A Project Report
on

IMPLEMENTATION AND VERIFICATION OF SERIALIZER-DESERIALIZER FOR DIGITAL COMMUNICATION USING 90NM CMOS TECHNOLOGY

at

Dharmsinh Desai University

by

Shah Harsh Jatin Kumar
EC090, 21ECOUS026

B.Tech. (EC) Semester - VIII

Faculty Supervisor | **Guide**

Prof. Manish K. Patel | **Dr. Pallavi G. Darji**

In Partial Fulfillment of Requirement of

Bachelor of Technology - Electronics & Communication

Submitted To

Department of Electronics & Communication Engineering
Faculty of Technology
Dharmsinh Desai University,
Nadiad - 387001.

(April 2025)

ACKNOWLEDGEMENT

I extend my heartfelt gratitude to Prof. Pallavi Darji, whose unwavering support and expertise have been pivotal to the successful completion of this project. Prof. Pallavi's guidance, insightful perspectives, and encouragement have not only shaped the trajectory of my research but have also played a crucial role in enhancing my understanding of the subject matter. I am fortunate to have had the opportunity to work under her mentorship, and I am truly appreciative of the knowledge and skills she has imparted002E

I would like to express my sincere appreciation to Prof. Manish Patel, my esteemed faculty supervisor, for his invaluable contributions to this project. Prof. Manish's meticulous attention to detail, constructive feedback, and dedication to academic excellence have significantly influenced the refinement of my work. His mentorship has been a source of inspiration, motivating me to strive for excellence in my academic pursuits.

I am grateful to the entire faculty at Dharmsinh Desai University for fostering an intellectually stimulating environment that encourages academic exploration and creativity. The support and resources provided by the university have been instrumental in the successful execution of this project.

I also extend my thanks to my fellow students and colleagues for their collaborative spirit and shared enthusiasm throughout this academic endeavour. The exchange of ideas and collective effort has contributed to the overall growth and development of this project.

In conclusion, I express my deepest appreciation to everyone who has been a part of this journey, contributing to the realization of this project. Your support has been invaluable, and I am truly grateful for the enriching learning experience provided by the academic community at Dharmsinh Desai University.

Harsh Shah

(21ECUOS026, EC090)

SYNOPSIS

This project focuses on Serializer-Deserializer (SerDes) system, which is used to convert data from parallel to serial and form serial to parallel. Use of such system reduces number of pins on chip and can also optimize the bandwidth capacity. The project begins with a theoretical exploration of SerDes principles, covering fundamental topics such as serialization, deserialization, clock data recovery (CDR), encoding-decoding, clock jitter, and synchronization. this thesis focusses on the clocking mechanism based Serdes architecture, The different SerDes architectures have been studied and among them Parallel Clock SerDes and Embedded Clock SerDes architectures have been implemented. Both architectures are implemented using Verilog and simulated using Cadence EDA tools to ensure precise verification and accurate performance assessment. The Parallel Clock SerDes utilizes a dedicated clock signal for synchronization, ensuring stable timing and reduced clock jitter, but requiring additional clock routing resources, which increases design complexity. On the other hand, the Embedded Clock SerDes embeds the clock signal within the data stream, eliminating the need for a separate clock signal and enhancing system scalability. However, this design presents additional challenges, particularly in terms of clock jitter and clock-data alignment, which significantly affect signal integrity. The performance of both architectures was assessed based on critical parameters, including clock jitter percentage, baud rate, bit error rate (BER), and clock-data alignment.

Simulation based results demonstrate that the Parallel Clock SerDes achieves a jitter rate of 3.5%, making it a more stable solution in terms of timing accuracy, while the Embedded Clock SerDes initially exhibits a higher jitter rate of 6.68%, indicating greater timing uncertainty. Given that high jitter can lead to increased bit errors and reduced data transmission reliability, modifying the Embedded Clock SerDes for better performance is essential. To overcome that problem, we implemented a Numerically Controlled Oscillator (NCO) to enhance clock stability and reduce jitter in the Embedded Clock SerDes. The NCO dynamically adjusts the clock frequency based on phase variations, effectively reducing timing deviations and improving data integrity. After integrating the NCO, the jitter rate of the Embedded Clock SerDes is reduced from 6.68% to 5.6%, marking a significant improvement in signal stability and overall transmission reliability. This result highlights the effectiveness of jitter compensation techniques in high-speed serial communication systems. The findings of this study provide valuable insights into the trade-offs between different

SerDes architectures, particularly in the context of clocking methodologies and their impact on jitter performance. While Parallel Clock SerDes offers lower jitter and better timing accuracy, it requires an external clock signal, increasing circuit complexity and design overhead. Conversely, Embedded Clock SerDes, although initially prone to higher jitter, benefits from clock embedding, simplifying interconnects and enabling more scalable high-speed data transmission. The successful implementation of the NCO in the Embedded Clock SerDes opens the potential of adaptive clock stabilization techniques in mitigating clock jitter effects, making this approach highly relevant for future advancements in SerDes technology.

Then after to implement the design, we have used the cadence EDA tools to simulate and verify the functionality of design and then we have performed physical design implementation of both embedded clock and parallel clock SerDes circuit using 90nm cmos technology we have calculated the physical parameters like area power and timing report of these modules we have used Innovus tool to perform the PNR flow and we have created the layout for these systems.

Table of content

Content
page no.

Acknowledgement **iii**

Synopsis **iv**

Table of content **vi**

List of Figure **ix**

List of Tables **xii**

Chapter 1 Project Definition and Specifications **1**

1.1 Design 1

1.2 Serializer system 1

1.3 Deserializer system 5

1.4 Versatility 8

1.5 applications 9

1.6 Types of SerDes Systems and Their Theoretical Explanation 9

Chapter 2 Parallel Clock Serdes **14**

2.1 Introduction 14

2.2 Implementation and Verification of Transmitter Block 15

2.3 Implementation and verification OF receiver block 18

Chapter 3 Embedded Clock Serdes **21**

3.1 Theory of Embedded Clock SerDes 21

3.2 Implementation and verification of transmitter using embedded clock serializer 22

3.3 8B/10B encoder & decoder 24

3.4 Implementation and verification of receiver using embedded clock deserializer 30

3.5 Clock Divider Logic-Based Clock Data Recovery (CDR) Module 32

3.6 Buffer Module 34

Chapter 4 Numerical Controlled Oscillator (Nco) System **36**

4.1 Design 36

4.2 Implementation and Verification of NCO Module 38

4.3 Implementation and verification of DAC circuit 40

4.4 Implementation and verification of digital sigma delta LPF 41

4.5 FPGA testing of NCO (Altera DE1) 43

4.6 Hybrid CDR 45

Chapter 5 Physical Design of transmitter and receiver circuits 48

5.1 introduction of ASIC design flow 48

5.2 ASIC implementation using cadence EDA tools. 49

5.3 Implementation of embedded clock SerDes based communication system 49

5.4 Implementation of parallel clock SerDes based communication system 65

Chapter 6 Observed Report and Performance Parameters 74

Conclusion 76

References 77

Appendix A Abbrevations 79

Appendix B Codes 80

Design Verilog Code 80

top_module_tx.v 80

top_module_rx.v 83

decodePipe.v 89

encoder_8b10.v 93

Parallel_clock_serializer.v 95

Parallel_clock_deserializer.v 97

NCO.v 99

Hybride_CDR.v 103

sigma_delta_dac.v 106

sigma_delta_filter.v 107

Testbench Codes 108

top_module_tx_tb.v 108

top_module_rx_tb.v 109

Parallel_Clock_Serializer_tb.v 110

tb_Parallel_Clock_Deserializer.v 111

NCO_tb.v 112
Hybrid_CDR_tb.v 113
tb_sigma_delta_dac.v 114
testbench_decodePipe.v 115
testbench_encoder_8b10.v 116

Appendix C Software setup 118

NC launch steps 118
Geneus steps 119
Innovus steps 120

LIST OF FIGURES

Fig. 1.1 Block Diagram Of Sterilizer 2
Fig. 1.2 8-Bit Serializer Module 4
Fig. 1.3 Simulation Of Serializer System 4
Fig. 1.4 Basic Block Diagram Of Deserializer System 5
Fig. 1.5 8-Bit Deserializer Block Diagram 8
Fig. 1.6 Simulation Of Deserializer 8
Fig. 2.1 Parallel Clock Serdes Based Transmitter Module 16
Fig. 2.2 Simulation Of Transmitter Module 17
Fig. 2.3 Digital Receiver System For Short Distance Communication 18
Fig. 2.4 Simulation Of Receiver Module 20
Fig. 3.1 Block Diagram Of Transmitter Module 22
Fig. 3.2 Implemented Block Diagram Of Transmitter Module 23
Fig. 3.3 Simulation Report Of Transmitter Module 24
Fig. 3.4 Converting First 5-Bits Into 6-Bit Symbols Using This Table 27
Fig. 3.5 Converting First 3-Bits Into 4-Bit Symbols Using This Table 27
Fig. 3.6 Controlling Signal 28
Fig. 3.7 8b/10b Encoding System 29
Fig. 3.8 Encoding Implementation And Verification 29
Fig. 3.9 10b/8b Decoding System 29
Fig. 3.10 Decoding Implementation And Verification 30
Fig. 3.11 Basic Block Diagram Of Receiver 31
Fig. 3.12 The Receiver Module 32
Fig. 3.13 Simulation Result Of Embedded Clock-Based Receiver 32
Fig. 3.14 Basic Block Diagram Of Theoretical Cdr Block 33
Fig. 3.15 Block Diagram Of Cdr 34
Fig. 3.16 Simulation Report Of Cdr Module 34
Fig. 3.17 Block Diagram Of Buffer Module 34
Fig. 3.18 Simulation Report Of Buffer Module 35
Fig. 4.1 Block Diagram Of Nco System 37
Fig. 4.2 Block Diagram Of Nco Module 38
Fig. 4.3 Output Of Nco For Random Frequency 39
Fig. 4.4 Signal Generation Of 100khz 39
Fig. 4.5 Signal Generation For 50khz 39
Fig. 4.6 Signal Generation For 1khz 40
Fig. 4.7 Block Diagram Of Dac Module 41
Fig. 4.8 Simulation Result Of Dac 41
Fig. 4.9 Block Diagram Of Filer 43
Fig. 4.10 Test Setup Of Fpga 43
Fig. 4.11 3.3khz Signal 44

Fig. 4.12 12.2khz Signal 44
Fig. 4.13 4.5khz Signal 44
Fig. 4.14 103.3hz Signal 44
Fig. 4.15 191.4khz Signal 44
Fig. 4.16 Block Diagram Of Hybride Cdr Module 46
Fig. 4.17 Output Of Hybrid Cdr Module 47
Fig. 4.18 Output Of Hybrid Cdr Module 47
Fig. 5.1 Asic Design Flow Block Diagram 49
Fig. 5.2 Simulation Output Of Transmitter Module 50
Fig. 5.3 Window Of Nc Sim 50
Fig. 5.4 Analyzer Window 50
Fig. 5.5 Timing Report Of Transmitter Module 51
Fig. 5.6 Area Report Of Transmitter Module 52
Fig. 5.7 Power Report Of Transmitter Module 52
Fig. 5.8 Constrain File Of Transmitter Module 53
Fig. 5.9 Gate Level Rtl Simulation Of Transmission 53
Fig. 5.10 First Floor Planning Report Of Transmitter Module 54
Fig. 5.11 Cts Optimized Output 54
Fig. 5.12 Cts Optimized Report 55
Fig. 5.13 Post Routing Report 56
Fig. 5.14 Post Routing And Sign Off Optimized Output 56
Fig. 5.15 Simulation Report Of Receiver Module 57
Fig. 5.16 Verification Coverage Output 57
Fig. 5.17 Timing Report Of Receiver 58
Fig. 5.18 Area Report Of Receiver Module 59
Fig. 5.19 Power Report Of Receiver Module 59
Fig. 5.20 Constrain File Of Receiver Module 60
Fig. 5.21 Rtl Synthesis Of Receiver Module 60
Fig. 5.22 Power And Floor Planning Of Receiver 61
Fig. 5.23 Cts Optimized Report Of Receiver Module 61
Fig. 5.24 Post Cts Report 62
Fig. 5.25 Post Routing Optimized Setup Report 63
Fig. 5.26 Post Routing Optimized Hold Report 63
Fig. 5.27 Post Routing Optimized Report 64
Fig. 5.28 Simulation Output Of Parallel Clock Transmitter Module 65
Fig. 5.29 Verification Coverage Report 65
Fig. 5.30 Area Report 66
Fig. 5.31 Power Report 66
Fig. 5.32 Timing Report 67
Fig. 5.33 Power And Floor Planning Report 67

Fig. 5.34 Post Cts Optimized Output 68
Fig. 5.35 Post Routing Output 69
Fig. 5.36 Simulation Report Of Parallel Clock Serdes Based Receiver 69
Fig. 5.37 Timing Report 70
Fig. 5.38 Power Report 70
Fig. 5.39 Area Report 71
Fig. 5.40 Power And Floor Planning Report 71
Fig. 5.41 Final Sign Off Report 72
Fig. 5.42 Final Layout Of Parallel Clock Serdes Based Receiver 73

LIST OF TABLES

Table 3.1 Output Comparison Of Encoder -Decoder 30
Table 6.1 Embedded Clock Serdes 74
Table 6.2 Parallel Clock Serdes 74

CHAPTER 1
PROJECT DEFINITION AND SPECIFICATIONS

In our delay life our processors and controllers are designed to prosses parallel data they are very good with pressing parallel data but when we have to transmit these data we need to convert this data in serial form we know that parallel processing is very effective ,but when we talk about transmitting the data we have to use serial communication is always very effective and feasible and for wireless communication it is always good .the goal is to create a system's which helps to convert data form serial- parallel and parallel form and helps data to communicate transmit and receive

1.1Design

The integration of SerDes system in digital communication is crucial this system is very important in serial communication systems, enabling efficient data transmission by converting parallel data into a serial stream and vice versa. In high-speed digital systems, parallel data transmission faces challenges such as skew, crosstalk, and excessive wiring, which limit performance and scalability. SerDes overcomes these issues by reducing the number of interconnections, minimizing signal integrity problems, and supporting long-distance data transmission with lower power consumption. It plays a vital role in applications such as PCIe, Ethernet, USB, SATA, and high-speed interconnects in FPGAs and SoCs. The serializer section compresses multiple parallel data lines into a high-speed serial data stream, reducing the number of physical channels required. At the receiving end, the deserializer reconstructs the original parallel data. Modern SerDes systems incorporate clock recovery, equalization, and error correction techniques to maintain signal quality and mitigate losses over long distances. In FPGA-based designs, SerDes enables high-bandwidth communication between chips, peripherals, and memory interfaces. It is also widely used in optical Fiber communication to achieve ultra-fast data rates with minimal latency. Overall, SerDes systems are indispensable for achieving high-speed, low-power, and reliable serial communication in advanced digital systems.

