top module to form a complete SerDes system. Design the interfaces between the components to ensure proper data transfer and synchronization.

6. **Verify the Integrated System:** Verify that the integrated SerDes system meets all functional and timing requirements specified for the overall system. Perform comprehensive testing to ensure that the system operates correctly under various conditions and edge cases. Generate Coverage reports for the functional verification of the SerDes.

7. **Clock Gating:** Clock gating is a technique used in digital design to reduce power consumption by selectively stopping the clock signal to portions of a circuit when they are not in use. By gating the clock signal, unnecessary switching activity and power dissipation can be minimized in idle or unused parts of the circuit. This technique is particularly effective in reducing dynamic power consumption in high-speed designs where clock frequencies are high and switching activity is frequent.

8. **Synthesize Circuit with Clock Gating:** Use Genus tool to add clock gating feature to the circuit to optimize power consumption and synthesize the circuit, ensuring that it does not affect the performance or functionality of the design.

9. **Perform Equivalence Check:** Use Conformal tool to verify that the synthesized circuit with clock gating is functionally equivalent to the original design, ensuring that the synthesis and clock gating modifications have not affected the circuit.

10. **Perform Static Timing Analysis (STA):** Perform simulation on the synthesized netlist with delays obtained and analyze the timing of the circuit to ensure that all timing constraints, including setup and hold times, are met. Identify any timing violations and make necessary adjustments to the design.

11. **Floorplan, Placement, and Layout:** Use the tool Innovus to plan the physical layout of the SerDes system on the chip, considering factors such as IO pin placement, signal routing, power distribution, and thermal management. Place and route the components to minimize signal delays, ensure signal integrity, and meet area constraints.

12. **Fix Timing:** Iterate on the placement and routing of the components to resolve any timing violations and achieve timing closure. Use tools such as tempus and innovus to optimize timing and placement while minimizing impact on area and power.

By achieving these objectives, the project will result in a functional Serializer/Deserializer (SerDes) that reliably converts parallel data to serial data for transmission and vice versa. The SerDes will be capable of recognizing specified input patterns and generating appropriate outputs according to the provided relationship between input and output. It will demonstrate the desired behaviors, such as data synchronization, clock recovery, and data integrity, which are crucial for reliable communication in digital systems.

This SerDes will serve as a valuable tool for various applications, including high-speed data communication in digital logic circuits, control systems, and event detection systems. Its reliable and efficient operation will enhance the performance and functionality of these systems, enabling them to meet the requirements of modern communication standards and protocols.

4. Block Diagram

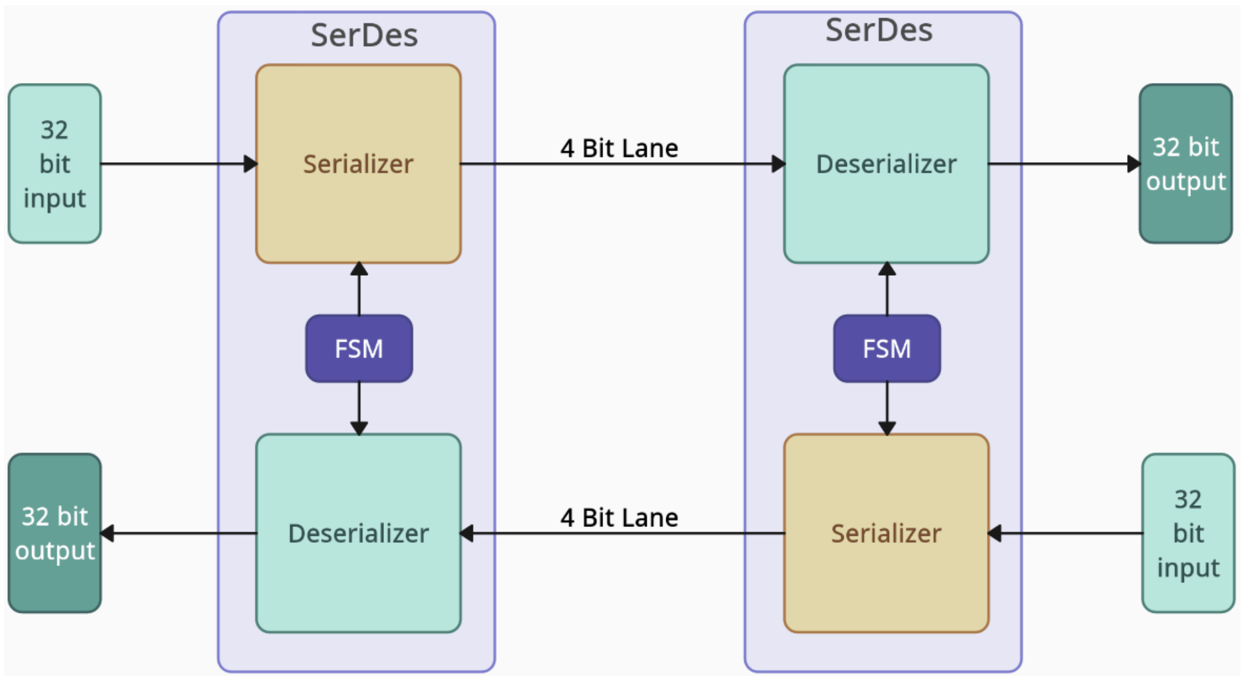


Fig 4.1 Full duplex SerDes Block Diagram

5. Methodology

A SerDes is mainly made up of 2 components, a serializer and a deserializer. Serializer of a SerDes sends the serial data to the deserializer of another SerDes and vice versa. This is achieved using various sub modules such as encoders, decoders, PISO and SIPO and other elements.

The following are the functional requirements of the SerDes module:

1. Encoding: Perform 8b/10b encoding on the parallel input data and give out encoded parallel data.
2. Data Serialization: Use PISO to convert parallel data inputs into a serial data stream for transmission.
3. Data Deserialization: Convert received serial data back into parallel data using SIPO.
4. Decoding: Perform 10b/8b decoding on the parallel data out of SIPO.
5. High-Speed Operation: Support high-speed data transmission rates, in the gigabit per second range.

The serializer works in stages, as follows:

1. Encoding: The parallel data is loaded onto the encoder when the encoder-enable is made high; the encoder module then encodes and gives us the output when the enable is made low. The code snippet of the encoder is provided below.

```
module encoder(
        input   logic   clk,
        input   logic   rst,
        input logic [7:0]       data_8b_in,
        input   logic   ser_en,
        output logic [9:0]      data_10b_out
);

        logic [3:0]     temp_4b;
        logic [5:0]     temp_6b;

        always @ (posedge clk) begin
```

```
if (rst) begin
temp_4b <= 4'b0000;
temp_6b <= 6'b000000;
end
else begin
if (ser_en) begin
        case (data_8b_in[7:5])
        3'b000: temp_4b <= 4'b0100;
        3'b001: temp_4b <= 4'b1001;
        3'b010: temp_4b <= 4'b0101;
        3'b011: temp_4b <= 4'b0011;
        3'b100: temp_4b <= 4'b0010;
        3'b101: temp_4b <= 4'b1010;
        3'b110: temp_4b <= 4'b0110;
        3'b111: temp_4b <= 4'b0001;
        default: temp_4b <= 4'b0000;
        endcase
        case (data_8b_in[4:0])
        5'b00000: temp_6b <= 6'b011000;
        5'b00001: temp_6b <= 6'b011101;
        5'b00010: temp_6b <= 6'b010010;
        5'b00011: temp_6b <= 6'b110001;
        5'b00100: temp_6b <= 6'b110101;
        5'b00101: temp_6b <= 6'b101001;
        5'b00110: temp_6b <= 6'b011001;
        5'b00111: temp_6b <= 6'b111000;
        5'b01000: temp_6b <= 6'b111001;
        5'b01001: temp_6b <= 6'b100101;
        5'b01010: temp_6b <= 6'b010101;
        5'b01011: temp_6b <= 6'b110100;
        5'b01100: temp_6b <= 6'b001101;
        5'b01101: temp_6b <= 6'b101100;
        5'b01110: temp_6b <= 6'b011100;
        5'b01111: temp_6b <= 6'b010111;
        5'b10000: temp_6b <= 6'b011011;
        5'b10001: temp_6b <= 6'b100011;
        5'b10010: temp_6b <= 6'b010011;
        5'b10011: temp_6b <= 6'b110010;
        5'b10100: temp_6b <= 6'b001011;
        5'b10101: temp_6b <= 6'b101010;
```

```
                5'b10110: temp_6b <= 6'b011010;
                5'b10111: temp_6b <= 6'b111010;
                5'b11000: temp_6b <= 6'b110011;
                5'b11001: temp_6b <= 6'b100110;
                5'b11010: temp_6b <= 6'b010110;
                5'b11011: temp_6b <= 6'b110110;
                5'b11100: temp_6b <= 6'b001110;
                5'b11101: temp_6b <= 6'b101110;
                5'b11110: temp_6b <= 6'b011110;
                5'b11111: temp_6b <= 6'b101011;
                default: temp_6b <= 6'b000000;
                endcase
        end
        else data_10b_out <= {temp_4b, temp_6b};
        end
        end
endmodule
```

2. PISO: A parallel in serial out converter is used to convert the encoded parallel data into serial data to be transmitted. The encoded data is loaded onto PISO when the PISO enable is made high. When the enable signal is made low the PISO starts to send the parallel data serially one bit at every clock edge, LSB first while using a shift counter which moves the data to right(towards MSB) and appends 0 at MSB.

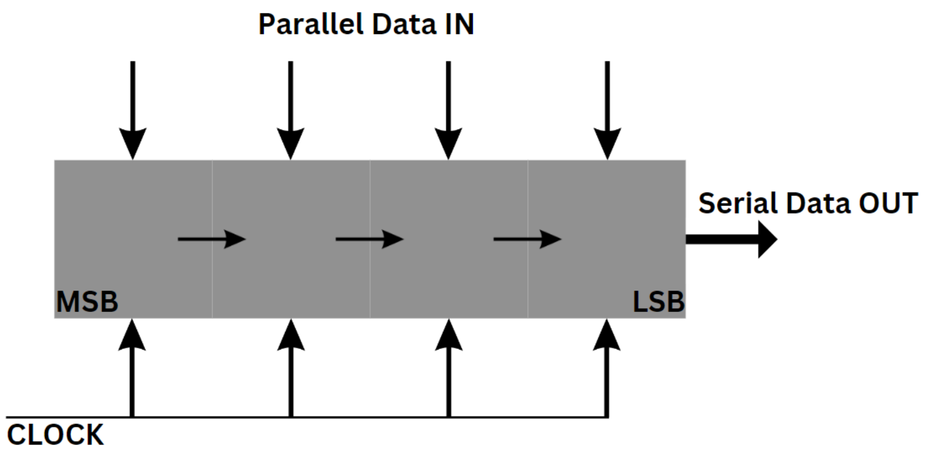


Fig 5.1 PISO shift register

The code snippet of the PISO module is given below.

```
module PISO(
        input logic clk,
        input logic rst,
        input logic [9:0] par_in,
        input logic load_en,
        output logic ser_out
);

        logic [9:0] shift_reg;

        always @ (posedge clk)begin
          if (rst) begin
        shift_reg <= 0;
          end
          else begin
        if (load_en) begin
                shift_reg <= par_in;
        end
        else begin
                ser_out <= shift_reg[0];
                shift_reg <= {1'b0, shift_reg[9:1]};
                end
          end
        end
endmodule
```

The deserializer works in stages, as follows:

1. SIPO: A serial in parallel out converter is used to convert the input serial data into parallel data, which is then fed to the decoder. When the SIPO enable is low, it loads the incoming data onto a shit register at MSB and successively pushes all other bits to right, discarding the LSB. The parallel data out is obtained when the enable is set to high.