1.2Serializer system

A serializer system is an essential component in modern digital systems, enabling the

efficient transmission of data over high-speed serial communication channels. The primary function of a serializer is to convert parallel data, which consists of multiple bits transmitted simultaneously, into a single high-speed serial stream, reducing the number of interconnections required for data transfer. This conversion plays a critical role in various high-speed communication protocols such as PCIe, USB, Ethernet, HDMI, SATA, and optical fibre networks, where minimizing the number of signal lines and improving signal integrity is essential.

The fundamental working principle of a serializer system involves sequentially transmitting parallel data bits onto a single serial line. This is typically achieved using a combination of shift registers, multiplexers, and clocking mechanisms. When a parallel data word is input into the serializer, it is temporarily stored in a register. A clock signal, which determines the transmission speed, controls a shift register or a multiplexer that selects one bit at a time from the stored parallel word and sends it out sequentially on the serial output line. The process continues cyclically, ensuring that every parallel data word is transmitted serially within a given time frame. This mechanism allows data to be sent efficiently over long distances without the need for multiple transmission lines, significantly reducing complexity and power consumption.

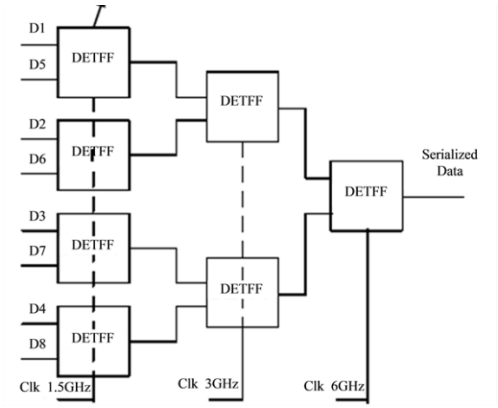


Fig. 1.1 Block Diagram of Sterilizer

The fig 1.1 depicts a multi-stage parallel-to-serial converter using D flip-flops (DETFF) operating at different clock frequencies (1.5 GHz, 3 GHz, and 6 GHz) to achieve data serialization.One of the most important components in a serializer system is the shift register. A shift register is a sequential logic system that shifts data bits in response to a clock signal. In a serializer, the shift register takes parallel data as input and shifts each bit out one by one in synchronization with the clock. Typically, a serializer employs either a parallel-in serial-out (PISO) shift register or a multiplexer-based approach to achieve serialization. The PISO

shift register receives a complete word of parallel data and then shifts out one bit per clock cycle, ensuring that the bits are transmitted in the correct order. The clock signal plays a crucial role in synchronizing the data transfer process, and a higher clock frequency results in a faster data transmission rate.

In some serializer designs, a multiplexer is used instead of a shift register to select one bit at a time from the parallel data inputs. A multiplexer-based serializer operates by sequentially selecting each bit of the input word and directing it to the serial output line. This selection is controlled by a counter or a control logic system that ensures the correct bit order is maintained. Although shift registers are more commonly used, multiplexer-based designs can be advantageous in certain applications where power efficiency and flexibility are critical.

Clock domain management is another essential aspect of serializer systems. Since serialization involves converting a wide parallel data bus into a high-speed serial stream, the clock signal must be carefully managed to ensure proper synchronization between the input data and the serialized output. Many serializer systems incorporate a phase-locked loop (PLL) or a delay-locked loop (DLL) to generate the high-speed clock signal required for serialization. A PLL is a feedback control system that adjusts the output clock phase and frequency to match the reference clock, ensuring stable and jitter-free data transmission. Jitter is an important factor to consider in serializer systems, as it can introduce errors in high-speed communication systems. Jitter refers to the small variations in timing between transmitted data bits, which can result from noise, interference, or clock instability. To mitigate jitter, serializer systems often include clock recovery and equalization techniques to maintain signal integrity.

Another critical aspect of serializer design is data encoding. Since high-speed serial transmission is susceptible to signal degradation, encoding schemes such as 8b/10b encoding or scrambling techniques are employed to ensure reliable data transmission. Encoding helps maintain DC balance, eliminate long runs of consecutive identical bits, and improve signal integrity by introducing transitions that facilitate clock recovery at the receiver end. In some cases, differential signaling techniques such as low-voltage differential signaling (LVDS) or current mode logic (CML) are used to reduce noise and improve immunity to electromagnetic interference (EMI).

Signal integrity is a major concern in serializer systems, particularly when transmitting data

at very high speeds. As data rates increase, issues such as attenuation, crosstalk, and inter-symbol interference (ISI) become more pronounced. To address these challenges, serializers often incorporate equalization techniques such as pre-emphasis and de-emphasis, which modify the transmitted signal to compensate for high-frequency losses in the transmission medium. Pre-emphasis boosts the high-frequency components of the signal at the transmitter, while de-emphasis reduces the amplitude of consecutive identical bits to minimize inter-symbol interference.

Implementation and verification of serializer system

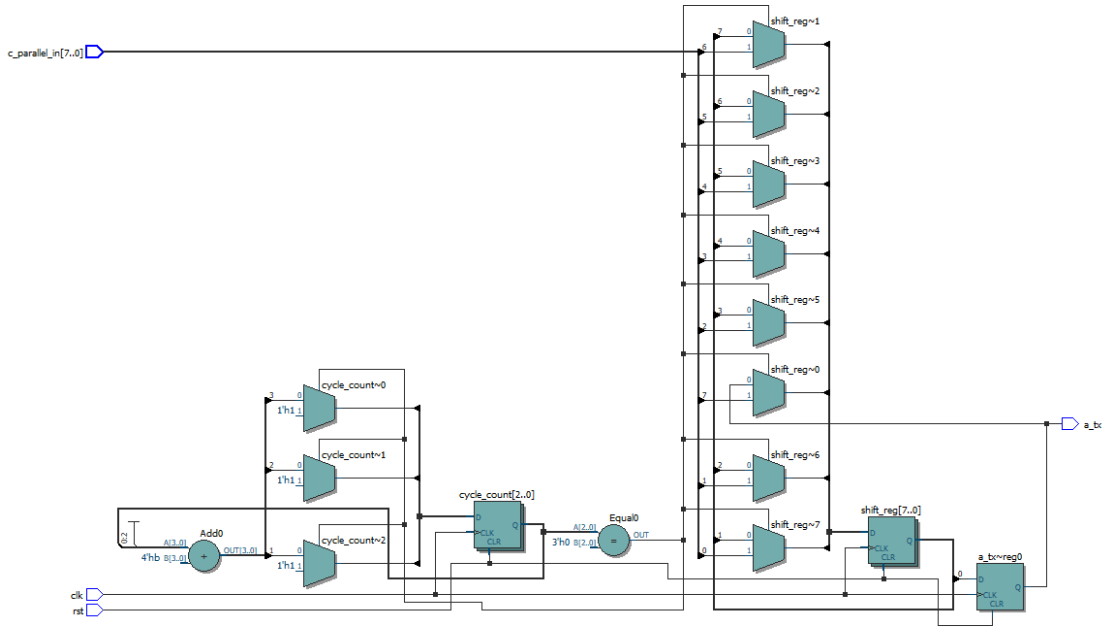


Fig. 1.2 8-Bit Serializer Module

The fig 1.2 shows the designed system is for 8-bit conversion form parallel to serial (serializer)

Testing and verification of serializer system

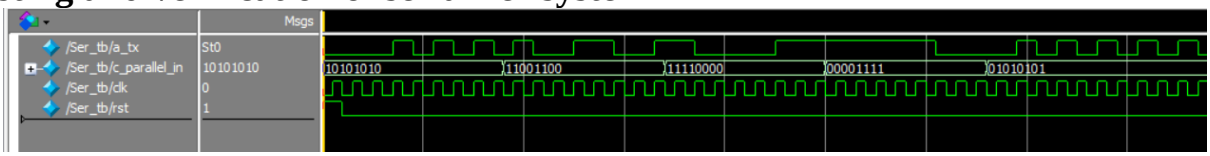


Fig. 1.3 Simulation of Serializer System

In this fig. 1.3 the random data is generated and then it is converted from serial to parallel, then it is transmitted over any medium for communication

1.3 Deserializer system:

A deserializer system is a fundamental component in high-speed digital communication systems, performing the crucial function of converting serialized data back into parallel form. This process is essential in applications that utilize serial communication, such as PCIe, Ethernet, USB, SATA, HDMI, and optical fiber networks. Serial communication is widely used due to its efficiency in reducing the number of interconnects, improving signal integrity, and enabling high-speed data transmission over long distances. However, at the receiving end, the serial data must be reconstructed into its original parallel format for processing by digital systems such as microcontrollers, FPGAs, or ASICs. The deserializer system ensures that the received serial data stream is accurately and efficiently converted back into multiple parallel data lines, maintaining synchronization and data integrity throughout the process.

The core functionality of a deserializer involves extracting individual bits from a high-speed serial data stream and assembling them into a parallel word, which can then be processed by digital logic. This process is typically achieved using shift registers, latches, and clock recovery mechanisms. At the input of a deserializer, a serial data stream arrives at a high rate, synchronized with a clock signal that dictates the bit transmission rate. The incoming serial bits are loaded into a shift register, which continuously shifts data as new bits arrive. Once a complete parallel word is formed, the data is latched and made available at the output as parallel data. The shift register operates in conjunction with a control system that determines when a full parallel word has been received, ensuring that data is correctly aligned and valid before being passed to the output stage.

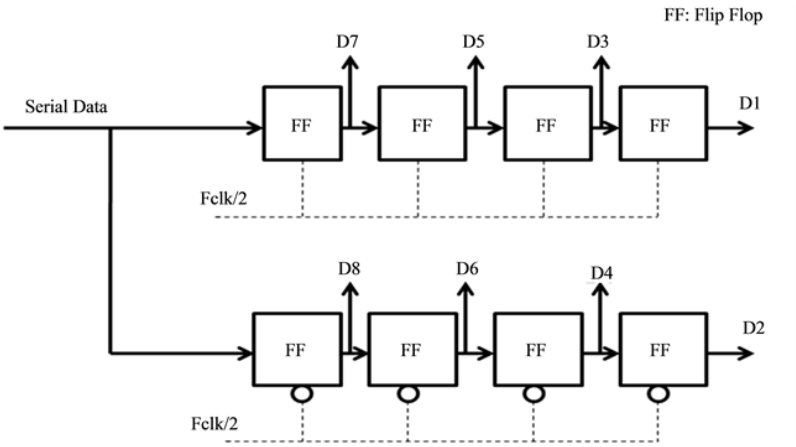


Fig. 1.4 Basic Block Diagram of Deserializer System

The fig1.4 shows a serial-to-parallel converter using flip-flops (FF) operating at half the clock frequency (Fclk/2) to deserialize incoming serial data into parallel outputs.One of the most critical challenges in designing a deserializer system is clock and data recovery (CDR). Since

serial data does not inherently carry a separate clock signal, the receiver must extract timing information from the incoming data stream to correctly sample and deserialize the bits. This is achieved using a phase-locked loop (PLL) or a delay-locked loop (DLL), which synchronizes an internal clock with the incoming data transitions. The PLL continuously adjusts its phase and frequency to align with the received data, ensuring that bits are sampled at the optimal time to minimize errors. In high-speed communication systems, jitter—a variation in signal timing—can cause bit misalignment and data corruption. The PLL compensates for jitter by dynamically adjusting the clock phase to maintain accurate sampling.

Another important aspect of deserializer design is frame alignment. Since the serial data stream does not inherently indicate word boundaries, the receiver must determine the correct alignment of parallel words within the continuous stream. This is typically accomplished using special synchronization patterns or frame markers embedded within the data. These patterns, often defined by the communication protocol, allow the receiver to detect the start of a new word and align the deserialization process accordingly. If misalignment occurs, the deserializer can shift the sampling window until the correct framing is established. Some advanced deserializer designs implement adaptive alignment techniques that continuously monitor and adjust the sampling position to compensate for variations in data transmission.

Data integrity is another crucial factor in deserializer systems, especially in high-speed applications where signal degradation, noise, and interference can introduce bit errors. To ensure reliable data reconstruction, deserializers often incorporate error detection and correction mechanisms. Cyclic redundancy check (CRC) codes, parity bits, and forward error correction (FEC) schemes are commonly used to detect and correct errors in the received data. Additionally, encoding techniques such as 8b/10b, Manchester encoding, and scrambling are employed to maintain DC balance, reduce long runs of identical bits, and facilitate clock recovery. These encoding schemes introduce transitions in the data stream, improving the accuracy of bit synchronization and reducing the likelihood of errors.

Signal integrity plays a significant role in deserializer performance, particularly in systems operating at high data rates. As serial data travels through transmission lines, it experiences attenuation, dispersion, and inter-symbol interference (ISI), which can distort the received signal. To mitigate these effects, deserializer systems often include equalization techniques such as continuous-time linear equalization (CTLE), decision feedback equalization (DFE), and adaptive equalization. These techniques compensate for signal degradation by

amplifying high-frequency components, reducing noise, and improving the overall signal-to-noise ratio (SNR). Proper equalization ensures that the deserializer can accurately recover data even in challenging transmission environments.

Deserializer systems are widely integrated into FPGA and ASIC designs, where they play a critical role in enabling high-speed communication between various components. Modern FPGAs come equipped with built-in high-speed transceivers that include deserialization functionality, allowing direct interfacing with serial communication protocols. These transceivers provide configurable data rates, encoding options, and clock recovery mechanisms to support a wide range of applications. When designing a deserializer in an FPGA, engineers must carefully manage timing constraints, clock domain crossings, and metastability issues to ensure reliable operation. FPGA development tools such as Quartus, Vivado, and ISE offer features for implementing and verifying deserializer systems, including simulation, timing analysis, and signal integrity testing.

In optical communication systems, deserializer systems are essential for converting high-speed serial optical signals into electrical parallel data for further processing. Optical transceivers such as SFP, QSFP, and CFP modules incorporate deserializers along with photodetectors and signal conditioning systems to facilitate high-speed data reception. The ability of deserializers to handle gigabit and terabit data rates makes them indispensable in data centers, telecommunications infrastructure, and high-performance computing environments. Optical deserializers often include advanced equalization and clock recovery techniques to compensate for fiber dispersion and nonlinearities, ensuring reliable data reception over long distances.

Testing and verification of deserializer systems are crucial to ensuring their performance and compliance with industry standards. High-speed deserialization requires thorough validation using test equipment such as oscilloscopes, bit-error rate testers (BERTs), and protocol analyzers. Eye diagrams are commonly used to visualize signal integrity, showing the quality of the received data and identifying potential issues such as jitter, noise, and timing errors. Compliance testing ensures that deserializer systems meet the specifications of communication protocols such as PCIe, USB, HDMI, and Ethernet, guaranteeing interoperability with other devices and systems.

As technology advances, deserializer systems continue to evolve to support higher data rates, lower power consumption, and improved signal integrity. Emerging trends in deserializer design include the adoption of advanced modulation schemes such as pulse amplitude modulation (PAM), which increases data throughput by encoding multiple bits per symbol.

PAM-4, for example, uses four amplitude levels per symbol, effectively doubling the data rate compared to traditional non-return-to-zero (NRZ) signaling. Additionally, innovations in semiconductor technology, such as FinFET transistors and silicon photonics, are enabling the development of high-performance deserializer systems with enhanced efficiency and scalability.

Implementation and Verification of The Deserializer System

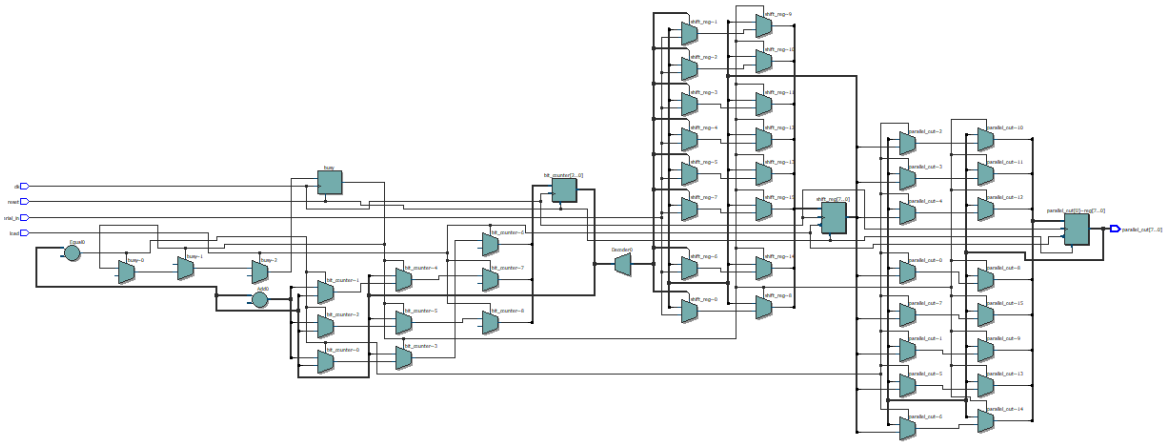


Fig. 1.5 8-Bit Deserializer Block Diagram

Testing and Verification of Deserializer System

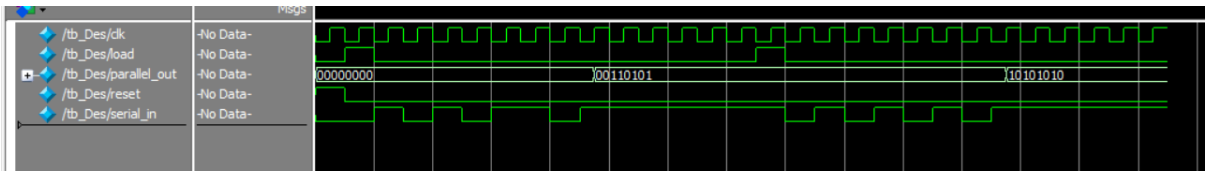


Fig. 1.6 Simulation of Deserializer

The fig. 2.6 shwes the simulation report of deserialization prosses, This system Serial data is received and then it is converted to parallel

1.4 Versatility

- **Used in Various Protocols:** SerDes is fundamental in protocols such as:
- **Ethernet** (10GbE, 40GbE, 100GbE) for high-speed networking.
- **PCIe (Peripheral Component Interconnect Express)** for connecting high- speed peripherals.
- **MIPI (Mobile Industry Processor Interface)** for connecting cameras and displays in mobile devices.
- **HDMI/DisplayPort** for transmitting video and audio.
- **USB** for high-speed data transfer

1.5applications

Serializer/Deserializer (SerDes) systems are widely used in various fields to enable high-speed data transmission over fewer physical connections. Here are some key areas where SerDes is commonly employed:

1.5.1 Telecommunications & Networking

- **Fiber-optic communication**: Transmitting high-speed data over long distances.
- **Ethernet**: High-speed connections in LANs (e.g., 10GbE, 40GbE).
- **SONET/SDH**: Synchronous optical networking for telecommunications.

1.5.2 Data Centers

- **Server-to-server communication**: Facilitates fast data transfers between servers.
- **Storage area networks (SANs)**: Used in protocols like Fibre Channel for storage systems.

1.5.3 Automotive Systems

- **ADAS (Advanced Driver Assistance Systems)**: High-speed data links for cameras and sensors.
- **In-vehicle infotainment (IVI)**: SerDes supports video and audio transmission between control units.

1.5.4 Consumer Electronics

- **Display Interfaces**: HDMI, DisplayPort, and MIPI (Mobile Industry Processor Interface) use SerDes for transmitting video data.
- **Gaming Consoles & VR**: Fast data transmission between GPUs and displays.

1.5.5 Aerospace & Defense

- **Radar Systems**: High-speed data links for processing radar signals.
- **Avionics**: Efficient data communication between onboard systems.

1.5.6 Semiconductor Testing

- **IC testing equipment**: Used for high-speed testing and debugging of integrated systems.

1.5.7 Industrial Automation

- **Robotics**: Fast data exchange between controllers and sensors.
- **PLC Systems**: High-speed data transmission in programmable logic.

1.6Types of SerDes Systems and Their Theoretical Explanation

Serializer/Deserializer (SerDes) systems are critical components in high-speed digital communication systems, enabling efficient data transfer between different system components. The fundamental purpose of a SerDes system is to convert parallel data into a serial data stream for transmission over a high-speed channel and then recover the original

parallel data at the receiver end. This conversion is essential in modern computing, networking, storage, and display technologies, where reducing the number of interconnects while maintaining high data throughput is crucial.

The need for SerDes systems arises due to several challenges in parallel data transmission. As data rates increase, parallel buses face issues such as skew, crosstalk, electromagnetic interference (EMI), and power consumption. Skew occurs when different parallel data lines experience slightly different propagation delays, leading to timing misalignment at the receiver. Crosstalk refers to interference between closely spaced parallel traces, causing signal degradation. EMI results from high frequency switching noise, affecting signal integrity and increasing the risk of data errors. To overcome these issues, serial communication using SerDes systems is preferred, as it requires fewer transmission lines, reduces power consumption, and improves overall signal integrity.

Working Principle of SerDes Systems

A SerDes system comprises a **serializer** at the transmitter and a **deserializer** at the receiver. The serializer converts parallel data from a processor, memory, or digital logic into a high-speed serial bitstream using a shift register or multiplexer. A clock signal ensures synchronization, while encoding techniques maintain signal integrity. At the receiver end, the deserializer reconstructs the original parallel data through **Clock and Data Recovery (CDR)**, extracting timing information from the incoming stream. A **Phase-Locked Loop (PLL) or Delay-Locked Loop (DLL)** generates a stable clock for accurate sampling. The recovered bits are stored in a buffer before being outputted in parallel. The efficiency of serialization and deserialization directly impacts the reliability of high-speed data transmission, making clock recovery and signal integrity critical in SerDes design.

Types of SerDes Systems

SerDes systems can be classified based on several factors, including clocking mechanisms, duplexing methods, equalization techniques, and implementation architectures. Below is a detailed discussion of various types of SerDes systems.

1. **Based on Clocking Mechanism**

   One of the most important distinctions among SerDes systems is how the clock is handled during transmission. This affects timing synchronization, jitter tolerance, and overall reliability.

- Parallel Clock SerDes: This is the simplest type, where a separate clock signal is transmitted alongside the data. The receiver directly uses this clock for sampling the incoming bits. However, at high data rates, clock skew between the data and clock signals can cause timing mismatches, making this method less suitable for very high-speed communication.
- Embedded Clock SerDes: In this type, clock information is embedded within the serial data stream using encoding schemes such as 8b/10b or 64b/66b. This eliminates the need for a separate clock line and reduces skew-related issues. The receiver uses clock and data recovery (CDR) techniques to extract the timing information from the incoming data. This method is widely used in high-speed protocols like PCIe, Ethernet, and USB.

2. Based on Duplexing Method

Duplexing defines whether the transmitter and receiver can communicate simultaneously or need to alternate.

- Full-Duplex SerDes: In full-duplex operation, data transmission and reception occur simultaneously using separate lanes for each direction. This is commonly used in high-speed networking standards such as Ethernet and PCIe, where bidirectional data flow is required for efficient communication.
- Half-Duplex SerDes: In half-duplex mode, data transmission and reception occur on the same channel but at different times. This is useful for applications where bandwidth is limited, and bidirectional communication is needed without requiring extra transmission lines.

3. Based on Signal Encoding and Modulation

Different SerDes systems use different encoding techniques to optimize signal integrity and transmission efficiency.

- PAM-2 (NRZ) SerDes: The Non-Return-to-Zero (NRZ) encoding scheme, also known as Pulse Amplitude Modulation with 2 levels (PAM-2), is the most common signalling [pmethod. It represents binary data with two voltage levels (0 and 1). This method is widely used in conventional communication systems but has limitations at very high data rates due to increased inter-symbol interference (ISI).
- PAM-4 SerDes: To improve data rates without increasing bandwidth, Pulse Amplitude Modulation with 4 levels (PAM-4) is used. PAM-4 encodes two bits per symbol, effectively doubling the data rate while keeping the signal bandwidth constant. This is

used in next-generation high-speed communication standards like 400G Ethernet and PCIe Gen6.