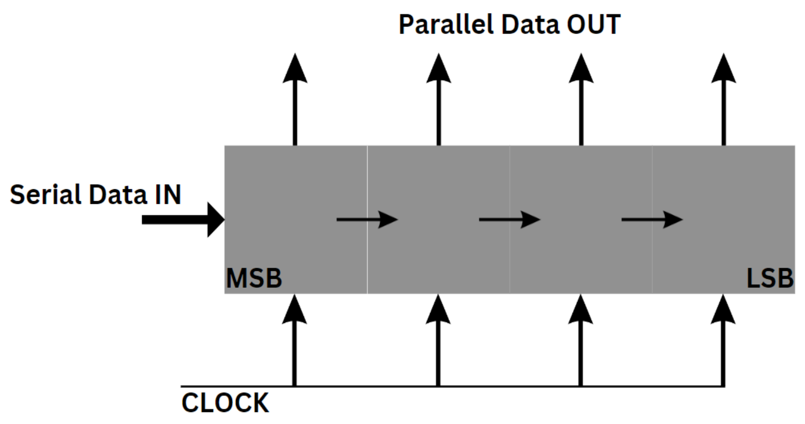


Fig 5.2 SIPO shift register

The code snippet of the SIPO module is given below.

```
module SIPO(
        input logic clk,
        input logic rst,
        input logic ser_in,
        input logic shift_en,
        output logic [9:0]      par_out
);

        logic [9:0]     shift_reg;

        always @ (posedge clk)begin
        if (rst) begin
        shift_reg <= 0;
        par_out <=0;
        end
        else begin
        if (shift_en) begin
                par_out <= shift_reg;
        end
        else
                shift_reg <= {ser_in,shift_reg[9:1] };
        end
        end

endmodule
```

2. Decoder: When the decoder enable is made high the parallel encoded data is loaded into the decoder and when the enable signal is made low the decoder provides the decoded parallel data out. The code snippet of the decoder module is given below.

```
module decoder(
        input   logic   clk,
        input           logic   rst,
        input logic [9:0]       data_10b_in,
        input   logic   par_en,
        output logic [7:0]      data_8b_out
);

        logic [2:0]     temp_3b;
        logic [4:0]     temp_5b;

        always @ (posedge clk) begin
        if (rst) begin
        temp_3b <= 4'b0000;
        temp_5b <= 6'b000000;
        end
        else begin
        if (par_en) begin
                case (data_10b_in[9:6])
                4'b0100: temp_3b <= 3'b000;
                4'b1001: temp_3b <= 3'b001;
                4'b0101: temp_3b <= 3'b010;
                4'b0011: temp_3b <= 3'b011;
                4'b0010: temp_3b <= 3'b100;
                4'b1010: temp_3b <= 3'b101;
                4'b0110: temp_3b <= 3'b110;
                4'b0001: temp_3b <= 3'b111;
                default: temp_3b <= 3'b000;
                endcase
                case (data_10b_in[5:0])
                6'b011000: temp_5b <= 5'b00000;
                6'b011101: temp_5b <= 5'b00001;
                6'b010010: temp_5b <= 5'b00010;
                6'b110001: temp_5b <= 5'b00011;
                6'b110101: temp_5b <= 5'b00100;
                6'b101001: temp_5b <= 5'b00101;
                6'b011001: temp_5b <= 5'b00110;
                6'b111000: temp_5b <= 5'b00111;
                6'b111001: temp_5b <= 5'b01000;
                6'b100101: temp_5b <= 5'b01001;
                6'b010101: temp_5b <= 5'b01010;
                6'b110100: temp_5b <= 5'b01011;
                6'b001101: temp_5b <= 5'b01100;
                6'b101100: temp_5b <= 5'b01101;
                6'b011100: temp_5b <= 5'b01110;
                6'b010111: temp_5b <= 5'b01111;
                6'b011011: temp_5b <= 5'b10000;
                6'b100011: temp_5b <= 5'b10001;
                6'b010011: temp_5b <= 5'b10010;
                6'b110010: temp_5b <= 5'b10011;
                6'b001011: temp_5b <= 5'b10100;
                6'b101010: temp_5b <= 5'b10101;
                6'b011010: temp_5b <= 5'b10110;
                6'b111010: temp_5b <= 5'b10111;
                6'b110011: temp_5b <= 5'b11000;
                6'b100110: temp_5b <= 5'b11001;
                6'b010110: temp_5b <= 5'b11010;
                6'b110110: temp_5b <= 5'b11011;
                6'b001110: temp_5b <= 5'b11100;
                6'b101110: temp_5b <= 5'b11101;
                6'b011110: temp_5b <= 5'b11110;
                6'b101011: temp_5b <= 5'b11111;
                default: temp_5b <= 5'b00000;
                endcase
        end
        else data_8b_out <= { temp_3b,temp_5b};
        end
        end

endmodule
```

All the submodules are instantiated and connected accordingly in the TOP module which also implements the control signals i.e enable high and low signals of encoder, PISO, SIPO and decoder. This is done by implementing a FSM, modeling the behavior of a SerDes. Two FSMs are implemented, one for the operation of the serializer and another for the deserializer.

The first iteration of the project consisted of designing a 8 bit SerDes consisting of single instances of all the sub modules and a single serial out and serial in line. To optimize the data rate and throughput of the SerDes system in the further iterations clock speed and serial in/out lines were increased whose affects on the overall performance of the system is further discussed in the results section. The input was expanded to 32 bits and the serial in/out lines were extended to 4 lines each. To achieve this instead of the single pair of Encoder-PISO and SIPO-Decoder, 4 such pairs were used where each pair can handle 8 bits of data accounting to a total of 32 bits of data handled by the serializer and 32 bits handled by deserializer.

In the final iteration of the project the top SerDes module can be broken down into the following components:

1. Inputs and Outputs:

   Inputs:

   a. data_in (32 bits): The input parallel data that needs to be serialized.
   b. clk (1 bit): The clock signal used to synchronize the operations of the modules.
   c. rst (1 bit): The reset signal used to initialize the modules and reset the state machines.
   d. serial_in (4 x 1bit) : the serial input lines for the input of deserializer.
   e. start_i: start signal for the deserializer.

   Outputs:

   a. data_out (32 bits): The output parallel data that has been deserialized
   b. Serial_out (4 x 1 bit): the serial out lines from the output of serializer.
   c. start_o: the start signal, which is the output of the serializer.

2. Internal Signals:

   Serialization Control Signals:

   a. en_en: Enables the encoder to encode the input data.
   b. p_en: Enables the PISO converter to shift out the encoded data.
   c. s_en: Enables the SIPO converter to shift in the serial data.
   d. de_en: Enables the decoder for decoding the serial data.

   State Machines:

   a. s_state: State machine for controlling the serializer operation.
   b. d_state: State machine for controlling the deserializer operation.

   Other Signals:

   a. start: Signal to indicate the start of the deserialization process.
   b. count and count_2: Counters for controlling the shifting process in the serializer and deserializer, respectively.

Operation of the TOP module is explained in the following section.

Serializer Operation:

- When s_state is in state S0, the serializer is in the idle state. The encoder is enabled (en_en) to encode the input data (data_in).
- In state S2, the PISO converter is enabled (p_en) to load and shift out the encoded data (en_out) serially.
- State S4 manages the shifting process of the PISO converter. Once all bits are shifted out, s_state transitions back to S0.

Deserializer Operation:

- The deserializer starts in state P0, where it waits for the start signal to be asserted.
- In state P1, the SIPO converter is enabled (s_en) to load and shift in the serial data (ser_out).
- State P2 enables the decoder (de_en) to decode the parallel data (si_out).
- Once all bits are shifted in, d_state transitions back to P0.

State Machine Transition:

- State transitions are controlled by clock edges. Each state machine transitions to the next state on the rising edge of the clock.

Clock and Reset Handling:

- The clock signal (clk) is used to synchronize all operations in the top module.
- The reset signal (rst) is used to initialize the state machines and modules to their initial states.

Data Flow:

- The data_in is serialized and transmitted serially through the encoder and PISO converter.
- The serial data is received by the SIPO converter and deserialized back into parallel form, which is then output as data_out.

The top module acts as the orchestrator for the entire SerDes system, coordinating the serialization and deserialization of data. It controls the timing and operation of the encoder, PISO, SIPO, and decoder modules to ensure that data is correctly encoded, transmitted, received, and decoded.

The code snippet of the TOP module is provided in the following pages.

```
module SerDes(
        input logic [31:0] data_in,
        output logic [31:0] data_out,
        input logic clk,
        input logic rst,
        output logic ser_out,
        output logic ser_out2,
        output logic ser_out3,
        output logic ser_out4,
        input logic ser_in,
        input logic ser_in2,
        input logic ser_in3,
        input logic ser_in4,
        output logic start_o,
        input logic start_i
);

  logic [9:0] en_out, en_out2, en_out3, en_out4;
  logic [9:0] si_out, si_out2, si_out3, si_out4;

  logic en_en, de_en, p_en, s_en;
  reg [2:0] s_state, s_next;
  reg [3:0] d_state, d_next;
  logic [3:0] count, count_2;

  encoder E1(.clk(clk), .rst(rst), .data_8b_in(data_in[7:0]), .ser_en(en_en),
.data_10b_out(en_out));
  PISO P1S01(.clk(clk), .rst(rst), .par_in(en_out), .load_en(p_en), .ser_out(ser_out));

  SIPO S1P01(.clk(clk), .rst(rst), .ser_in(ser_in), .shift_en(s_en), .par_out(si_out));
  decoder D1(.clk(clk), .rst(rst), .data_10b_in(si_out), .par_en(de_en),
.data_8b_out(data_out[7:0]));

  encoder E2(.clk(clk), .rst(rst), .data_8b_in(data_in[15:8]), .ser_en(en_en),
.data_10b_out(en_out2));
  PISO P1S02(.clk(clk), .rst(rst), .par_in(en_out2), .load_en(p_en), .ser_out(ser_out2));

  SIPO S1P02(.clk(clk), .rst(rst), .ser_in(ser_in2), .shift_en(s_en), .par_out(si_out2));
  decoder D2(.clk(clk), .rst(rst), .data_10b_in(si_out2), .par_en(de_en),
.data_8b_out(data_out[15:8]));

  encoder E3(.clk(clk), .rst(rst), .data_8b_in(data_in[23:16]), .ser_en(en_en),
.data_10b_out(en_out3));
  PISO P1S03(.clk(clk), .rst(rst), .par_in(en_out3), .load_en(p_en), .ser_out(ser_out3));

  SIPO S1P03(.clk(clk), .rst(rst), .ser_in(ser_in3), .shift_en(s_en), .par_out(si_out3));
  decoder D3(.clk(clk), .rst(rst), .data_10b_in(si_out3), .par_en(de_en),
.data_8b_out(data_out[23:16]));

  encoder E4(.clk(clk), .rst(rst), .data_8b_in(data_in[31:24]), .ser_en(en_en),
.data_10b_out(en_out4));
```

```
  PISO P1S04(.clk(clk), .rst(rst), .par_in(en_out4), .load_en(p_en), .ser_out(ser_out4));

  SIPO S1P04(.clk(clk), .rst(rst), .ser_in(ser_in4), .shift_en(s_en), .par_out(si_out4));
  decoder D4(.clk(clk), .rst(rst), .data_10b_in(si_out4), .par_en(de_en),
.data_8b_out(data_out[31:24]));

  parameter S0 = 4'b0000;
  parameter S1 = 4'b0001;
  parameter S2 = 4'b0010;
  parameter S3 = 4'b0011;
  parameter S4 = 4'b0100;

  parameter P0 = 4'b0101;
  parameter P1 = 4'b0110;
  parameter P2 = 4'b0111;
  parameter P3 = 4'b1000;

  always@(posedge clk)begin
        if(rst)begin
        s_state <= S0;
        d_state <= P0;
        end
        else begin
        s_state <= s_next;
        d_state <= d_next;
        end
  end

  always@(posedge clk)
  begin
        case(s_state)
        S0 :  begin
                en_en = 1;
                p_en = 0;
                start_o = 0;
                s_next = S1;
                end
        S1 :  begin
                en_en = 0;
                p_en = 0;
                count = 4'b0000;
                start_o = 0;
                s_next = S2;
                end
        S2 : begin
                p_en = 1;
                s_next = S3;
                start_o=1;
        end
        S3 : begin
                p_en = 0;
                s_next = S4;
        end
        S4 : if(count == 4'b1010) begin
                start_o = 0;
                en_en=1;
                s_next = S1;
        end
        else begin
                count = count + 4'b001;
                s_next = S4;
                end
        endcase

        case(d_state)
        P0 : if (start_i == 1) begin
                s_en = 0;
                de_en = 0;
                count_2 = 4'b0000;
                d_next = P1;
        end else begin
                s_en = 1;
                d_next = P0;
        end
        P1 : if(count_2 == 4'b1010)begin
                s_en = 1;
                count_2 = 4'b0000;
                d_next = P2;
        end else begin
                count_2 = count_2 + 4'b0001;
                d_next = P1;
        end
        P2 : begin
                s_en = 0;
                de_en = 1;
                d_next = P3;
        end
        P3 : begin
                de_en = 0;
                d_next = P0;
        end
        endcase
        end
endmodule
```

6. VERIFICATION

Verification is a critical aspect of the design process for Serializer/Deserializer (SerDes) modules, as it ensures that the implemented design functions correctly and meets the specified requirements. For SerDes, which are integral to high-speed data transmission in modern electronics, verification is even more crucial due to the complex nature of the functionality and the high data rates involved. Verifying a SerDes design involves testing its encoding, serialization, deserialization, and decoding capabilities under various scenarios and conditions to ensure reliable data transmission. Failure to verify a SerDes design adequately can result in costly errors, such as data corruption or loss, which can impact the overall performance and reliability of the system in which the SerDes is used. The following are the various parts of the UVM test bench implemented.