4. Based on Multi-Lane vs. Single-Lane Transmission

SerDes systems can use multiple lanes to increase throughput.

- Single-Lane SerDes: In this configuration, data is transmitted over a single serial channel. It is suitable for low-power, low-complexity designs but is limited in terms of bandwidth.
- Multi-Lane SerDes: For higher bandwidth requirements, multiple serial lanes are used to transmit data in parallel. For example, PCIe and SATA use multi-lane SerDes to increase overall throughput. The challenge in multi-lane SerDes is ensuring proper synchronization and alignment between lanes to prevent data corruption.

5. Based on Implementation

Different types of SerDes systems are implemented based on the target application and system requirements.

- ASIC-Based SerDes: Application-Specific Integrated System (ASIC)-based SerDes systems are optimized for specific use cases, offering high performance, low latency, and power efficiency. These are commonly used in high-speed networking and data center applications.
- FPGA-Based SerDes: Field-Programmable Gate Arrays (FPGAs) offer flexible, reconfigurable SerDes implementations, allowing designers to modify data rates, encoding schemes, and protocols based on application needs. FPGA-based SerDes are widely used in prototyping, research, and embedded system applications.

Challenges in SerDes System Design

While SerDes systems offer many advantages, their design involves several challenges:

- Signal Integrity Issues: High-speed serial transmission introduces problems such as jitter, crosstalk, and attenuation. Advanced equalization techniques like decision feedback equalization (DFE) and continuous time linear equalization (CTLE) help mitigate these effects.
- Power Consumption: High-speed SerDes systems consume significant power, making energy efficiency a critical design consideration, especially in battery-powered applications.

- Clock Synchronization: Ensuring accurate clock recovery at the receiver is complex, especially at very high data rates.
- Error Detection and Correction: Bit errors due to noise and channel imperfections require robust error correction mechanisms.

CHAPTER 2
PARALLEL CLOCK SERDES

A **Parallel Clock SerDes (Serializer/Deserializer) system** is one of the fundamental types of SerDes used in digital communication systems. It operates by transmitting a parallel clock signal alongside the serial data stream, ensuring synchronization between the transmitter and receiver. This type of SerDes is widely used in moderate-speed communication systems where maintaining timing integrity across the transmission channel is critical. The fundamental operation of a parallel clock SerDes is divided into two primary stages: serialization at the transmitter and deserialization at the receiver.

2.1Introduction

At the transmitter end, the serializer takes parallel data, typically consisting of multiple bits (e.g., 8, 16, or 32 bits), and converts it into a high-speed serial bitstream. This conversion process is achieved using shift registers or multiplexers, which sequentially output the bits one at a time onto the transmission line. Along with the serialized data, a dedicated clock signal is also transmitted. This clock signal ensures that the receiver correctly samples the incoming serial data at the intended rate. Since both data and clock signals travel together, the receiver does not need to extract timing information from the data stream, simplifying the design.

At the receiving end, the deserializer reconstructs the original parallel data from the incoming serial stream. Since the clock is transmitted alongside the data, the receiver uses this clock to determine the precise sampling points for the incoming bits. A shift registers or demultiplexer collects the incoming serial bits and groups them back into parallel format. The reconstructed parallel data is then forwarded to the receiving logic, ensuring seamless communication between the transmitter and receiver.

One of the main advantages of a parallel clock SerDes system is its simplicity. Unlike embedded clock SerDes, which requires complex clock and data recovery (CDR) systems, a parallel clock SerDes directly transmits the clock signal, eliminating the need for phase-locked loops (PLLs) or delay-locked loops (DLLs) for clock extraction. This makes it easier to design and implement, particularly in systems where clock synchronization is critical but does not require extremely high data rates.

However, a major drawback of parallel clock SerDes is clock skew—a phenomenon where the transmitted clock and data signals experience different propagation delays across the transmission medium. Since the clock and data travel as separate signals, even slight variations in their arrival times at the receiver can cause sampling errors. This problem becomes more pronounced at higher data rates and longer transmission distances, as signal integrity issues like jitter, crosstalk, and reflections can further exacerbate timing mismatches. To mitigate these issues, careful PCB layout design, impedance matching, and the use of differential signaling techniques are required.

One of the most common applications of Parallel Clock SerDes is in FPGA-to-FPGA communication, where high-speed serial links are necessary to transfer large amounts of data between programmable logic devices. It is also used in memory interfaces, such as DDR and high-speed RAM communication, where precise timing synchronization between the memory controller and the memory module is crucial. Additionally, Parallel Clock SerDes is found in industrial automation and control systems, where robust and low-latency data transfer is required for sensors, actuators, and processing units.

2.2Implementation and Verification of Transmitter Block:

In modern digital communication systems, transmitting data efficiently and reliably over wired channels is essential. High-speed serial communication techniques are preferred over parallel transmission due to their reduced wiring complexity, minimized electromagnetic interference (EMI), and ability to maintain signal integrity over longer distances. The block diagram provided represents a **digital serialization and transmission system**, which consists of three primary functional units: the **serializer, NRZ encoder, and LVDS driver**. Together, these components facilitate the transmission of parallel digital data in a serial format over a differential signaling medium.

Serializer Circuit

The **serializer** is responsible for converting parallel data into a serial bitstream. In digital communication, systems often generate data in parallel form, such as an 8-bit or 16-bit word. However, transmitting multiple parallel bits simultaneously requires multiple wires, which increases the complexity and susceptibility to crosstalk and timing issues. A serializer mitigates this by converting the parallel data into a serial format, reducing the number of transmission lines needed.

The serialization process starts when an **8-bit data word (data_in[7:0])** is loaded into the serializer upon receiving a **load** signal. The circuit then sequentially shifts out each bit at the rate determined by the **clock (clk)**. A shift register or multiplexer-based approach is typically used to output one bit per clock cycle onto the **serial_out** line. This ensures that all eight bits are transmitted one after another within eight clock cycles.

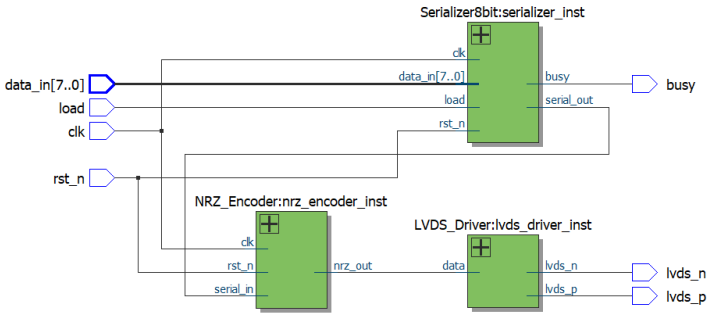


Fig. 2.1 Parallel Clock Serdes Based Transmitter Module

The figur2.1 shows Digital transmitter system for short distance communication which consist of a NRZ encoder and LVDS driver which sends data over a parallel line

The serializer also includes a **busy signal**, which indicates whether data is currently being processed. This prevents new data from being loaded until the current serialization cycle is complete. Additionally, an **active-low reset (rst_n)** is used to initialize or reset the circuit, ensuring that all operations start from a known state.

NRZ (Non-Return-to-Zero) Encoding

Once data is serialized, it is forwarded to the **NRZ encoder**, which modifies the signal format for reliable transmission. NRZ encoding is one of the simplest and most widely used data encoding schemes, where:

- A logic **'1'** is represented by a **high voltage level**,
- A logic **'0'** is represented by a **low voltage level**,
- There are **no transitions or returns to zero** between consecutive bits of the same value.

NRZ encoding improves signal efficiency by eliminating unnecessary transitions, reducing bandwidth requirements, and ensuring proper signal reconstruction at the receiver. However, one of the challenges of NRZ encoding is the potential for long sequences of consecutive '1's or '0's, which can lead to synchronization issues due to the absence of transitions. In high-speed applications, additional techniques such as **clock encoding** or **DC balancing** (e.g., Manchester encoding or 8b/10b encoding) may be used to complement NRZ.

LVDS (Low-Voltage Differential Signalling) Driver

The final stage of the system is the **LVDS driver**, which converts the NRZ-encoded data into a **differential signal** for high-speed, low-noise transmission. LVDS is a signalling standard designed to operate at high data rates while minimizing power consumption and EMI. Unlike single-ended signaling, where data is transmitted using a single wire referenced to ground, LVDS transmits data as a differential pair:

- **lvds_p (positive signal line)**
- **lvds_n (negative signal line)**

The LVDS driver takes the encoded data as input and generates two complementary signals. When the input data is **high (1)**, lvds_p is driven high, and lvds_n is driven low. When the input data is **low (0)**, lvds_p is driven low, and lvds_n is driven high. This differential transmission reduces noise susceptibility and allows data to be transmitted reliably over long distances.

Verification of transmitter block:

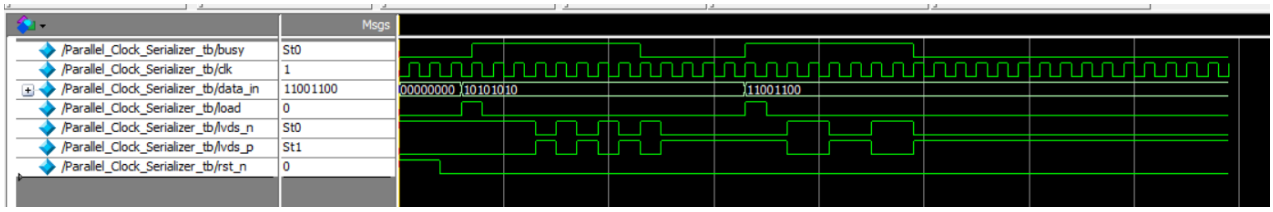


Fig. 2.2 Simulation of Transmitter Module

The fig. 2.2 shows conversion of data from parallel data to serial data and the transmission over a serial line

2.3Implementation and verification OF receiver block

The given block diagram represents a digital wired communication receiver system, which consists of three main components: the LVDS Receiver, NRZ Decoder, and Deserializer. These components work together to convert a high-speed serial data stream into parallel data, making it suitable for processing by digital circuits such as microcontrollers, FPGAs, or processors. This system is commonly used in applications where data needs to be transmitted efficiently over a wired interface with minimal noise and power consumption.

LVDS Receiver

The LVDS (Low-Voltage Differential Signaling) Receiver is the first stage in the system, responsible for recovering the transmitted signal from the differential pair (lvds_p and lvds_n). LVDS is a widely used high-speed communication standard that transmits data using a differential voltage between two wires rather than a single-ended signal. This approach significantly reduces electromagnetic interference (EMI), power consumption, and signal degradation, making LVDS ideal for long-distance and high-speed data transmission.

In the diagram, the LVDS receiver takes the lvds_p and lvds_n signals as input and converts them into a single-ended digital output (data_out). This recovered signal is then passed to the next stage for further processing. The LVDS receiver operates using a clock (clk) signal to ensure proper timing synchronization and a reset (reset) signal to initialize or reset its state when needed.

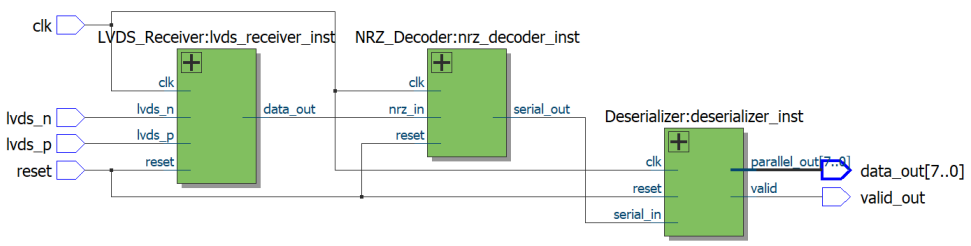


Fig. 2.3 Digital Receiver System for Short Distance Communication

NRZ Decoder

Once the LVDS receiver converts the differential signal into a single-ended signal, the data is still encoded in the **NRZ (Non-Return-to-Zero) format**. The **NRZ Decoder** is responsible for **decoding the serialized data** into a form that can be processed by digital circuits.

NRZ encoding is a commonly used data representation technique in which:

- A **logic '1'** is represented by a **high voltage**.
- A **logic '0'** is represented by a **low voltage**.
- The signal remains at the same level for consecutive bits of the same value (i.e., no automatic transitions).

While NRZ encoding simplifies signal transmission by reducing bandwidth usage, it can introduce challenges in long sequences of '0's or '1's, potentially causing synchronization issues. The NRZ decoder reconstructs the original serialized digital data from the **nrz_in** input signal and outputs it as **serial_out**, which is then fed into the next stage.

The NRZ decoder also operates using the **clock (clk)** to maintain proper synchronization and the **reset (reset)** signal to clear its state when necessary.

Deserializer

The Deserializer is the final stage of the receiver system. It takes the serial data stream from the NRZ decoder and converts it back into parallel data. This step is necessary because most digital processing units (such as FPGAs, microcontrollers, and CPUs) operate on parallel data, whereas serial transmission is preferred for long-distance and high-speed communication.

- The deserialization process involves:
- Receiving the serial_in signal, which represents a stream of bits arriving sequentially.
- Storing the received bits in a shift register until a complete set (e.g., 8-bit word) is collected.
- Outputting the collected data as parallel_out[7:0], making it available for further processing.
- Generating a valid_out signal to indicate when a valid parallel word has been assembled and is ready for use.

The deserializer operates using the **clock (clk)** to ensure correct timing and a **reset (reset)** signal to initialize or reset the circuit when needed.

Verification of Receiver block:

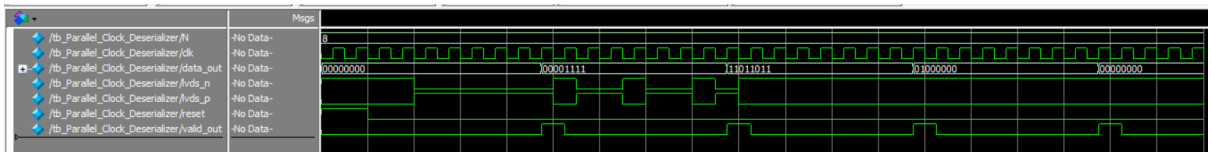


Fig. 2.4 Simulation of Receiver Module

- Clock Jitter Rate refers to the rate of variation in the timing of a clock signal from its ideal periodicity over time.
- Data rate is crucial as it determines the speed and efficiency of data transmission, directly impacting system performance, bandwidth utilization, and communication reliability.

CHAPTER 3
EMBEDDED CLOCK SERDES

A Serializer/Deserializer (SerDes) is a crucial circuit used in high-speed digital communication systems to convert parallel data into serial form for transmission and then back into parallel form at the receiver end. One of the key challenges in serial communication is ensuring that the transmitter and receiver remain synchronized, as traditional parallel communication inherently includes a clock signal to coordinate data transfer. In Embedded Clock SerDes, the clock signal is embedded within the data stream rather than being sent as a separate signal. This technique ensures accurate data recovery at the receiver while minimizing the number of required signal lines, reducing complexity, and improving signal integrity. Embedded clock SerDes is widely used in high-speed data transmission protocols such as PCIe, USB, HDMI, DisplayPort, and Gigabit Ethernet.

3.1Theory of Embedded Clock SerDes

In traditional parallel communication, each data line has a dedicated clock signal that ensures synchronization between the transmitter and receiver. However, in high-speed applications, sending a separate clock signal can introduce timing skews, jitter, and electromagnetic interference (EMI), leading to data integrity issues. Embedded clock SerDes addresses this by incorporating the clock within the data stream itself, allowing the receiver to extract timing information directly from the received signal. This approach enables high-speed, long-distance communication while ensuring minimal signal degradation and optimal timing synchronization.

The embedded clock technique is based on data encoding methods that ensure frequent transitions within the serial data stream. These transitions help the receiver to recover the clock using Clock and Data Recovery (CDR) circuits. Some of the most used encoding techniques include 8b/10b encoding, 64b/66b encoding, Manchester encoding, and Scrambling.

Working Principle of Embedded Clock SerDes

An Embedded Clock SerDes consists of two primary blocks:

1. **Serializer (Transmitter Side)**: Converts parallel data into a serial data stream while embedding clock information into the signal.
2. **Deserializer (Receiver Side)**: Recovers the clock from the serial stream and reconstructs the original parallel data.

3.2 Implementation and verification of transmitter using embedded clock serializer

The block diagram presented represents a data transmission system that utilizes 8b/10b encoding and serialization to convert parallel data into a high-speed serial data stream. This technique is widely used in digital communication systems, such as PCI Express (PCIe), USB 3.0, SATA, Ethernet, and HDMI, to ensure reliable and efficient data transmission. The two primary components of the system are an 8b/10b encoder and a serializer, both of which play crucial roles in data integrity, clock synchronization, and transmission efficiency.

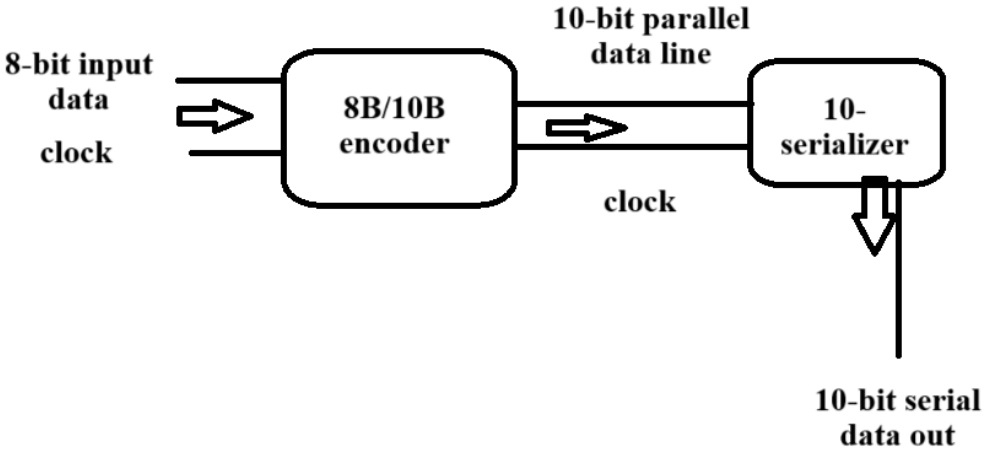


Fig. 3.1 Block Diagram of Transmitter Module

The Fig3.1 illustrates an 8B/10B encoding and serialization process, where 8-bit input data is encoded into a 10-bit parallel format and then serialized into a 10-bit serial data stream.

Overview of the System

The system starts with an 8-bit parallel data input (`din[7..0]`), which is fed into an 8b/10b encoder to produce a 10-bit encoded output (`dout[9..0]`). This encoded data is then passed to a serializer, which converts the 10-bit parallel data into a serial bitstream (`serial_out`). The serial output is then transmitted over a single data line, making the system well-suited for high-speed digital communications.

8b/10b Encoding

The 8b/10b encoding scheme was introduced by IBM to address multiple challenges in serial data transmission, including DC balance, clock recovery, and error detection. The encoding process maps each 8-bit data word into a 10-bit code, ensuring certain desirable properties in the transmitted signal.

- **DC Balance:** Long sequences of only '1's or '0's in the transmitted data can cause **baseline wander**, affecting receiver sensitivity. The 8b/10b encoding ensures that the number of '1's and '0's remains balanced, minimizing DC bias.
- **Clock Recovery:** In a serial data system, the receiver must extract the clock signal from the data stream. The encoding ensures that no long runs of consecutive identical bits occur, enabling easy clock extraction using a phase-locked loop (PLL).
- **Error Detection:** Certain 10-bit codes are illegal, meaning that if they appear in the received data, an error has occurred. This feature provides basic error detection capabilities.
- **Control Character Transmission:** Apart from data encoding, 8b/10b encoding supports **control characters** used for signaling, such as **start-of-frame, end-of-frame, and idle sequences**.

3.2.1 Implementation of transmission logic

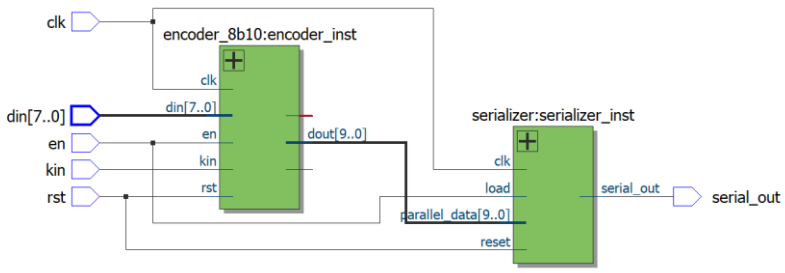


Fig. 3.2 Implemented Block Diagram of Transmitter Module

This fig 3.2 design includes the 8B/10B encoder system and 10Bit serializer system construction, which shows the simulation of this in next fig 3.3 this will convert the data in to serial and increases the frequency and converts that in to 10 times

Verification of this transmitter logic

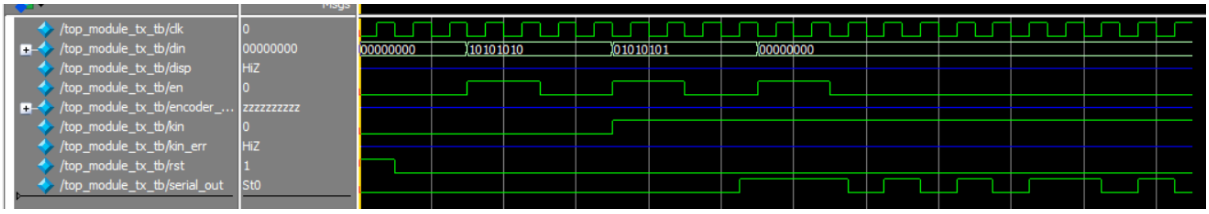


Fig. 3.3 Simulation Report of Transmitter Module

In this fig 3.3 we can observe the 8-bit input is generated and then it is converted in 10-bit then it is converted into serial format

3.2.2 Output and observation

Synchronization between the encoder and serializer is crucial. The encoder operates at a given system clock frequency (`clk`), and the serializer must operate at 10 times the clock rate to maintain the correct data rate. This means if the encoder is clocked at 100 MHz, the serializer must shift out data at 1 GHz to maintain a continuous data stream.

The 8b/10b encoding and serialization system provides several benefits for modern digital communication applications:

1. **Efficient High-Speed Data Transmission:** Converts parallel data into a serial stream, allowing for faster transmission rates with minimal interconnects.
2. **Reliable Data Integrity:** 8b/10b encoding ensures proper DC balance, helping maintain signal integrity over long distances.
3. **Improved Clock Recovery:** Eliminates the need for a separate clock signal by embedding timing information within the data.
4. **Error Detection:** The system inherently detects certain types of bit errors by identifying invalid 10-bit sequences.
5. **Compatibility with Industry Standards:** The encoding and serialization process is widely used in high-speed interfaces like **PCIe, SATA, USB 3.0, Ethernet, and DisplayPort**.

3.3 8B/10B encoder & decoder

The 8B/10B encoding scheme is a widely used line code for high-speed serial data transmission. It was developed by IBM to address issues related to DC balance, clock recovery, and error detection in digital communication systems. This encoding technique

converts 8-bit data words into 10-bit transmission characters, ensuring that the data stream has controlled characteristics for reliable signal transmission over long distances. The scheme is commonly used in protocols like PCI Express, SATA, USB 3.0, Ethernet, and Fibre Channel.

The 8B/10B encoding process takes 8-bit input data and converts it into a 10-bit code that maintains an equal distribution of 1s and 0s (DC balance) and prevents long sequences of consecutive identical bits. The encoding scheme consists of two stages. The first five bits of the input data, often referred to using the notation `D.x.y`, are encoded into a 6-bit symbol, while the remaining three bits are encoded into a 4-bit symbol. This structured conversion ensures that the transmitted signal always contains sufficient transitions to enable clock recovery at the receiver end. Additionally, the scheme provides special codes for control characters, enabling the transmission of signaling information along with data.

One of the primary advantages of 8B/10B encoding is its ability to maintain DC balance in the transmitted data. By ensuring that the number of 1s and 0s remains approximately equal, the encoding prevents baseline wander, which can affect the accuracy of data transmission. Another important advantage is clock recovery, as the scheme eliminates long sequences of consecutive 1s or 0s, allowing the receiver to extract the clock from the data without the need for a separate clock signal. The encoding scheme also supports error detection by ensuring that only valid 10-bit codes are used; if an invalid code is received, an error can be flagged. Furthermore, the scheme allows for the encoding of control characters, which are essential for marking important events in communication, such as the start or end of a transmission frame.

The 8B/10B decoding process is the inverse of encoding. The 10-bit received data is mapped back into an 8-bit data word. The decoder must verify the validity of the 10-bit code, detect errors, and ensure proper synchronization. If a received 10-bit symbol is not present in the encoding table, the decoder flags an error. The decoder also ensures that the running disparity (RD), which keeps track of the number of 1s and 0s transmitted, remains balanced over time. Once a valid 10-bit symbol is confirmed, it is converted back to its original 8-bit format and delivered as output.

The 8B/10B encoding scheme is widely used in high-speed communication interfaces due to its ability to ensure reliable data transmission. It is an essential part of PCI Express (PCIe),

where it guarantees efficient data transfer between components on a motherboard. The Serial ATA (SATA) standard uses 8B/10B encoding to facilitate high-speed connections between storage devices. In USB 3.0 and beyond, this encoding scheme enhances data integrity for high-speed USB communication. The scheme also plays a critical role in Gigabit Ethernet and Fibre Channel networking, ensuring error-free data transmission across long distances. Additionally, digital video interfaces such as HDMI and DisplayPort employ this encoding method to maintain signal integrity in high-resolution video transmissions.

The 8B/10B encoding and decoding scheme is a fundamental component of modern serial communication systems. By transforming 8-bit data into a structured 10-bit code, it ensures signal integrity, error detection, and efficient clock recovery. This encoding method has been adopted in multiple high-speed digital protocols, making it an essential technique for FPGA designers, embedded engineers, and communication system developers. The ability to maintain DC balance, prevent signal degradation, and support control characters makes 8B/10B encoding one of the most reliable and efficient data transmission schemes in modern electronics.