1. Interface:

   The SerDes_if interface defines the communication interface for a Serializer/Deserializer (SerDes) module. It includes signals for clock (clk), reset (rst), parallel data input (data_in), and parallel data output (data_out). The clocking block cb defines the timing and synchronization requirements for the interface signals. Specifically, it specifies that the data_out signal is sampled on the positive edge of the clock (posedge clk), and the data_in signal is driven after a delay of 1 time unit (#1step) and held for 3 nanoseconds (#3ns). This interface is essential for connecting the SerDes module to the test environment and ensures proper timing and data flow during verification.

2. Sequence Item:

   The Item class, as a UVM sequence item, serves as a structured representation of a data transaction within the SerDes verification environment. It encapsulates two key components: data_in, a random 32-bit vector representing input data, rst a 1 bit reset signal, and data_out ,a 32-bit vector representing output data. The rand keyword enables randomization of data_in and rst facilitating diverse test scenarios. The convert2str function provides a human-readable string representation of the Item object, aiding in debugging and logging. Overall, the Item class streamlines data handling and representation, crucial for modeling and verifying SerDes data transactions in a structured and efficient manner within the UVM framework.

```
class Item extends uvm_sequence_item;
  `uvm_object_utils(Item)
  rand bit [31:0] data_in;
  rand bit rst;
  bit [31:0] data_out;

  virtual function string convert2str();
        return $sformatf("data_in=%0d, data_out=%0d", data_in, data_out);
  endfunction

  function new(string name = "Item");
        super.new(name);
  endfunction

endclass
```

3. Sequence

The gen_item class, a UVM sequence, is designed to generate test items for verifying a Serializer/Deserializer (SerDes) module. It utilizes the UVM methodology to provide structured and reusable testbench development. The for loop controls the number of iterations in the body() task, ensuring controlled yet randomized test item generation. Within the body() task, each iteration creates a new Item object, dynamically instantiated using Item::type_id::create("m_item");. Transactional behavior is managed using start_item(m_item); and finish_item(m_item);, ensuring proper handling within the UVM framework. The m_item.randomize(); call randomizes the data_in field of each Item object, facilitating diverse test scenarios. Informative log messages, generated using $sformatf, detail the contents of the generated items, aiding in debugging and analysis.

Overall, the gen_item sequence demonstrates how UVM sequences can automate complex test scenario generation, enhancing the efficiency and effectiveness of SerDes design verification.

```
class gen_item extends uvm_sequence;
  `uvm_object_utils(gen_item)

  function new(string name="gen_item");
        super.new(name);
  endfunction

  virtual task body();
        for(int i=0; i < 10000; i++) begin
        Item m_item = Item::type_id::create("m_item");
        start_item(m_item);
        m_item.randomize();
        `uvm_info("SEQ", $sformatf("Generate new item: %s", m_item.convert2str()),
UVM_HIGH);
        finish_item(m_item);
        end
        `uvm_info("SEQ", $sformatf("Done generation of %0d items", num),
UVM_LOW)

  endtask
endclass
```

4. Sequencer

The default UVM sequencer has been used in our project to manage the flow of transactions between the sequencer and the driver, serving as a central controller for transaction distribution. Working alongside sequences, which specify transaction sequences, the sequencer facilitates the creation of diverse test scenarios. It communicates with the driver through the seq_item_port interface, ensuring transactions are sent and synchronized effectively with the DUT's clock. This approach has significantly streamlined our verification efforts, particularly in verifying the complex functionality of our Serializer/Deserializer (SerDes) module.

5. Driver

The driver class is a UVM component responsible for driving stimulus into the Serializer/Deserializer (SerDes) module during verification. It extends the uvm_driver templated with Item, indicating that it handles items of type Item(sequence item). The build_phase function initializes the vif (Virtual Interface) by retrieving it from the UVM configuration database. The run_phase task, which runs indefinitely, waits for an item from the sequencer using seq_item_port.get_next_item(m_item), drives the item into the SerDes using drive_item(m_item), and then notifies the sequencer that the item has been processed with seq_item_port.item_done(). The drive_item task drives the data_in and rst field of the m_item into the SerDes through the vif.cb clocking block. This class demonstrates the role of the driver in coordinating the flow of test items between the sequencer and the SerDes, crucial for verifying the SerDes module's functionality under different test scenarios.

```
class driver extends uvm_driver #(Item);
 `uvm_component_utils(driver)

 function new(string name = "driver", uvm_component parent=null);
        super.new(name, parent);
 endfunction

 virtual SerDes_if vif;

 virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        if (!uvm_config_db#(virtual SerDes_if)::get(this, "", "des_vif", vif))
        `uvm_fatal("DRV", "Could not get vif")
  endfunction

  virtual task run_phase(uvm_phase phase);
        super.run_phase(phase);
        forever begin
        Item m_item;
        `uvm_info("DRV", $sformatf("Wait for item from sequencer"), UVM_HIGH)
        seq_item_port.get_next_item(m_item);
        drive_item(m_item);
        seq_item_port.item_done();
        end
  endtask

  virtual task drive_item(Item m_item);
        @(vif.cb);
        vif.cb.data_in <= m_item.data_in;
        vif .rst <= m_item.rst;
  endtask
endclass
```

6. Monitor

The monitor class in this UVM environment is crucial for observing and capturing the behavior of the SerDes module. It interfaces with the module through a virtual SerDes_if called vif, monitoring its signals and interactions. During the run_phase, the monitor continuously checks for a complete transaction by waiting for a rising clock edge (@(vif.cb);) and ensuring the reset signal (vif.rst) is inactive. When a complete transaction is detected, the monitor creates a new Item object, populates it with the relevant data (vif.data_in, vif.rst and vif.cb.data_out), and writes it to the mon_analysis_port for further analysis. This class facilitates detailed analysis and verification of the SerDes module's functionality by providing visibility into its internal operations.

```
class monitor extends uvm_monitor;
 `uvm_component_utils(monitor)

 function new(string name="monitor", uvm_component parent=null);
         super.new(name, parent);
 endfunction

 uvm_analysis_port  #(Item) mon_analysis_port;
 virtual SerDes_if vif;

 virtual function void build_phase(uvm_phase phase);
         super.build_phase(phase);
         if (!uvm_config_db#(virtual SerDes_if)::get(this, "", "des_vif", vif))
         `uvm_fatal("MON", "Could not get vif")
         mon_analysis_port = new ("mon_analysis_port", this);
 endfunction

 virtual task run_phase(uvm_phase phase);
         super.run_phase(phase);
         forever begin
         @ (vif.cb);
         if (!vif.rst) begin
         Item item = Item::type_id::create("item");
         item.data_in = vif.data_in;
         item.rst = vif.rst;
         item.data_out = vif.cb.data_out;
         mon_analysis_port.write(item);
```

```
          `uvm_info("MON", $sformatf("Saw item %s", item.convert2str()),
UVM_HIGH)
          end
          end
  endtask
endclass
```

7. Agent

The agent class in this UVM environment serves as a central coordinator, managing the interactions between the driver, monitor, and sequencer components. During the build_phase, the agent creates instances of these components (driver, monitor, sequencer) using type_id::create, ensuring they are correctly instantiated. The connect_phase establishes the connection between the sequencer's seq_item_export and the driver's seq_item_port, enabling the flow of transactions from the sequencer to the driver. This flow allows the agent to control the generation, driving, and monitoring of transactions within the verification environment. Overall, the agent class plays a pivotal role in organizing and orchestrating the components of the UVM environment to verify the functionality of the DUT effectively.

```
class agent extends uvm_agent;
  `uvm_component_utils(agent)
  function new(string name="agent", uvm_component parent=null);
          super.new(name, parent);
  endfunction

  driver           d0;
  monitor          m0;
  uvm_sequencer #(Item)    s0;

  virtual function void build_phase(uvm_phase phase);
```

```
          super.build_phase(phase);
          s0 = uvm_sequencer#(Item)::type_id::create("s0", this);
          d0 = driver::type_id::create("d0", this);
          m0 = monitor::type_id::create("m0", this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
          super.connect_phase(phase);
          d0.seq_item_port.connect(s0.seq_item_export);
  endfunction

endclass
```

8. Scoreboard

This scoreboard class serves as a critical component within a UVM verification environment, ensuring the accuracy and reliability of a design's functionality. Its functionality revolves around a FIFO (First-In-First-Out) buffer, which acts as a reference for expected data transactions. Upon receiving data items, the scoreboard meticulously verifies them against the expected values stored in the FIFO buffer. It dynamically adjusts pointers within the buffer to simulate the flow of data, accommodating both reset signals and regular data transactions. This adaptability enables the scoreboard to accurately track the sequence of expected data and validate each incoming item accordingly. Additionally, the scoreboard maintains a history of previously observed data to facilitate comparison with subsequent outputs, ensuring consistency and correctness. Error reporting mechanisms, such as UVM macros, are employed to promptly flag any discrepancies between expected and actual results, providing valuable insights into potential design flaws or verification issues. Overall, this scoreboard's comprehensive functionality contributes significantly to the verification process, enhancing the confidence and reliability of the verified design.

```
class scoreboard extends uvm_scoreboard;
        `uvm_component_utils(scoreboard)

        logic [31:0] fifo [0:9];
        int unsigned push_ptr = 0;
        int unsigned pop_ptr = 1;
        logic [31:0] prev = 32'bx;

        function new(string name="scoreboard", uvm_component parent=null);
        super.new(name, parent);
        endfunction

        uvm_analysis_imp #(Item, scoreboard) m_analysis_imp;

        virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        m_analysis_imp = new("m_analysis_imp", this);
        foreach (fifo[i]) begin
        fifo[i] = 32'b0;
        end
        endfunction

        virtual function void write(Item item);
        if(item.rst) begin
        pop_ptr = pop_ptr+1;
        end
        else if(!item.rst) begin
        if (item.data_in != fifo[push_ptr]) begin
        push_ptr = (push_ptr == 9) ? 0 : push_ptr + 1;
        fifo[push_ptr] = item.data_in;
        end
        if (item.data_out != prev) begin
        pop_ptr = (pop_ptr >= 9) ? 0 : pop_ptr + 1;
        if (fifo[pop_ptr] != item.data_out) begin
                `uvm_error("SCBD", $sformatf("FAIL! data_out=%0d expected=%0d",
item.data_out, fifo[pop_ptr]))
        end else begin
                `uvm_info("SCBD", $sformatf("PASS! data_out=%0d expected=%0d",
item.data_out, fifo[pop_ptr]), UVM_MEDIUM)
        end
        end
        prev = item.data_out;
        end
        endfunction
endclass
```

9. Environment

The env class in this UVM environment acts as the overarching framework for organizing and connecting various components to facilitate the verification process. Its build_phase method instantiates a scoreboard and an agent component which in turn manages the driver, monitor, and sequencer. The connect_phase is used to establish connection between the monitor's analysis port and the scoreboard's analysis export for result analysis. Overall, the env class plays a crucial role in structuring the UVM environment, ensuring that components are created, connected, and orchestrated effectively to verify the functionality of the Design Under Test (DUT).

```
class env extends uvm_env;
  `uvm_component_utils(env)
  function new(string name="env", uvm_component parent=null);
        super.new(name, parent);
  endfunction


  agent          a0;
  scoreboard   sb0;

  virtual function void build_phase(uvm_phase phase);
          super.build_phase(phase);
          a0 = agent::type_id::create("a0", this);
          sb0 = scoreboard::type_id::create("sb0", this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
          super.connect_phase(phase);
          a0.m0.mon_analysis_port.connect(sb0.m_analysis_imp);
  endfunction
endclass
```

10. Test

The base_test class in this UVM environment defines a basic test scenario for verifying the SerDes module. It instantiates an env component to manage the verification environment, including the agent and its components. In the build_phase, it retrieves a virtual interface vif from the UVM configuration database and sets it for the agent within the environment. The test also creates a gen_item sequence to generate test items and randomizes its data. During the run_phase, the test raises an objection to begin the test, applies a reset to the SerDes module, and starts the gen_item sequence on the sequencer. After all the sequence items are driven, it then waits for 14 time units before dropping the objection to end the test. This test scenario demonstrates a basic flow of operations for verifying a SerDes module using the UVM methodology, including environment setup, test execution, and result analysis.

```
`include"package.sv"
class base_test extends uvm_test;
 `uvm_component_utils(base_test)
 function new(string name = "base_test", uvm_component parent=null);
          super.new(name, parent);
 endfunction

 env  e0;
 gen_item seq;
 virtual  SerDes_if vif;

 virtual function void build_phase(uvm_phase phase);
          super.build_phase(phase);

          e0 = env::type_id::create("e0", this);

          if (!uvm_config_db#(virtual SerDes_if)::get(this, "", "SerDes_vif", vif))
          `uvm_fatal("TEST", "Did not get vif")
          uvm_config_db#(virtual SerDes_if)::set(this, "e0.a0.*", "des_vif", vif);

          seq = gen_item::type_id::create("seq");
          seq.randomize();

 endfunction

 virtual task run_phase(uvm_phase phase);
          phase.raise_objection(this);
          apply_reset();
          seq.start(e0.a0.s0);
          #14;
```

```
        phase.drop_objection(this);
  endtask

  virtual task apply_reset();
        vif.rst <= 1;
        vif.data_in <= 0;
        repeat(1) @ (posedge vif.clk);
        vif.rst <= 0;
        repeat(1) @ (posedge vif.clk);
        vif.rst <= 1;
        repeat(1) @ (posedge vif.clk);
        vif.rst <= 0;
        repeat(1) @ (posedge vif.clk);
  endtask
endclass
```

11. Top

The top module of this UVM environment, tb, instantiates the components required for the testbench. It defines a clock signal clk and toggles it every 0.5 time units to simulate a clock with frequency 1GHz. The SerDes_if interface instance _if is created using the clk signal. The top module instance u0 is instantiated with connections to the clock signal, reset signal (_if.rst), and data signals (_if.data_in and _if.data_out). In the initial block, the clock signal is initially set to 0, and the virtual interface _if is set in the UVM configuration database to be accessible by the UVM test. Finally, the run_test function is called to start the UVM test named "test_SerDes". The initial block also includes commands to dump variable values and waveform information to a VCD file for further analysis. This top module sets up the test environment for the UVM test, providing the necessary interfaces and configurations for verifying the SerDes module.

Additionally, covergroups to gather functional coverage data are implemented. Cover group cg_data is written to gather coverage on input data values and covergroup cg_FSM_1 is written to gather coverage on the valid states, valid transitions and valid reset transitions on the 2 FSMs in the design. Assertions are written to check proper functioning of the ‘start’ signal and the proper functioning of the PISO and SIPO. This is done by making sure the start signal stays active only for 11 cycles and PISO SIPO functionality is checked by asserting that the data out of the encoder is the same as the data out of PISO.

```
`include "uvm_macros.svh"
import uvm_pkg::*;
`include "base_test.sv"
module tb;
        reg clk;

        always #0.5 clk = ~clk;

        SerDes_if _if (clk);
        top u0 (
        .clk(clk),
        .rst(_if.rst),
        .data_in(_if.data_in),
        .data_out(_if.data_out)
        );

        covergroup cg_data @(posedge clk);
        option.per_instance = 1;
        valid_data : coverpoint _if.data_in {
        bins default_bin = default; // Default bin to capture unspecified values
        bins till_eightG = {[0:858993459]};
        bins till_one_SevenG = {[858993460:1717986919]};
        bins till_remaining = {[1717986920:$]};
        }
        endgroup : cg_data;
        covergroup cg_FSM_1 @ (posedge clk);
        option.per_instance = 1;
        valid_ser : coverpoint u0.s_state iff (_if.rst == 0){
        bins valid_state[] = {u0.S0, u0.S1, u0.S2, u0.S3, u0.S4};
        bins valid_tran_ser[] = (u0.S1 => u0.S2 ), (u0.S2 => u0.S3), (u0.S3 => u0.S4),
(u0.S4 => u0.S4), (u0.S4 => u0.S1);
        bins reset_ser = (u0.S1, u0.S2, u0.S3, u0.S4 => u0.S0);
        bins others_ser[] = default;
        }
        valid_des : coverpoint u0.d_state iff (_if.rst == 0) {
        bins valid_des[] = {u0.P0, u0.P1, u0.P2, u0.P3};
        bins valid_tran_des[] = (u0.P0 => u0.P1),(u0.P1 => u0.P1), (u0.P1 => u0.P2),
(u0.P2 => u0.P3), (u0.P3 => u0.P0);
        bins reset_des = (u0.P1, u0.P2, u0.P3 => u0.P0);
        bins others_des[] = default;
        }

        cross_cov : cross u0.s_state , u0.d_state;
        endgroup : cg_FSM_1;

        property start_goes_low_after_11_clocks;
        @(posedge clk)
          disable iff(_if.rst)
            ($rose(u0.start) |-> ##11 $fell(u0.start));
        endproperty

        property en_equals_si_on_change_after_rst;
        @(posedge clk)
          disable iff (_if.rst)
            ($changed(u0.si_out) && !$past(_if.rst,2)) |-> (u0.si_out == u0.en_out);
        endproperty

        assert property (start_goes_low_after_11_clocks)
        else $error("Assertion failed: start did not go low after 11 clocks");
        assert property (en_equals_si_on_change_after_rst)
        else $error("Assertion failed: en_out is not equal to si_out");

        cg_FSM_1 fc_inst = new;
        cg_data fc_inst2 = new;

        always @(posedge clk) begin
        fork
        fc_inst.sample();
        fc_inst2.sample();
        join
        end
        initial begin
        clk = 0;
        uvm_config_db#(virtual SerDes_if)::set(null, "uvm_test_top", "SerDes_vif",
_if);
        run_test("test_SerDes");
        end
        initial begin
        $dumpvars;
        $dumpfile("dump.vcd");
        end
endmodule
```

7. Physical Design

Physical design is crucial in the development of Serializer/Deserializer (SerDes) systems as it focuses on translating the logical design into a physical layout that can be fabricated. This process is essential for ensuring that the SerDes module meets the required performance, power, and area targets. Physical design involves floorplanning, placement, and routing of the various components of the SerDes circuit to minimize signal delays, optimize power consumption, and reduce area overhead. Implementing physical design for SerDes is necessary to address key considerations such as high-speed operation, signal integrity, and clock distribution. It helps in meeting timing requirements, reducing crosstalk, and ensuring that the layout is robust against process variations. Overall, physical design plays a vital role in realizing the functionality and performance goals of SerDes systems, making it an integral part of the design process.

SerDes modules operate at high speeds, often in the gigabit per second range. Physical design ensures that signal paths are carefully laid out to minimize propagation delays, skew, and signal integrity issues, which can degrade the quality of the transmitted data. Secondly, physical design helps in optimizing power consumption. By carefully placing and routing components, power distribution networks can be designed to minimize voltage drop and ensure that power is efficiently distributed throughout the circuit. Thirdly, physical design considers the layout's manufacturability and reliability. It takes into account factors such as process variations, electromigration, and thermal effects, ensuring that the layout is robust and reliable under various operating conditions. Furthermore, physical design plays a crucial role in meeting stringent timing requirements. It involves detailed timing analysis and optimization to ensure that setup and hold times are met, and that there are no timing violations that could lead to functional failures.

The following are the steps followed for the physical design of the SerDes.

1. Synthesis:

   Synthesis is vital in the design of Serializer/Deserializer (SerDes) systems due to their complexity and stringent performance requirements. In SerDes, synthesis plays a crucial role in translating high-level RTL descriptions of the design into optimized gate-level representations. This process is essential for achieving the required performance, area, and power efficiency in SerDes designs. Synthesis tools, such as Genus, are instrumental in this process. Genus helps map the RTL description to the target technology library, optimizing the design for area, power, and performance. It also performs critical timing analysis to ensure that the design meets its timing requirements, which are crucial in high-speed SerDes applications. Furthermore, Genus enables design exploration, allowing designers to experiment with different optimizations and configurations to achieve the best possible design for their SerDes system. Overall, synthesis, particularly with tools like Genus, is indispensable in the development of high-performance and efficient SerDes systems. We synthesize the SerDes with Clock Gating enabled to reduce power consumption.

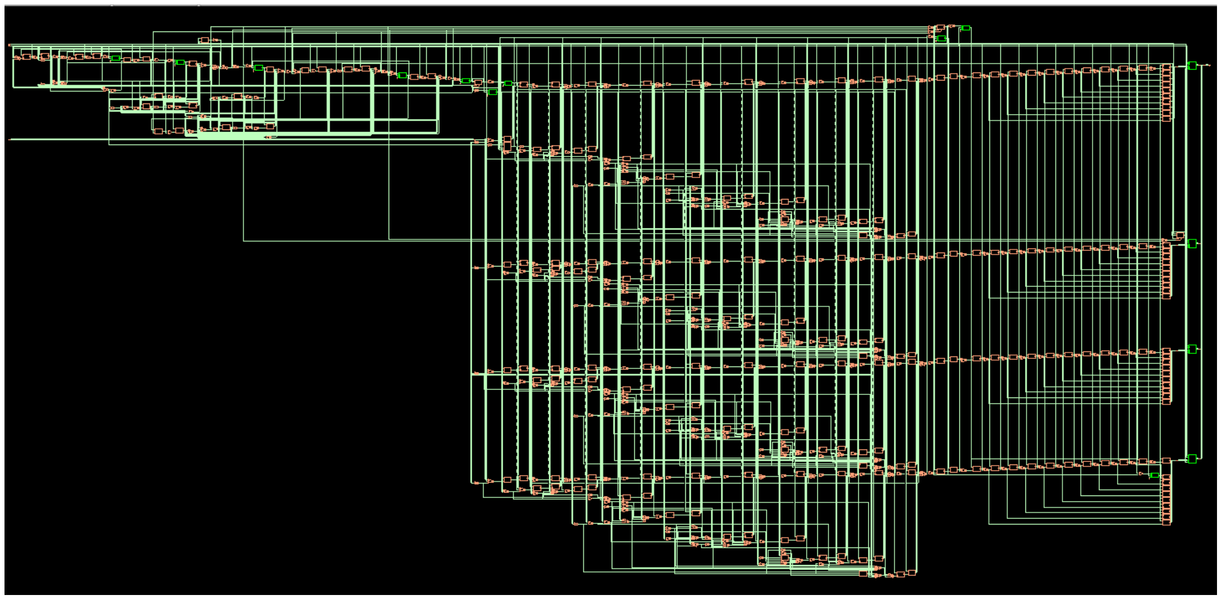
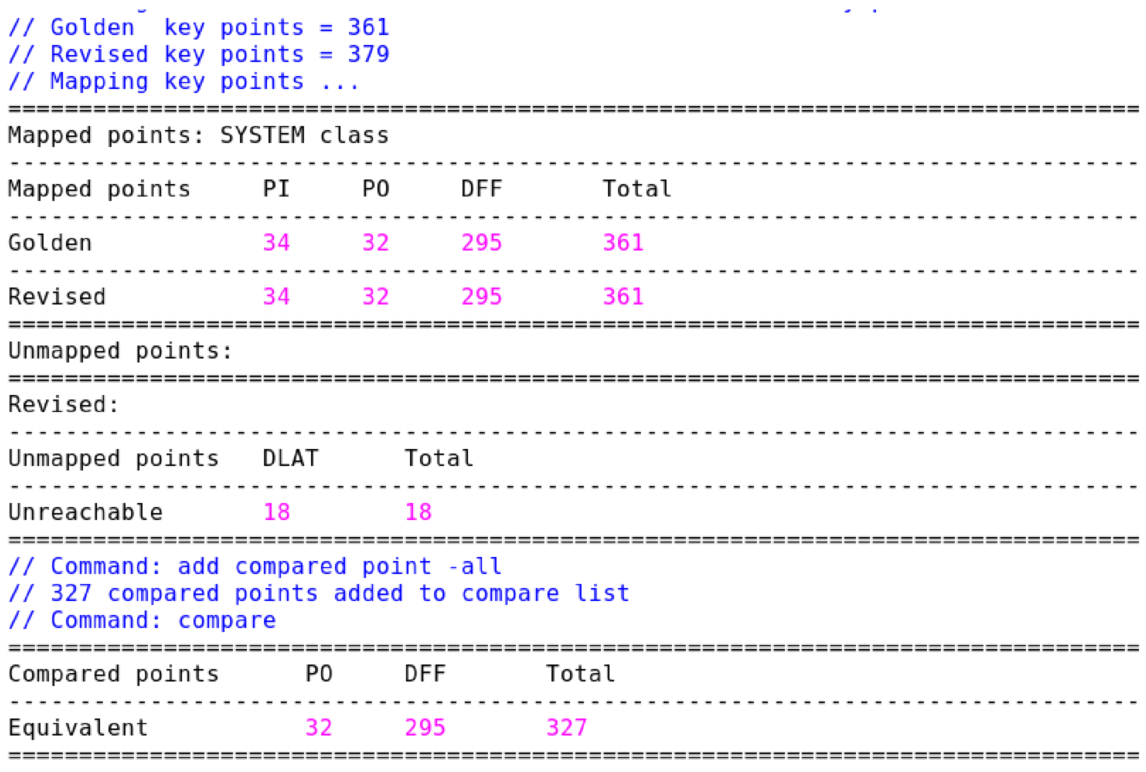