Working of the Encoder

The working of the 8B/10B encoding and decoding scheme revolves around converting an 8-bit input into a 10-bit output while maintaining DC balance, ensuring clock recovery, and enabling error detection. The encoding process starts with an 8-bit data word, which is divided into two parts: the first five bits are encoded into a 6-bit group, and the remaining three bits are encoded into a 4-bit group. These transformations are performed using predefined lookup tables that assign specific 10-bit symbols to each possible 8-bit input. The choice of a particular 10-bit code depends on the running disparity (RD), which tracks the number of 1s and 0s transmitted. If the disparity becomes too positive (more 1s) or too negative (more 0s), an alternate 10-bit code is chosen to maintain balance. This mechanism ensures that the transmitted signal does not develop excessive baseline wander, which can lead to signal degradation over long distances.

At the receiver side, the decoder extracts the original 8-bit data by identifying the corresponding 10-bit symbol from the predefined encoding table. If an invalid 10-bit code is

5b/6b code (abcdei) [edit]

Input		RD = −1	RD = +1	Input		RD = −1	RD = +1	
Code	EDCBA	a b c d e i		Code	EDCBA	a b c d e i		
D.00	00000	100111	011000	D.16	10000	011011	100100	
D.01	00001	011101	100010	D.17	10001	100011		
D.02	00010	101101	010010	D.18	10010	010011		
D.03	00011	110001		D.19	10011	110010		
D.04	00100	110101	001010	D.20	10100	001011		
D.05	00101	101001		D.21	10101	101010		
D.06	00110	011001		D.22	10110	011010		
D.07	00111	111000	000111	D.23 †	10111	111010	000101	*also used for the K.23.7 symbol*
D.08	01000	111001	000110	D.24	11000	110011	001100	
D.09	01001	100101		D.25	11001	100110		
D.10	01010	010101		D.26	11010	010110		
D.11	01011	110100		D.27 †	11011	110110	001001	*also used for the K.27.7 symbol*
D.12	01100	001101		D.28	11100	001110		
D.13	01101	101100		D.29 †	11101	101110	010001	*also used for the K.29.7 symbol*
D.14	01110	011100		D.30 †	11110	011110	100001	*also used for the K.30.7 symbol*
D.15	01111	010111	101000	D.31	11111	101011	010100	
not used		**1111**00	**0000**11	K.28 ‡	11100	00**1111**	11**0000**	exclusively used for K.28.x symbols

Fig. 3.4 Converting First 5-Bits Into 6-Bit Symbols Using This Table

3b/4b code (fghj) [edit]

Input		RD = −1	RD = +1	Input		RD = −1	RD = +1
Code	HGF	f g h j		Code	HGF	f g h j	
D.x.0	000	1011	0100	K.x.0	000	1011	0100
D.x.1	001	1001		K.x.1 ‡	001	**0**110	**1**001
D.x.2	010	0101		K.x.2	010	1010	0101
D.x.3	011	1100	0011	K.x.3	011	1100	0011
D.x.4	100	1101	0010	K.x.4	100	1101	0010
D.x.5	101	1010		K.x.5 ‡	101	**0**101	**1**010
D.x.6	110	0110		K.x.6	110	1001	0110
D.x.P7 †	111	1110	0001	K.x.7 ‡	111	**0**111	**1**000
D.x.A7 †		0111	1000				

Fig. 3.5 Converting First 3-Bits Into 4-Bit Symbols Using This Table

Control symbols

The control symbols within 8b/10b are 10b symbols that are valid sequences of bits (no more than six 1s or 0s) but do not have a corresponding 8b data byte. They are used for low-level control functions. For instance, in Fibre Channel, K28.5 is used at the beginning of four-byte sequences (called "Ordered Sets") that perform functions such as Loop Arbitration, Fill Words, Link Resets, etc.

Control symbols

Input				RD = −1	RD = +1
Symbol	DEC	HEX	HGF EDCBA	abcdei fghj	abcdei fghj
K.28.0	28	1C	000 11100	001111 0100	110000 1011
K.28.1 †	60	3C	001 11100	00**1111 1**001	11**0000 0**110
K.28.2	92	5C	010 11100	001111 0101	110000 1010
K.28.3	124	7C	011 11100	001111 0011	110000 1100
K.28.4	156	9C	100 11100	001111 0010	110000 1101
K.28.5 †	188	BC	101 11100	00**1111 1**010	11**0000 0**101
K.28.6	220	DC	110 11100	001111 0110	110000 1001
K.28.7 ‡	252	FC	111 11100	00**1111 1**000	11**0000 0**111
K.23.7	247	F7	111 10111	111010 1000	000101 0111
K.27.7	251	FB	111 11011	110110 1000	001001 0111
K.29.7	253	FD	111 11101	101110 1000	010001 0111
K.30.7	254	FE	111 11110	011110 1000	100001 0111

Fig. 3.6 controlling Signal

Resulting from the 5b/6b and 3b/4b tables the following 12 control symbols are allowed to be sent. The 8B/10B encoding scheme provides significant advantages in high-speed serial communication by ensuring DC balance, reducing transmission errors, and enabling reliable clock recovery. One key benefit is its ability to maintain a balanced number of ones and zeros in the transmitted data, preventing long runs of identical bits, which helps receivers maintain synchronization. This encoding also includes built-in disparity control, which reduces the risk of baseline wander in AC-coupled systems. Additionally, 8B/10B encoding introduces controlled transitions, improving error detection and correction capabilities. By ensuring frequent signal transitions, it facilitates clock recovery in serial links, eliminating the need for separate clock signals. This is particularly useful in gigabit Ethernet, Fibre Channel, and PCI Express, where reliable high-speed data transfer is essential. While the encoding increases data overhead by 25%, the trade-off in data integrity and synchronization far outweighs the cost, making it a widely used encoding standard in modern digital communication systems.

Encoder

Implementation

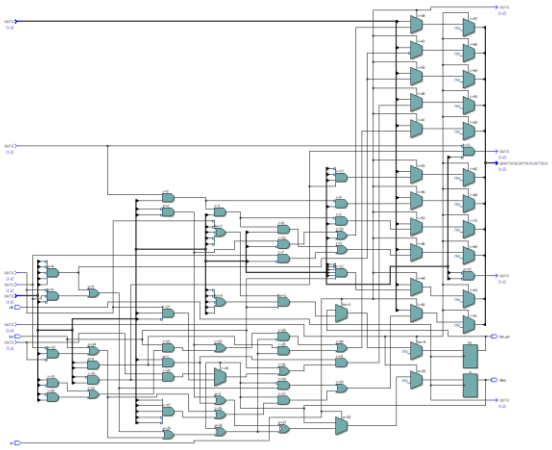


Fig. 3.7 8B/10B Encoding System

Verification of encoder:

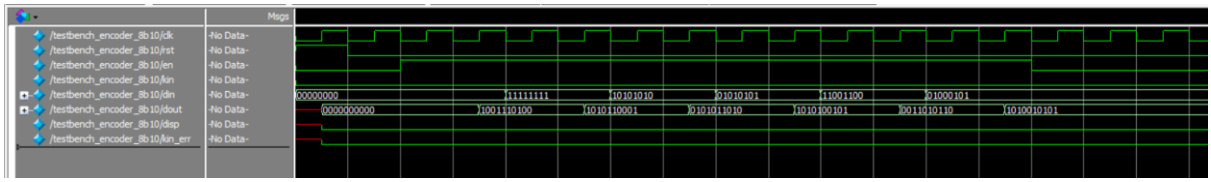


Fig. 3.8 Encoding Implementation and Verification

The simulation shows the encoding of random data and converted from 8 bit to 10 bit

Implementation of decoder:

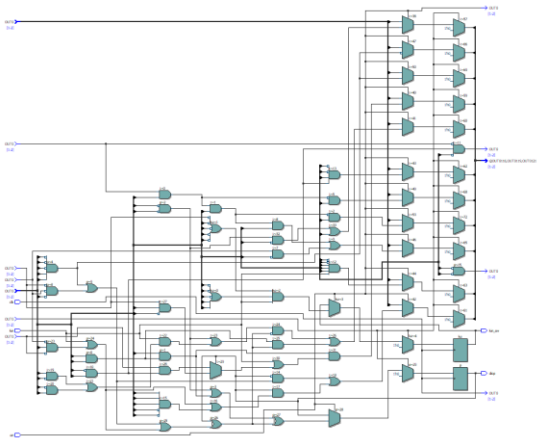


Fig. 3.9 10B/8B Decoding System

Verification of decoder system

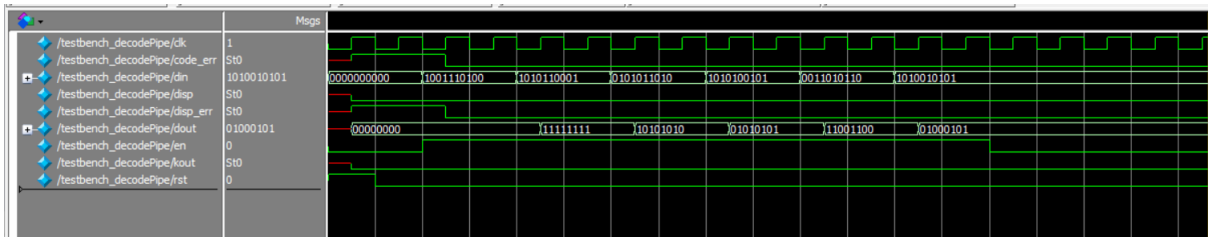


Fig. 3.10 Decoding Implementation and Verification

The 10-bit data is then recovered to 8-bit data

Theoretical checking:

Table 3.0-1 Output Comparison of Encoder -Decoder

Input data 8-bit	**Theoretical encoded 10-bit data**	**Encoded data 10-bit**	**Decoded data 8-bit**
00000000	1001110100	1001110100	00000000
11111111	1010110001	1010110001	11111111
10101010	0101011010	0101011010	10101010
01010101	1010100101	1010100101	01010101
11001100	0011010110	0011010110	11001100
00110011	1100101001	1100101001	00110011

3.4Implementation and verification of receiver using embedded clock deserializer

This block diagram represents a data reception and processing system, likely designed for high-speed serial data communication. The system consists of three main modules: an input buffer (inputBuffer), a deserializer (deserializer), and a decoder pipeline (decodePipe). These modules work together to receive a serial data stream, convert it into parallel data, and decode it for further processing.

The system starts with the input Buffer, labelled as input_buffer_inst. This module takes in the incoming serial data signal a_rx_serial, along with a clock signal clk, a reset signal rst, and a disparity signal disparity_d. The input Buffer processes the serial input and outputs the buffered data (dout). This buffered data is then sent to the next stage, which is the deserializer.

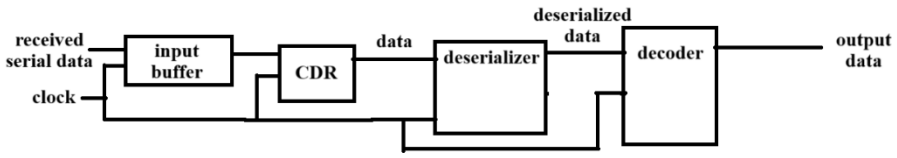


Fig. 3.11 Basic Block Diagram of Receiver

The fig 3.11 represents a high-speed serial data reception system. It consists of an input buffer, a clock and data recovery (CDR) unit, a deserializer, and a decoder. The received serial data is buffered, clock-synchronized, deserialized into parallel format, and decoded to reconstruct the original output data for further processing.

The deserializer, labelled as deserializer_inst, receives the buffered serial data (a_rx) from the input buffer. It operates using the same clk and rst signals and takes in the disparity signal (disparity_d). The deserializer's role is to convert the incoming serial data into parallel data. This is crucial for systems that need to process multiple bits of data simultaneously rather than one bit at a time. The deserializer generates two important outputs: c_parallel_out[9..0], which represents the 10-bit parallel data output, and c_data_valid, which indicates when the output data is valid. Additionally, it outputs clk_out, a synchronized clock signal, and disparity_q, which is likely used for tracking running disparity in encoded data systems like 8b/10b encoding.

The parallel data output from the deserializer (c_parallel_out[9..0]) is then fed into the decodePipe module, labelled as decoder_inst. This module is responsible for decoding the received 10-bit parallel data into meaningful 8-bit output data (dout[7..0]). It operates using the system clock (clk), reset (rst), and an enable signal (en). The decoder performs error checking and outputs various status signals: code_err, which indicates any code violations; disp_err, which flags any disparity errors in the received data; and disp, which provides disparity information. Additionally, the decoder outputs kout, which might be a control signal indicating whether a special character (K-character) was received.