Fig 7.1 Synthesized circuit schematic

2. Logic Equivalence Check:

Performing a logic equivalence check is crucial in Serializer/Deserializer (SerDes) design to ensure that optimizations, modifications, or translations made during the design process do not alter the functionality of the circuit, including potential effects of clock gating. In a SerDes design, where high-speed and high-data-rate operations are paramount, even small errors can lead to significant issues in data integrity and throughput. The logic equivalence check, often done using tools like Conformal, compares the pre- and post-optimized designs to ensure that they function identically, even after clock gating is applied. This step is particularly critical in SerDes designs where even minor discrepancies can lead to serious performance or functional issues. By verifying the logic equivalence, designers can be confident that their optimizations and changes have not inadvertently affected the functionality of the SerDes circuit, ensuring its reliability and performance.



```
// Golden  key points = 361
// Revised key points = 379
// Mapping key points ...
================================================================================
Mapped points: SYSTEM class
--------------------------------------------------------------------------------
Mapped points     PI     PO     DFF       Total
--------------------------------------------------------------------------------
Golden            34     32     295       361
--------------------------------------------------------------------------------
Revised           34     32     295       361
================================================================================
Unmapped points:
================================================================================
Revised:
--------------------------------------------------------------------------------
Unmapped points   DLAT      Total
--------------------------------------------------------------------------------
Unreachable       18        18
================================================================================
// Command: add compared point -all
// 327 compared points added to compare list
// Command: compare
================================================================================
Compared points      PO     DFF       Total
--------------------------------------------------------------------------------
Equivalent           32     295       327
================================================================================
```

Fig 7.2 Logical Equivalence Check report

3. Gate Level Simulation:

   Gate-level simulation is critical for verifying the functionality and timing of a Serializer/Deserializer (SerDes) design after synthesis. It involves converting the design into actual gate-level representations to ensure that the synthesized gates behave as expected and that timing requirements are met. When simulating a SerDes design, using Standard Delay Format (SDF) delays is common. SDF captures delays through various elements of the design, like gates and wires, to accurately represent timing behavior. Simulating with SDF delays ensures that the SerDes design meets timing requirements and operates correctly under different conditions, crucial for high-speed and high-data-rate applications. Overall, gate-level simulation with SDF delays is essential for verifying SerDes designs, guaranteeing they meet performance standards and operate reliably.

4. Place and Route:

   In Serializer/Deserializer (SerDes) design, the place and route stage using the Innovus tool is a critical phase in translating the synthesized netlist into a physical layout that meets performance, power, and area requirements. Innovus takes inputs such as the synthesized netlist, detailed timing constraints, libraries, and technology files to guide the placement and routing process. During placement, Innovus positions the standard cells on the chip floorplan to optimize for factors like wire length, signal integrity, and timing. Routing then connects these cells using metal layers according to specified rules, considering signal integrity and power consumption. Timing closure is a key focus, where Innovus iteratively refines the placement and routing to meet setup and hold times specified in the constraints. Techniques like buffer insertion and wire sizing are employed to optimize timing. Overall, Innovus' place and route process is crucial for ensuring that the SerDes design functions correctly and efficiently in silicon, meeting the stringent requirements of high-speed communication systems.

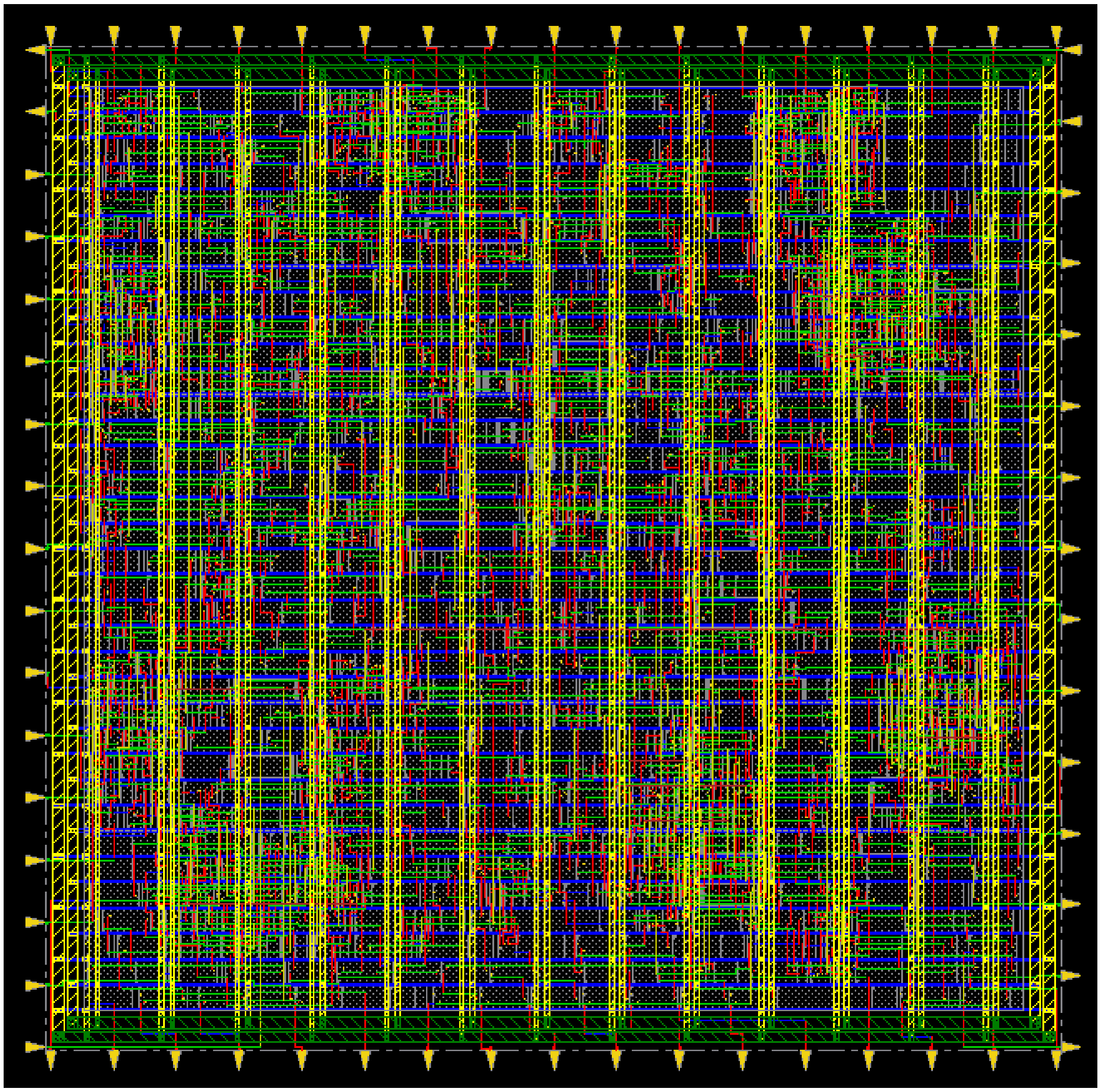


Fig 7.3 Place and Route Nets snapshot

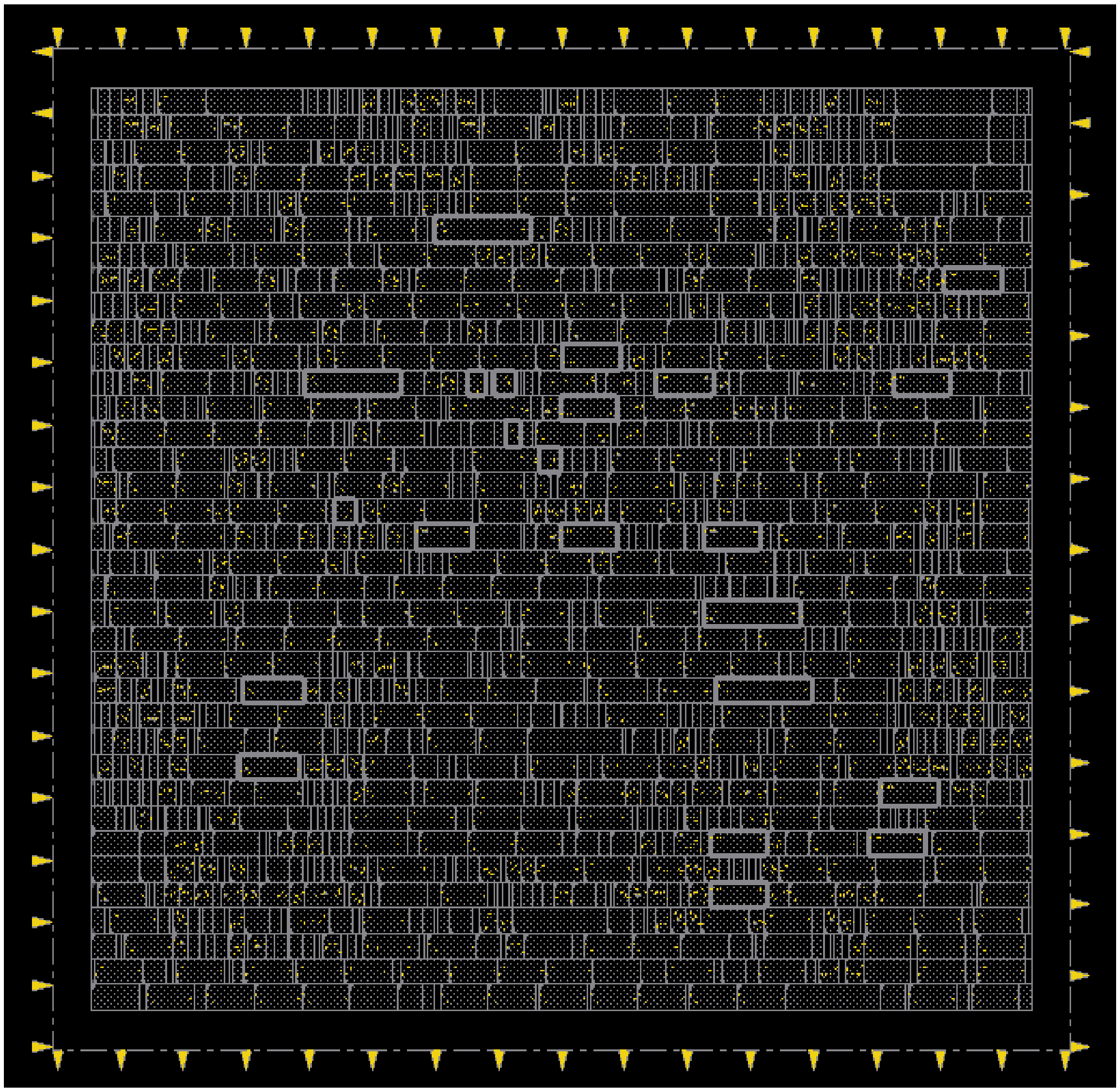


Fig 7.4 Place and Route Cells snapshot

5. Timing Analysis and Fix:

   In Serializer/Deserializer (SerDes) design, achieving timing closure while adhering to Design Rule Checks (DRCs) is a complex process that often requires an iterative approach between timing analysis and physical implementation tools like Tempus and Innovus, respectively. Tempus identifies timing violations and suggests fixes, such as buffer insertion or cell replacement, to improve timing slack. Innovus then implements these fixes in the physical layout while ensuring that DRCs are not violated. This iterative process continues until timing closure is achieved, meaning that all timing requirements are met and DRCs are satisfied. The iterative nature of this process allows designers to refine the design and optimize timing while ensuring that the final layout is manufacturable and meets all design constraints.

		Initial	**Final**
Hold	WNS	-0.134	0.0
	TNS	-24.314	0.0
	Total violating paths	306	0
Setup	WNS	0.398	0.012
	TNS	0.0	0.0
	Total violating paths	0	0

Table 7.1 Comparison of Pre and Post timing optimization.

8. Results and Discussions

8.1 Simulation results

This section presents the simulation outputs of all the individual components, i.e Encoder, PISO, SIPO, Decoder as well as the SerDes module, which is implemented by combining all of these sub-modules.

1. Encoder: Through simulation, it was verified that the encoder module was successful in encoding the 8b bit input data into 10 bit data.

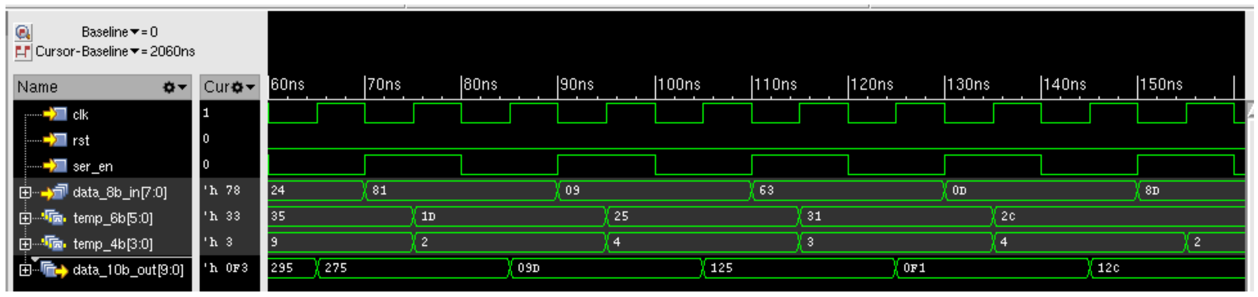

Baseline = 0
Cursor-Baseline = 2060ns
Name
Cur
60ns 70ns 80ns 90ns 100ns 110ns 120ns 130ns 140ns 150ns
clk 1
rst 0
ser_en 0
data_8b_in[7:0] 'h 78 — 24, 81, 09, 63, 0D, 8D
temp_6b[5:0] 'h 33 — 35, 1D, 25, 31, 2C
temp_4b[3:0] 'h 3 — 9, 2, 4, 3, 4, 2
data_10b_out[9:0] 'h 0F3 — 295, 275, 09D, 125, 0F1, 12C

Fig 8.1.1 Encoder simulation

2. PISO: Through simulation, it was verified that the PISO was able to serialize the input parallel data.

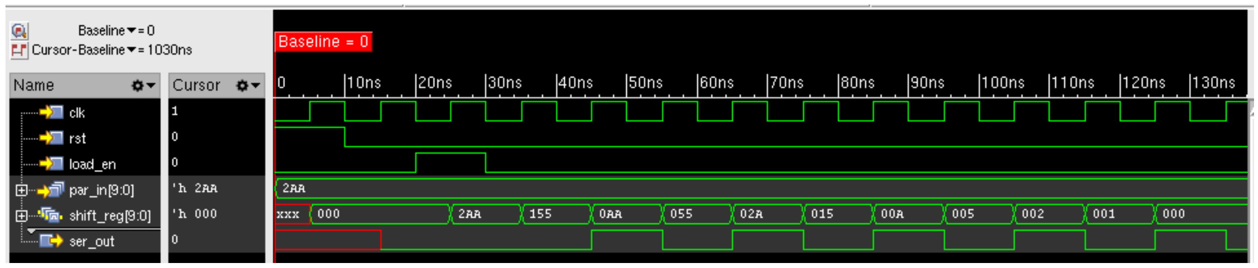

Baseline = 0
Cursor-Baseline = 1030ns
Baseline = 0
Name
Cursor
0 10ns 20ns 30ns 40ns 50ns 60ns 70ns 80ns 90ns 100ns 110ns 120ns 130ns
clk 1
rst 0
load_en 0
par_in[9:0] 'h 2AA — 2AA
shift_reg[9:0] 'h 000 — xxx, 000, 2AA, 155, 0AA, 055, 02A, 015, 00A, 005, 002, 001, 000
ser_out 0

Fig 8.1.2 PISO simulation

3. SIPO: Through simulation, it was verified that the SIPO was able to deserialize the input data and produce a 10 bit parallel data out.

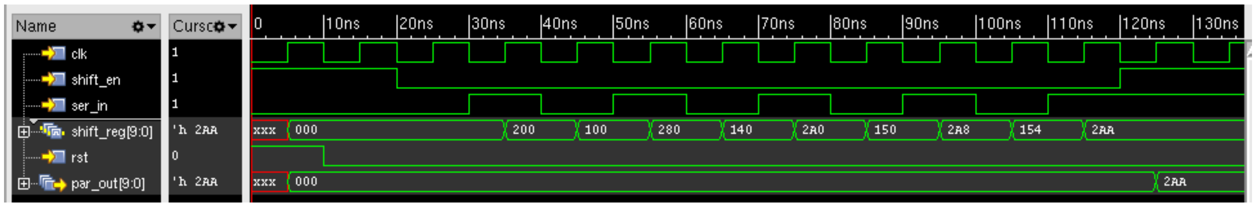


Fig 8.1.3 SIPO simulation

4. Decoder: Through simulation it was verified that the decoder was able to decode the input 10 bit data into 8 bit data out.

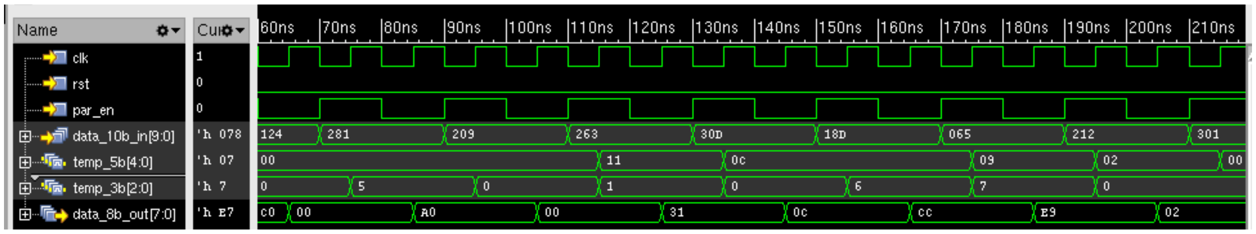


Fig 8.1.4 Decoder simulation

5. SerDes: After integrating all the components in a top module along with the FSM controlling it the following simulation results were obtained.

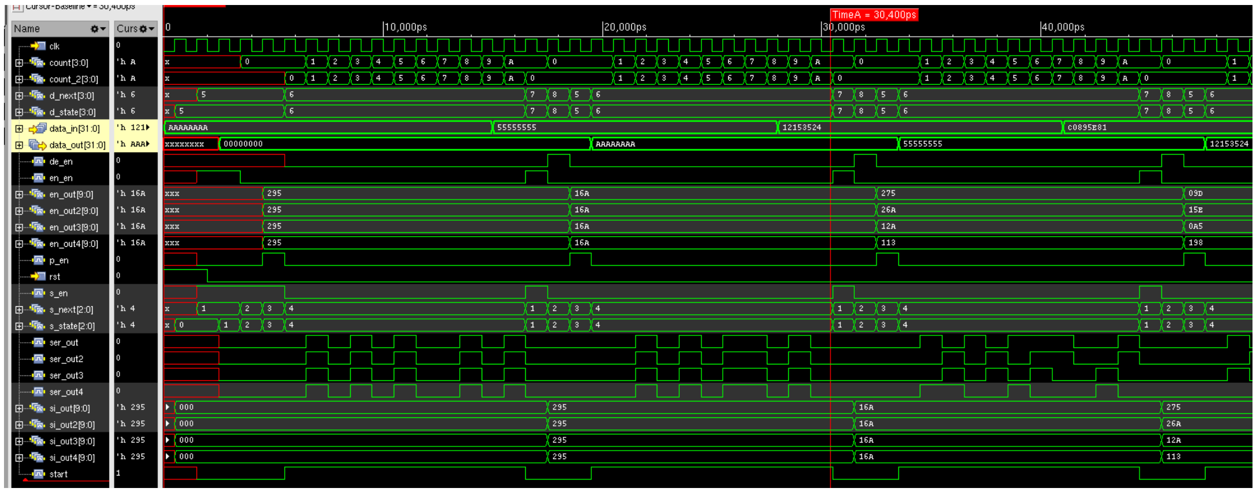


Fig 8.1.5 SerDes simulation

The operating clock speed of the SerDes circuit was set to be 1GHz (time period 1ns). The SeDes was designed in such a way that it takes 17 clock cycles for the data to be transmitted across after the reset occurs. We have also implemented a basic pipelined structure in design where the next data that is supposed to be sampled at input and given to PISO is not sampled after the first data's transmission ends at 17 clock cycles, it is sampled just after 14 clock cycles. This is achieved by leveraging the down time of the encoder and decoder. The encoder was supposed to sample data after the current data transmission ends but that is a lot of downtime for the encoder instead we sample data a few clock cycles earlier when the PISO is still serializing the data. On the other end SIPO need to wait for the decoder to complete its operation and start sampling (n+1)th data right after it gives out the nth data output.

The input need only change every 13 clock cycles i.e 13ns for it to be successfully sampled and transmitted, meaning the rest of the chip that is providing data can operate at a lower speed saving power. On the receiver side it looks like the data is received every 14 clock cycles.