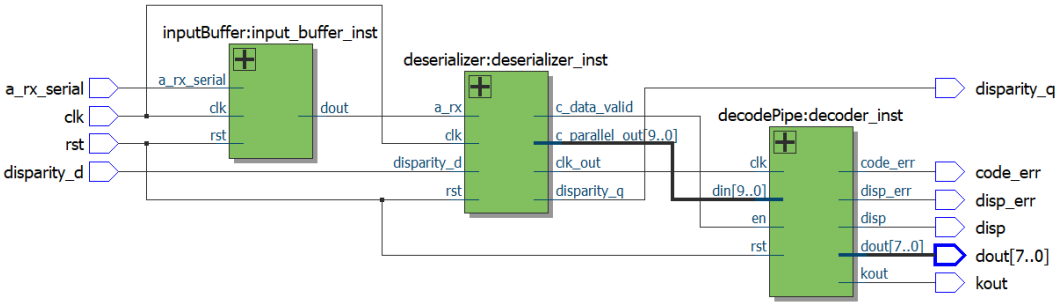


Fig. 3.12 The Receiver Module

Overall, this block diagram represents a structured approach to handling high-speed serial data. The system first buffers the incoming data, then converts it into parallel format, and finally decodes it while checking for errors. This architecture is commonly found in communication protocols such as PCIe, SATA, and Ethernet, where serial-to-parallel conversion and error detection are critical for reliable data transmission.

Verification of this system

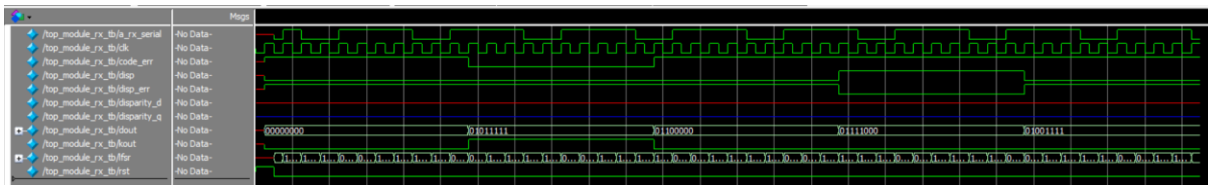


Fig. 3.13 Simulation Result of Embedded Clock-Based Receiver

in fig. 22 the Output of embedded clock deserializer based receiver system the data is received, and it converts the data from serial to parallel and then it converted and then decoded to 8-bit data

3.5Clock Divider Logic-Based Clock Data Recovery (CDR) Module

In this receiver module the deserializer contains the CDR module so that it can recover data and clock,A Clock and Data Recovery (CDR) module based on a clock divider logic is a fundamental circuit used in high-speed serial communication systems to extract a timing reference from incoming data streams. In serial communication, data is transmitted without a separate clock signal, requiring the receiver to recover the clock from the data itself. A clock divider-based CDR operates by using a frequency divider to generate a lower-frequency clock from a high-speed reference clock and align it with the incoming data transitions.

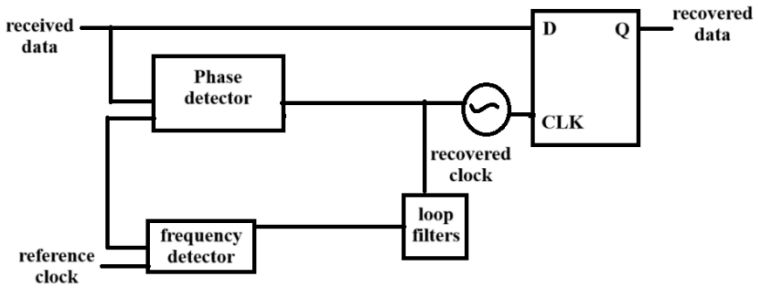


Fig. 3.14 Basic Block Diagram of Theoretical CDR Block

A Clock and Data Recovery (CDR) module is a fundamental component in high-speed digital communication systems, ensuring accurate timing synchronization between transmitted data and the receiving system. The CDR module primarily consists of three essential blocks: a phase detector, a loop filter, and a clock divider. These components work together to extract a clean clock signal from incoming serial data streams, thereby enabling reliable data sampling and minimizing errors due to timing mismatches.

The phase detector plays a crucial role in the CDR system by comparing the phase of the incoming data transitions with the phase of a locally generated clock signal. This comparison is necessary because, in practical communication scenarios, the incoming data stream does not inherently contain an explicit clock signal. Instead, the embedded timing information is extracted based on signal transitions. If a phase difference is detected between the incoming data and the local clock, the phase detector generates an error signal that represents the magnitude and direction of the phase misalignment. The most commonly used phase detectors in CDR circuits include Alexander-type detectors and Bang-Bang phase detectors, both of which are widely implemented in modern serial communication protocols.

Following the phase detection, the loop filter processes the generated error signal to produce a smooth control voltage or digital control word. The loop filter is responsible for determining the stability and dynamic response of the CDR system. It suppresses high-frequency noise components that may arise due to jitter or channel distortions while allowing low-frequency components to pass through for effective clock correction. The loop filter

Design of CDR Module

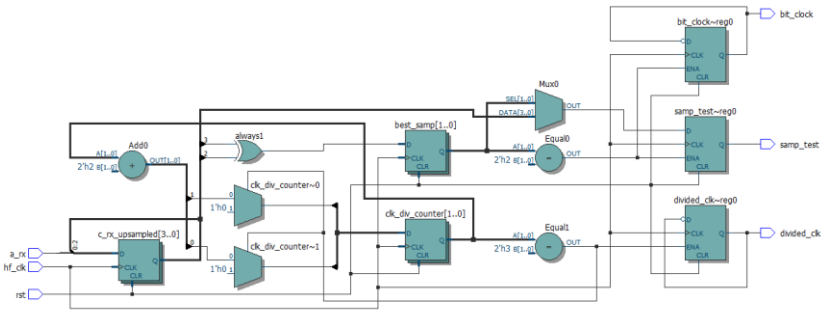


Fig. 3.15 Block Diagram Of CDR

Verification of CDR module

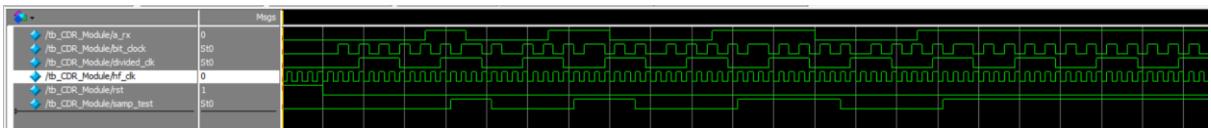


Fig. 3.16 Simulation Report of CDR Module

The output contains the divided clock with 25% of original frequency and recovered data which is transmitted over a single line

3.6Buffer Module

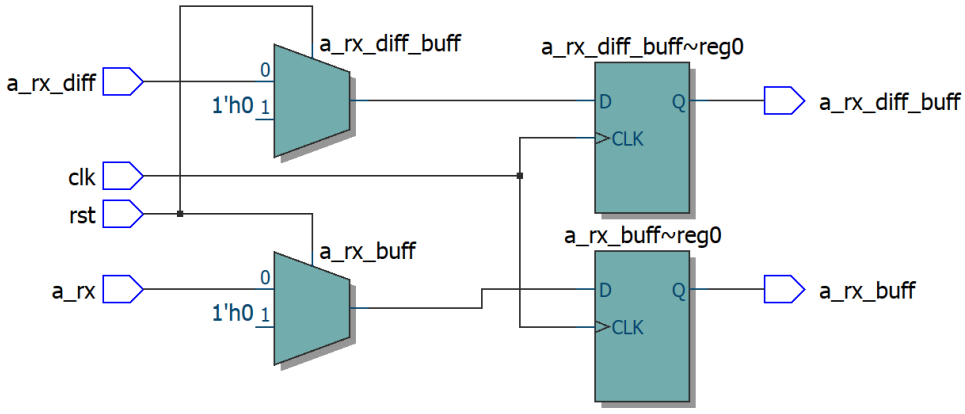


Fig. 3.17 Block Diagram of Buffer Module

This block diagram represents a simple serial-to-parallel data latch, primarily used for capturing and synchronizing serial input data with a system clock. It consists of two D flip-

flops: shift_reg[0] and reg0. The input signal a_rx_serial is fed into shift_reg[0], which captures the data on the rising edge of the clock (clk). The Q output of this register is then forwarded to reg0, which acts as a latching register to store the final output (dout).

The reset signal (rst) clears the registers, ensuring that the system starts from a known state. This design ensures stable and synchronized data transfer, minimizing metastability issues caused by asynchronous data inputs. The second register (reg0) prevents glitches by holding the data stable while new bits arrive in the shift register.

This type of synchronization circuit is critical in high-speed serial communication systems, such as UART receivers, SPI interfaces, and digital signal processing pipelines. It ensures that serial data is captured, processed, and transferred reliably into a parallel domain, where further logic can operate on it. This simple yet essential circuit improves data integrity, timing synchronization, and overall system robustness in FPGA-based and digital electronics applications.

Verification of input buffer module

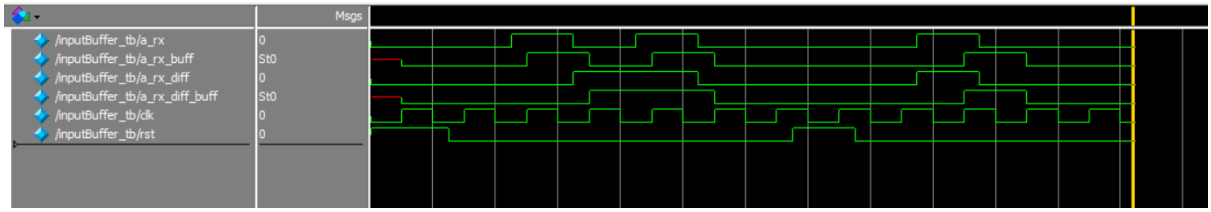


Fig. 3.18 Simulation Report of Buffer Module

Fig. 3.17 shows buffer module is storing data for time till the CDR module converts the frequency so that we can convert and compare data and then we can send the data to deserialization prosses

CHAPTER 4
NUMERICAL CONTROLLED OSCILLATOR (NCO) SYSTEM

A **Numerically Controlled Oscillator (NCO)** is a digital signal processing component used to generate sinusoidal, square, or other periodic waveforms with precise frequency control. It is widely used in applications such as digital communication systems, software-defined radios, signal synthesis, and modulation techniques. Unlike traditional analog oscillators that rely on resistors, capacitors, and inductors, an NCO operates purely in the digital domain, providing greater stability, accuracy, and programmability.

4.1Design

The core principle of an NCO is phase accumulation, where a phase accumulator continuously increments a phase value at each clock cycle. The rate at which the phase accumulates determines the output frequency. This phase information is then used to index a Look-Up Table (LUT) containing precomputed waveform values, typically for a sine or cosine function. By reading the stored values sequentially, the NCO can generate a smooth sinusoidal waveform. The frequency resolution of the NCO depends on the bit-width of the phase accumulator, with higher bit-widths providing finer control over frequency adjustments.

One of the significant advantages of an NCO is its programmability**,** allowing dynamic frequency changes without requiring hardware modifications. By adjusting the phase increment value, the output frequency can be modified in real time. This makes NCOs particularly useful in frequency synthesis applications, such as digital down-conversion, up-conversion, and signal modulation in wireless communication systems.

A Numerically Controlled Oscillator (NCO) is a digital circuit used for generating precise frequency signals. It consists of a phase accumulator, a look-up table (LUT), and a digital-to-analog converter (DAC). The phase accumulator increments phase values based on an input frequency control word, which addresses the LUT containing sine wave samples. The DAC converts these digital samples into an analog waveform, ensuring stable and programmable frequency synthesis in communication and signal processing systems.

Fig. 4.1 Block Diagram of NCO System

Another important feature of NCOs is their phase continuity, which ensures that frequency transitions do not introduce abrupt phase jumps. This is crucial in applications like frequency-hopping spread spectrum (FHSS) communication and quadrature amplitude modulation (QAM) systems, where maintaining a stable phase relationship is essential for signal integrity.

In modern FPGA and ASIC designs**,** NCOs are implemented using Verilog or VHDL and optimized for high-speed processing. They are often integrated into Direct Digital Synthesis (DDS) systems, where they work alongside digital-to-analog converters (DACs) to generate precise analog waveforms. The combination of an NCO with a DAC allows high-fidelity waveform generation with minimal distortion.

One of the main design considerations for an NCO is spurious noise and spectral purity. The finite resolution of the phase accumulator and LUT can introduce quantization errors and spurious frequency components, which may affect system performance. Techniques such as dithering, phase truncation, and higher-resolution LUTs are used to minimize these unwanted artifacts.

NCOs are also used in software-defined radios (SDRs) for precise frequency tuning and demodulation. They play a key role in orthogonal frequency-division multiplexing (OFDM) systems, where multiple carriers are generated and modulated digitally. Additionally, NCOs are found in radar systems, medical imaging (MRI), and test equipment where highly accurate and stable frequency generation is required.