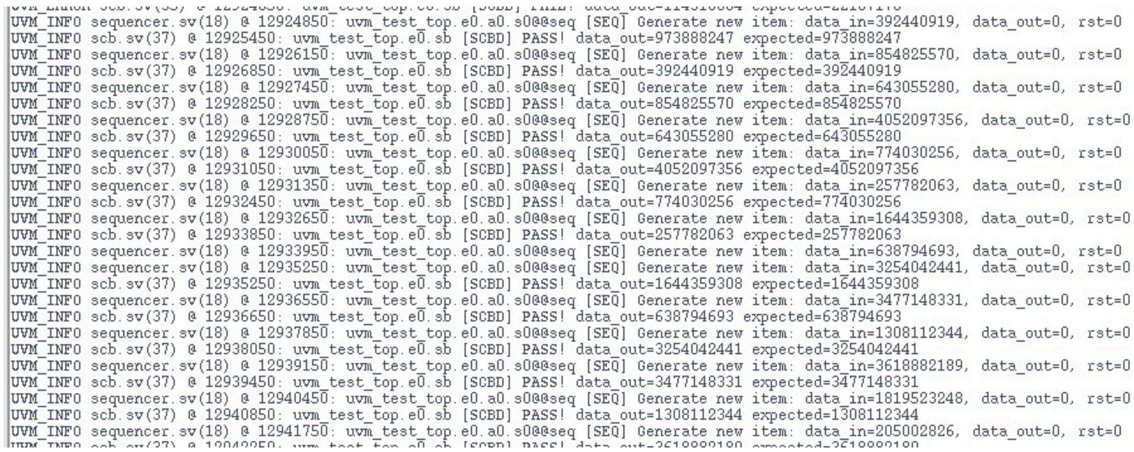
Calculating the data rate based on all the parameters, we have:
T = 1ns
Data transferred = 8 bits
Clock cycles required to transmit data = 14

Data rate = Data transferred / Time taken
= 8 / (14 x 1ns)
= 571.428 Mbps

Our design has four such serial lanes
Total Data Rate = 571.428 Mbps x 4 = 2.285 Gbps
The SerDes designed in this project is a full duplex SerDes, which means that it can serialize and deserialize at the same time. Hence, the throughput of the SerDes circuit is 4.57 Gbps.

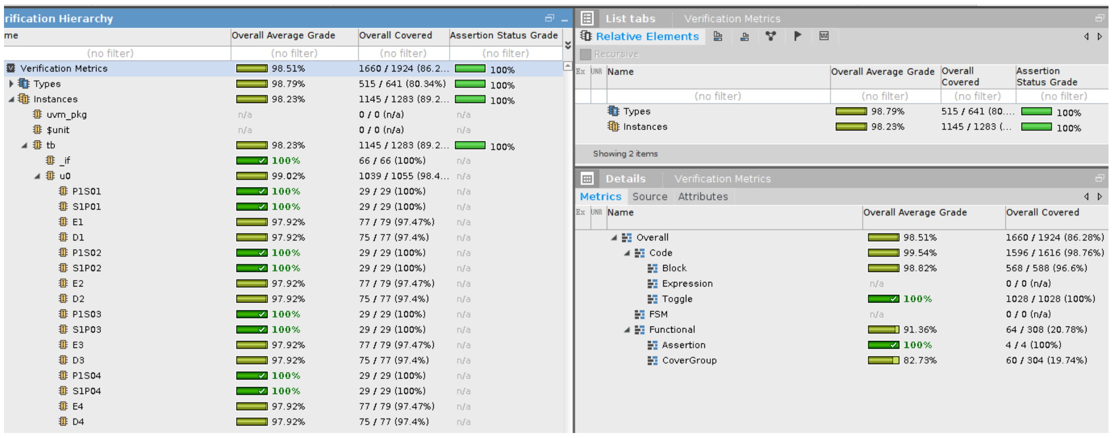
8.2 Verification results

The verification efforts of this project were mainly driven by simulation and coverage. UVM scoreboard implemented to verify the functionality of the SerDes, using a FIFO to store and compare the data_in and data_out. With the help of the scoreboard the checking was automated and it was made sure that the input data was successfully serialized , sent and deserialized on the receiver side without failure and in the same order of input.



```
UVM_INFO sequencer.sv(18) @ 12924850: uvm_test_top.e0.a0.s0@@seq [SEQ] Generate new item: data_in=392440919, data_out=0, rst=0
UVM_INFO scb.sv(37) @ 12925450: uvm_test_top.e0.sb [SCBD] PASS! data_out=973888247 expected=973888247
UVM_INFO sequencer.sv(18) @ 12926150: uvm_test_top.e0.a0.s0@@seq [SEQ] Generate new item: data_in=854825570, data_out=0, rst=0
UVM_INFO scb.sv(37) @ 12926850: uvm_test_top.e0.sb [SCBD] PASS! data_out=392440919 expected=392440919
UVM_INFO sequencer.sv(18) @ 12927450: uvm_test_top.e0.a0.s0@@seq [SEQ] Generate new item: data_in=643055280, data_out=0, rst=0
UVM_INFO scb.sv(37) @ 12928250: uvm_test_top.e0.sb [SCBD] PASS! data_out=854825570 expected=854825570
UVM_INFO sequencer.sv(18) @ 12928750: uvm_test_top.e0.a0.s0@@seq [SEQ] Generate new item: data_in=4052097356, data_out=0, rst=0
UVM_INFO scb.sv(37) @ 12929650: uvm_test_top.e0.sb [SCBD] PASS! data_out=643055280 expected=643055280
UVM_INFO sequencer.sv(18) @ 12930050: uvm_test_top.e0.a0.s0@@seq [SEQ] Generate new item: data_in=774030256, data_out=0, rst=0
UVM_INFO scb.sv(37) @ 12931050: uvm_test_top.e0.sb [SCBD] PASS! data_out=4052097356 expected=4052097356
UVM_INFO sequencer.sv(18) @ 12931350: uvm_test_top.e0.a0.s0@@seq [SEQ] Generate new item: data_in=257782063, data_out=0, rst=0
UVM_INFO scb.sv(37) @ 12932450: uvm_test_top.e0.sb [SCBD] PASS! data_out=774030256 expected=774030256
UVM_INFO sequencer.sv(18) @ 12932650: uvm_test_top.e0.a0.s0@@seq [SEQ] Generate new item: data_in=1644359308, data_out=0, rst=0
UVM_INFO scb.sv(37) @ 12933850: uvm_test_top.e0.sb [SCBD] PASS! data_out=257782063 expected=257782063
UVM_INFO sequencer.sv(18) @ 12933950: uvm_test_top.e0.a0.s0@@seq [SEQ] Generate new item: data_in=638794693, data_out=0, rst=0
UVM_INFO sequencer.sv(18) @ 12935250: uvm_test_top.e0.a0.s0@@seq [SEQ] Generate new item: data_in=3254042441, data_out=0, rst=0
UVM_INFO scb.sv(37) @ 12935250: uvm_test_top.e0.sb [SCBD] PASS! data_out=1644359308 expected=1644359308
UVM_INFO sequencer.sv(18) @ 12936550: uvm_test_top.e0.a0.s0@@seq [SEQ] Generate new item: data_in=3477148331, data_out=0, rst=0
UVM_INFO scb.sv(37) @ 12936650: uvm_test_top.e0.sb [SCBD] PASS! data_out=638794693 expected=638794693
UVM_INFO sequencer.sv(18) @ 12937850: uvm_test_top.e0.a0.s0@@seq [SEQ] Generate new item: data_in=1308112344, data_out=0, rst=0
UVM_INFO scb.sv(37) @ 12938050: uvm_test_top.e0.sb [SCBD] PASS! data_out=3254042441 expected=3254042441
UVM_INFO sequencer.sv(18) @ 12939150: uvm_test_top.e0.a0.s0@@seq [SEQ] Generate new item: data_in=3618882189, data_out=0, rst=0
UVM_INFO scb.sv(37) @ 12939450: uvm_test_top.e0.sb [SCBD] PASS! data_out=3477148331 expected=3477148331
UVM_INFO sequencer.sv(18) @ 12940450: uvm_test_top.e0.a0.s0@@seq [SEQ] Generate new item: data_in=1819523248, data_out=0, rst=0
UVM_INFO scb.sv(37) @ 12940850: uvm_test_top.e0.sb [SCBD] PASS! data_out=1308112344 expected=1308112344
UVM_INFO sequencer.sv(18) @ 12941750: uvm_test_top.e0.a0.s0@@seq [SEQ] Generate new item: data_in=205002826, data_out=0, rst=0
```

Fig 8.2.1 Scoreboard output



rification Hierarchy

me	Overall Average Grade	Overall Covered	Assertion Status Grade
(no filter)	(no filter)	(no filter)	(no filter)
Verification Metrics	98.51%	1660 / 1924 (86.2...	100%
Types	98.79%	515 / 641 (80.34%)	100%
Instances	98.23%	1145 / 1283 (89.2...	100%
uvm_pkg	n/a	0 / 0 (n/a)	n/a
$unit	n/a	0 / 0 (n/a)	n/a
tb	98.23%	1145 / 1283 (89.2...	100%
_if	✓ 100%	66 / 66 (100%)	n/a
u0	99.02%	1039 / 1055 (98.4...	n/a
P1S01	✓ 100%	29 / 29 (100%)	n/a
S1P01	✓ 100%	29 / 29 (100%)	n/a
E1	97.92%	77 / 79 (97.47%)	n/a
D1	97.92%	75 / 77 (97.4%)	n/a
P1S02	✓ 100%	29 / 29 (100%)	n/a
S1P02	✓ 100%	29 / 29 (100%)	n/a
E2	97.92%	77 / 79 (97.47%)	n/a
D2	97.92%	75 / 77 (97.4%)	n/a
P1S03	✓ 100%	29 / 29 (100%)	n/a
S1P03	✓ 100%	29 / 29 (100%)	n/a
E3	97.92%	77 / 79 (97.47%)	n/a
D3	97.92%	75 / 77 (97.4%)	n/a
P1S04	✓ 100%	29 / 29 (100%)	n/a
S1P04	✓ 100%	29 / 29 (100%)	n/a
E4	97.92%	77 / 79 (97.47%)	n/a
D4	97.92%	75 / 77 (97.4%)	n/a

List tabs — Verification Metrics — Relative Elements — Recursive

Name	Overall Average Grade	Overall Covered	Assertion Status Grade
(no filter)	(no filter)	(no filter)	(no filter)
Types	98.79%	515 / 641 (80....	100%
Instances	98.23%	1145 / 1283 (...	100%

Showing 2 items

Details — Verification Metrics — Metrics Source Attributes

Name	Overall Average Grade	Overall Covered
Overall	98.51%	1660 / 1924 (86.28%)
Code	99.54%	1596 / 1616 (98.76%)
Block	98.82%	568 / 588 (96.6%)
Expression	n/a	0 / 0 (n/a)
Toggle	✓ 100%	1028 / 1028 (100%)
FSM	n/a	0 / 0 (n/a)
Functional	91.36%	64 / 308 (20.78%)
Assertion	✓ 100%	4 / 4 (100%)
CoverGroup	82.73%	60 / 304 (19.74%)

Fig 8.2.2 IMC snapshot

The above image shows the snapshot of the coverage report obtained through the Cadence tool IMC. Through simulation and coverage collection we were able to obtain an overall average coverage grade of 98.51%. Coverage grading is an option used to rank the test cases based on the number of functional coverage points hit by the individual test case. The code coverage of the design was 98.23%, the drop can be attributed to the default cases present in the encoders and decoders which did not get exercised. Toggle coverage was 100% as the randomized test bench was able to toggle all the signals in the design.

Coming to the functional coverage report we obtained a grade of 91.36% meaning most of the covergroups were exercised and the ones which were not exercised can be attributed to the cross coverage bins of the FSM states. The overall covered part of the functional coverage is about 20%. This is a low score which can be attributed to the complex design where the input can have 2^32 (4G) total input vectors. Generating this huge amount of inputs and applying them to the design can take up days of simulation time. Part of why the coverage grade is high can be attributed towards the less number of cover groups and assertions defined. With well defined covergroups and a lot of assertions the design can further be verified thoroughly but this will take days of simulation time which is currently out of scope for this project.

8.3 Synthesis results

The SeDes design was synthesized using the Genus tool with the 45nm fast_vdd1v0_basicCells library, made available on the Cadence portal. Synthesis was carried for multiple iterations with different clock speed and techniques. The clock frequency used for the final iteration of synthesis was 1GHz with clock gating. The 'syn_gen' and 'syn_map' efforts were set high. After the synthesis Netlist file (.v), updated constraints file (.sdc), delays file (.sdf) and the area, power and timing reports were obtained.

Note: 'syn_opt' was not used as it was found that the tool optimized and removed parts of circuit leading to complete failure of circuit.

Design	Clock time period (ns)	Area (um^2)	Power(W)	Timing slack
SerDes	5	3969.936	0.262835	3724
SerDes with clock gating		3538.647	0.179580	3856
SerDes	1	3939.156	1.31163	678
SerDes with clock gating		3502.764	0.908443	733

Table 8.3.1 Comparison of Area, Power and Timing Slack after Synthesis

Initially the clock period was chosen to be 5ns (frequency 200MHz). 2 cases were tried out one with clock gating and another without clock gating. The clock gated circuit proved to be better in terms of power consumption as well as an improvement in timing slack by a small margin. Later on in the design data rate and throughput were prioritized leading to a reduced clock time period of 1ns (frequency 1GHz). Again two cases were tried w.r.t clock gating and the same trends in power and area were observed. Overall power increased when compared to circuit with 5ns clock, which is a result of power being directly proportional to frequency.

8.4 Gate level netlist simulation

After the circuit has been synthesized the generated netlist should be simulated and verified while applying the delays obtained through the synthesis step. It is worth noting that even though the golden reference and netlist were found to be equivalent by the tool, the gate level netlist simulation proves otherwise. It was found that only alternate data_in were successfully sent through the serializer and received on the other end. We are not sure why this odd behavior is but we think it has to do with the optimizations done by the synthesis tool. This issue is to be further looked into , analyzed and debugged.

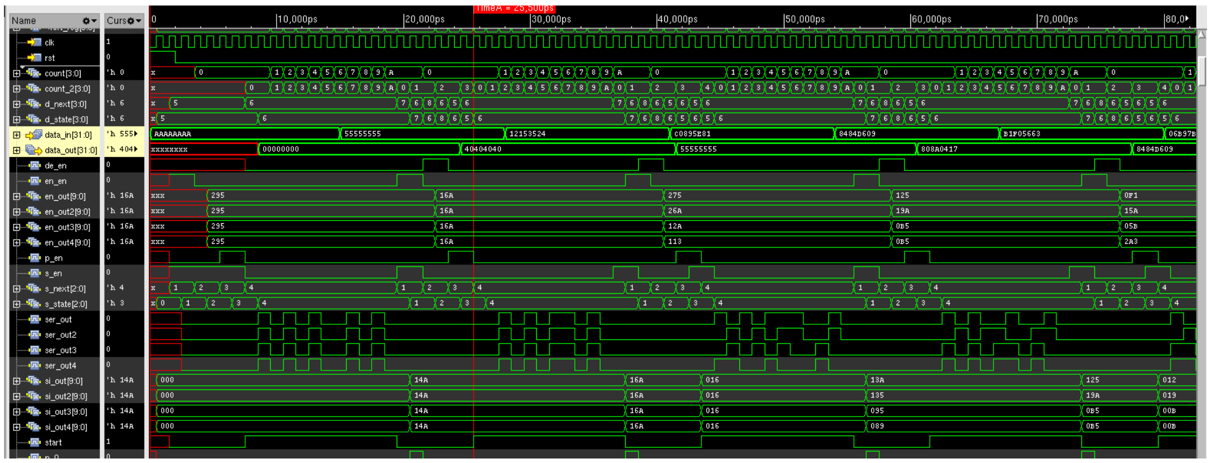


Fig 8.4.1 SerDes netlist simulation

As a work around for this issue we propose that the data_in and data_out sampling frequency be reduced, essentially extending the data_in for twice as long and sampling every other data_out value. This would unfortunately reduce the operation frequency and the usable speed of the SerDes by half.

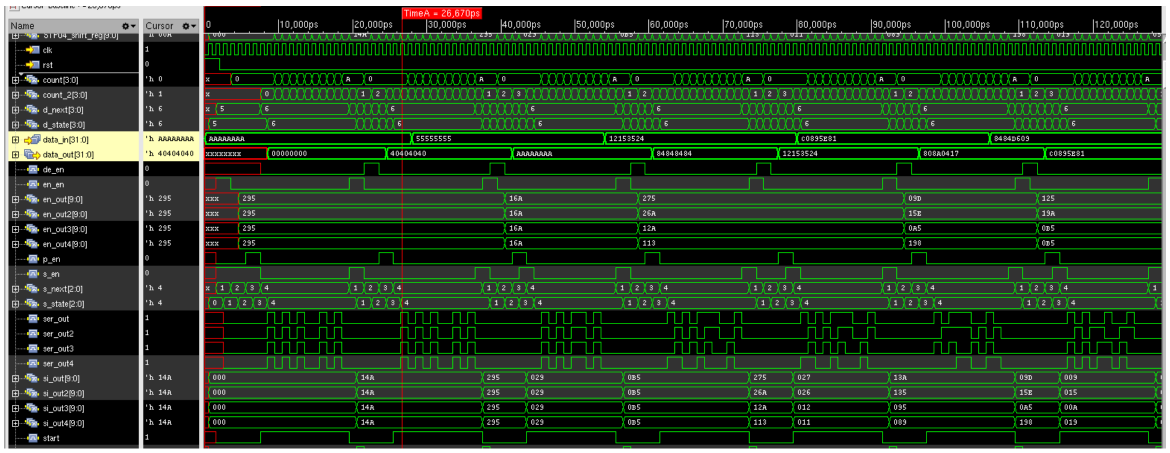


Fig 8.4.2 Simulation of work around for data loss

8.5 Placement and Routing results

After placement and routing the updated area timing and power statistics were obtained and are analyzed in this section. Similar to the synthesis step, place and route were carried out for 2 cases of the synthesized netlist. One with clock gating and another without, both the circuits used 1 GHz clock frequency.

		Without Clock gating	**With Clock gating**
Area (um^2)	Core	7609.500	6746.976
	Chip	8555.244	7638.760
Power (mW)	Internal	1.06541531	0.74708269
	Switching	0.50858643	0.32796183
	Leakage	0.00014243	0.00008084
	Total	1.57414419	1.07512535
Timing slack (ns)	Setup	0.056	0.012
	Hold	0.002	0.001

Table 8.5.1 Comparison of Area, Power and Timing slack after Place and Route

The results obtained after place and route showcase the same trend we observed in the synthesis results where clock gated circuit proves to be more area and power efficient. We observe a 10% decrease in the chip area in the clock gated SerDes vs the non clock gated SerDes, this is because clock gating removes large numbers of muxes and replaces them with clock gating logic. We observe that timing slack is a bit better in case of non clock gated circuit, but the difference can be overlooked as both the circuits meet the timing with 0 setup/hold violations.

The bigger difference was observed in terms of power, where clock gating resulted in about 30% reduction in internal power of the circuit, about 35% reduction in switching power, and about 43% reduction in leakage power. The total power consumed by the chip is reduced by about 31%. When compared with the power results we obtained from the synthesis step, it can be seen that the power consumed has increased, this is due to the addition of a huge number of buffers into the design to eliminate the timing violations. Overall we can say that clock gating helped in the reduction of power consumption enabling the chip to scale up in frequency increasing performance.

9. Conclusion

This project successfully designed, verified, and synthesized a high-speed Serializer/Deserializer (SerDes) system, focusing on achieving robust functionality and optimal performance using a 45 nm library. The project followed a systematic workflow, beginning with design and verification using advanced methodologies such as the Universal Verification Methodology (UVM), and culminating in synthesis and implementation targeting high-speed operation.

The design phase of the project emphasized the creation of a high-speed SerDes architecture capable of reliably transmitting data at ultra-high speeds. By implementing advanced techniques such as parallel processing and efficient data encoding, the design achieved a throughput of 4.57 Gb. The verification process was meticulous, leveraging UVM to ensure functional correctness and compliance with specifications. Coverage metrics and assertions were extensively utilized to validate the design and ensure its reliability under various operating conditions.

During synthesis and place and route using the 45 nm library, the design was optimized for high-speed operation, with careful consideration given to timing constraints and signal integrity. Advanced techniques such as clock gating and pipelining were employed to maximize performance while minimizing power consumption and area utilization.

The placement and routing stages were critical for achieving high-speed operation, requiring careful attention to signal routing, timing closure, and power distribution. Through iterative refinement, the design was optimized to fit within an area of 7638.760 um^2, while maintaining high-speed performance.

In conclusion, this project successfully designed, verified, and synthesized a high-speed SerDes system, showcasing the benefits of advanced design methodologies and optimization techniques using a 45 nm library. The high-speed SerDes demonstrated robust functionality and optimal performance, highlighting the importance of careful design and verification in achieving high-speed operation within a constrained area. Future work could explore further optimizations and innovations to push the boundaries of high-speed SerDes technology.

10. References

1. Hwang, H.; Kim, J. A 100 Gb/s Quad-Lane SerDes Receiver with a PI-Based Quarter-Rate All-Digital CDR. Electronics 2020, 9, 1113. https://doi.org/10.3390/electronics9071113
2. S. Samanta and A. Dastidar, "Implementation of 10bit SerDes for Gigabit Ethernet PHY," 2015 International Conference on Man and Machine Interfacing (MAMI), Bhubaneswar, India, 2015, pp. 1-5, doi: 10.1109/MAMI.2015.7456601.
3. S. Han, T. Kim, J. Kim and J. Kim, "A 10 Gbps SerDes for wireless chip-to-chip communication," 2015 International SoC Design Conference (ISOCC), Gyeongju, Korea (South), 2015, pp. 17-18, doi: 10.1109/ISOCC.2015.7401630.
4. J. M. T. R. Jayawickrama and S. Thayaparan, "Use of SerDes to Reduce the Cost of Packaging of VLSIs," Aug. 25, 2023. doi: 10.1109/ICIIS58898.2023.10253611

5. Maadi, Mohammad. (2015). An 8b/10b Encoding Serializer/Deserializer (SerDes) Circuit for High Speed Communication Applications Using a DC Balanced, Partitioned-Block, 8b/10b T. International Journal of Electronics and Electrical Engineering. 3. 144. 10.12720/ijeee.3.2.144-148.

6. P. -C. Pan, H. -H. Cheng and C. -C. Wang, "High speed SerDes design verification," 2014 9th International Microsystems, Packaging, Assembly and Circuits Technology Conference (IMPACT), Taipei, Taiwan, 2014, pp. 226-229, doi: 10.1109/IMPACT.2014.7048438.

7. W. Xie, G. Cao and W. Ji, "Research on High-speed SerDes Interface Testing Technology," 2021 22nd International Conference on Electronic Packaging Technology (ICEPT), Xiamen, China, 2021, pp. 1-5, doi: 10.1109/ICEPT52650.2021.9567964.

8. Nagesh, K.A., Shilpa, D.R. (2021). Verification of SerDes Design Using UVM Methodology. In: Nath, V., Mandal, J.K. (eds) Proceeding of Fifth International Conference on Microelectronics, Computing and Communication Systems. Lecture Notes in Electrical Engineering, vol 748. Springer, Singapore. https://doi.org/10.1007/978-981-16-0275-7_49

9. A. M. Sawaby et al., "A 10 Gb/s SerDes Transceiver," 2021 3rd Novel Intelligent and Leading Emerging Sciences Conference (NILES), Giza, Egypt, 2021, pp. 389-393, doi: 10.1109/NILES53778.2021.9600520.

10. D. Park, J. Yoon and J. Kim, "A low-power SerDes for high-speed on-chip networks," 2017 International SoC Design Conference (ISOCC), Seoul, Korea (South), 2017, pp. 252-253, doi: 10.1109/ISOCC.2017.8368879.

11. X. Zheng et al., "A 40-Gb/s Quarter-Rate SerDes Transmitter and Receiver Chipset in 65-nm CMOS," in IEEE Journal of Solid-State Circuits, vol. 52, no. 11, pp. 2963-2978, Nov. 2017, doi: 10.1109/JSSC.2017.2746672

12. R. Wenzel, T. Zhou and S. Karako, "Flip-chip package for 28G SerDes interface," 2016 IEEE 25th Conference on Electrical Performance Of Electronic Packaging And Systems (EPEPS), San Diego, CA, USA, 2016, pp. 11-14, doi: 10.1109/EPEPS.2016.7835407.

13. D. Park, J. Yoon and J. Kim, "A low-power SerDes for high-speed on-chip networks," 2017 International SoC Design Conference (ISOCC), Seoul, Korea (South), 2017, pp. 252-253, doi: 10.1109/ISOCC.2017.8368879.

14. J. Lee, P. -C. Chiang, P. -J. Peng, L. -Y. Chen and C. -C. Weng, "Design of 56 Gb/s NRZ and PAM4 SerDes Transceivers in CMOS Technologies," in IEEE Journal of Solid-State Circuits, vol. 50, no. 9, pp. 2061-2073, Sept. 2015, doi: 10.1109/JSSC.2015.2433269