In summary, an NCO is a powerful and flexible digital frequency synthesis tool that enables precise waveform generation, frequency modulation, and signal processing. Its digital nature ensures high stability, low phase noise, and easy integration into modern communication and signal processing systems

4.2Implementation and Verification of NCO Module

The design process of a Numerically Controlled Oscillator (NCO) revolves around the concept of phase accumulation and waveform synthesis. The key component is a phase accumulator, which continuously increments a phase value at each clock cycle. The phase increment value determines the frequency of the output waveform. The fundamental equation governing an NCO is:

$$Fout = (Fclk \times \Delta P)/2^N$$

where Fout is the output frequency, Fclk is the system clock frequency, ΔP is the phase increment value, and N is the bit-width of the phase accumulator. By adjusting ΔP, the output frequency can be finely controlled.

The phase output from the accumulator is used to address a Look-Up Table (LUT) containing precomputed values of a sine wave. The LUT maps the accumulated phase to amplitude values, generating a digital sinusoidal waveform. For applications requiring a square wave output, the most significant bit (MSB) of the phase accumulator can be directly used.

To improve spectral purity, techniques like dithering and phase truncation are used to reduce quantization errors and minimize unwanted spurious signals. This digital approach ensures precise, stable, and dynamically tuneable frequency generation, making NCOs highly suitable for communication, signal processing, and waveform synthesis applications.

Block Diagram of NCO Circuit

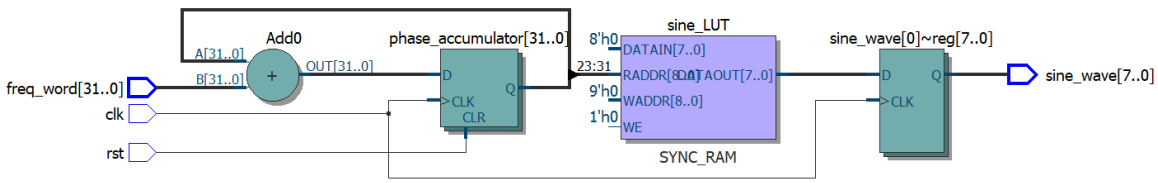


Fig. 4.2 Block Diagram of NCO Module

Frequency Calculation:

The output frequency is given by:

$f_{out} = \Delta P \times f_{clk} / 2^{32}$

- The maximum freq_word is $2^{32}-1$, which is approximately 4.29×10^9.

- Substituting the maximum freq_word:

  fout,max={(2^{32}-1)* fclk /{2^{32}} ≈ fclk
- However, due to the Nyquist limit, the actual maximum usable frequency without aliasing is:

  fout,max≈fclk/2

If your system clock fclkf_{clk}fclk is **100 MHz**, the maximum frequency that can be generated reliably is **50 MHz**.

Verification of NCO Circuit for Different Frequency

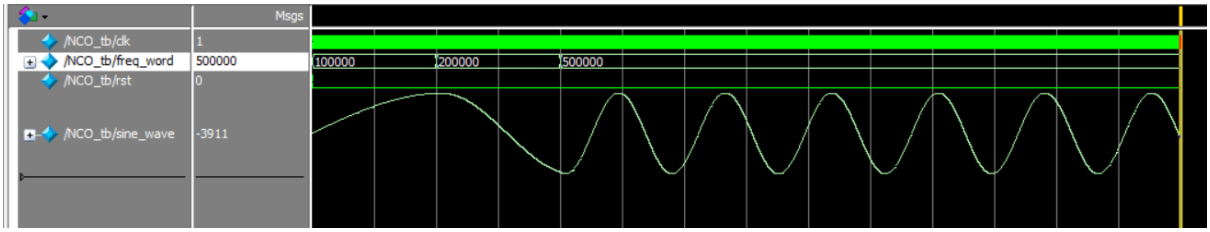


Fig. 4.3 Output of NCO For Random Frequency

Sine wave is generate for freq_word-500000(decimal) a random number for testing

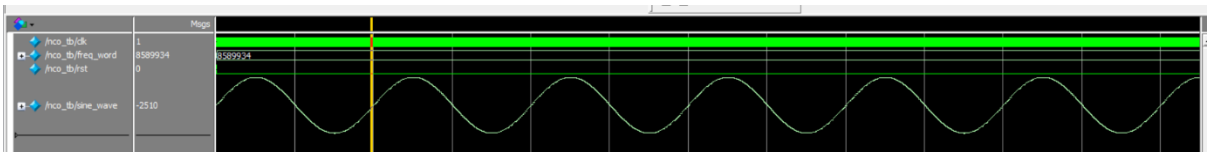


Fig. 4.4 Signal Generation of 100khz

Generates frequency of 100KHz for testing

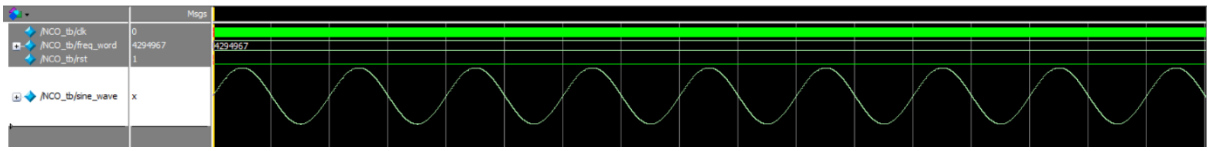


Fig. 4.5 Signal Generation for 50khz

freq_word≈4,294,967(decimal)≈32′h0041C71D(hex) Generated 50KHz signal for testing

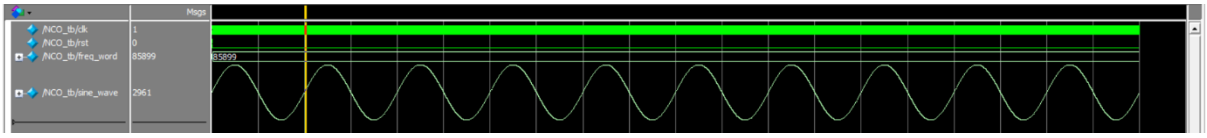


Fig. 4.6 Signal Generation For 1khz

Generated frequency 1Khz for testing the
freq_word≈85,899(decimal)≈32′h00014F13(hex)

Output conclusion:

The NCO design successfully generated sine waves at various frequencies, including 1 kHz, 10 kHz, 50 kHz, and 100 kHz. The lowest achievable frequency was approximately 0.0116 Hz, confirmed through simulation. The testbench effectively verified the design's correctness and frequency generation capabilities.

4.3Implementation and verification of DAC circuit

A **Digital-to-Analog Converter (DAC)** is an essential component in signal processing that converts discrete digital signals into continuous analog signals. The DAC takes a digital input, usually represented in binary, and transforms it into a corresponding voltage or current. The resolution of a DAC, determined by the number of bits in its digital input, dictates the accuracy of the output signal. Higher resolution results in finer quantization levels and reduced signal distortion. Common types of DAC architectures include Binary-Weighted Resistor DACs, R-2R Ladder DACs, and Sigma-Delta DACs**,** each optimized for different applications in audio processing, communication systems, and instrumentation.

A Sigma-Delta DAC (ΣΔ DAC) is a specialized type of DAC that leverages oversampling and noise shaping to achieve high resolution and improved signal fidelity. Unlike conventional DACs that directly convert digital values to analog, a ΣΔ DAC first modulates the input signal using a Sigma-Delta Modulator (SDM)**.** This process increases the effective sampling rate and distributes quantization noise to higher frequencies, where it can be filtered out efficiently. The modulated signal is then passed through a low-pass filter (LPF) to reconstruct a smooth analog waveform.

The key advantage of Sigma-Delta DACs is their ability to achieve high precision using low-resolution components, making them ideal for applications such as high-fidelity audio, precision measurement instruments, and software-defined radios**.** Their use of oversampling

and noise shaping significantly reduces distortion and improves dynamic range, making them one of the most widely used DAC architectures in modern digital-to-analog conversion applications.

Block diagram of DAC

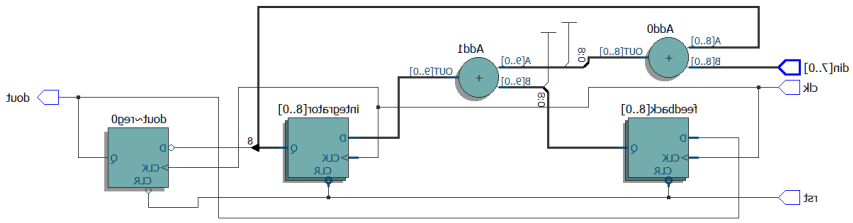


Fig. 4.7 Block Diagram of DAC Module

Verification of DAC

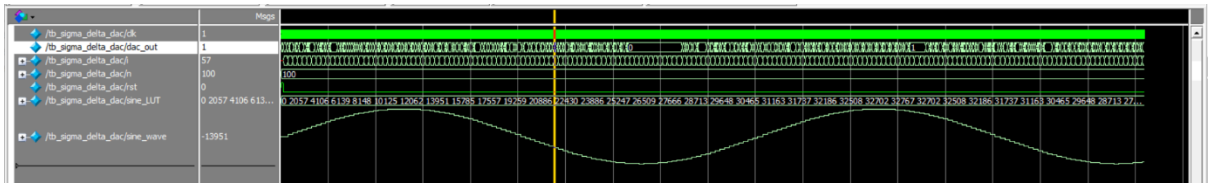


Fig. 4.8 Simulation Result of DAC

Fig.34 shows the output of DAC which gives PWM as output and by passing it form any analog filter, we can observe the proper sine wave

But if we use a digital filter which is used in Digital signal processing it can generate a clock pulse which is as good as clock signal, Now lets design a digital filter

4.4Implementation and verification of digital sigma delta LPF

A digital filter is a fundamental signal processing system that operates on discrete-time signals to modify, enhance, or suppress specific frequency components. These filters play a critical role in various applications such as communication systems, audio processing, control systems, biomedical signal processing, and instrumentation. By applying mathematical operations to digital signals, digital filters help improve signal quality by removing unwanted noise, enhancing signal components, and ensuring efficient data transmission and processing.

Digital filters function by taking an input signal in the form of digital samples, processing these samples using algorithms, and generating a modified output signal. They are implemented using software algorithms running on digital signal processors (DSPs), field-programmable gate arrays (FPGAs), or application-specific integrated circuits (ASICs). One of the major advantages of digital filters over analog filters is their precision and flexibility. Unlike analog filters, which rely on physical components like resistors, capacitors, and inductors, digital filters use arithmetic operations that can be finely controlled and adjusted in real-time. This programmability allows for easy modifications, adaptivity to different applications, and high stability without being affected by temperature variations or component aging.

Digital filters are broadly categorized into two main types: Finite Impulse Response (FIR) filters and Infinite Impulse Response (IIR) filters. FIR filters are characterized by having a finite-duration impulse response, meaning that their output eventually settles to zero after a finite number of time steps. These filters are inherently stable because they do not use feedback, making them suitable for applications requiring linear phase response, such as image and audio processing. FIR filters are typically implemented using convolution operations with a set of predefined filter coefficients.

In contrast, **IIR filters** have an infinite-duration impulse response due to their feedback mechanism. Unlike FIR filters, IIR filters use recursion, where past output values contribute to the current output computation. This feedback structure makes IIR filters computationally efficient, as they require fewer coefficients to achieve a similar frequency response compared to FIR filters. However, the presence of feedback introduces a risk of instability if the filter coefficients are not carefully chosen. Despite this drawback, IIR filters are widely used in applications where computational efficiency and real-time processing are critical, such as audio equalization, biomedical signal filtering, and communication systems.

One of the essential applications of digital filters is in Sigma-Delta (ΣΔ) Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (DACs). These converters employ Sigma-Delta Modulation (SDM), a technique that shifts quantization noise to higher frequencies, allowing the signal's important low-frequency components to be preserved. However, before the final digital signal can be obtained, the high-frequency quantization noise must be filtered out, which is where the Sigma-Delta Low-Pass Filter plays a crucial role.

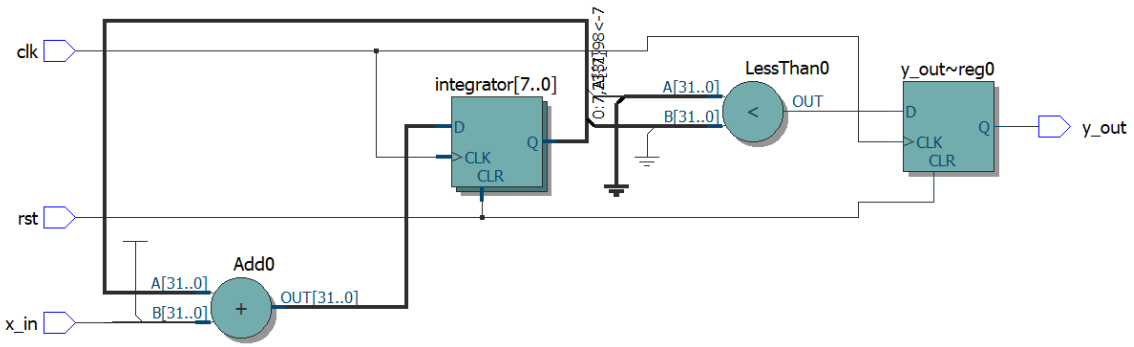


Fig. 4.9 Block Diagram of Filer

4.5FPGA testing of NCO (Altera DE1)

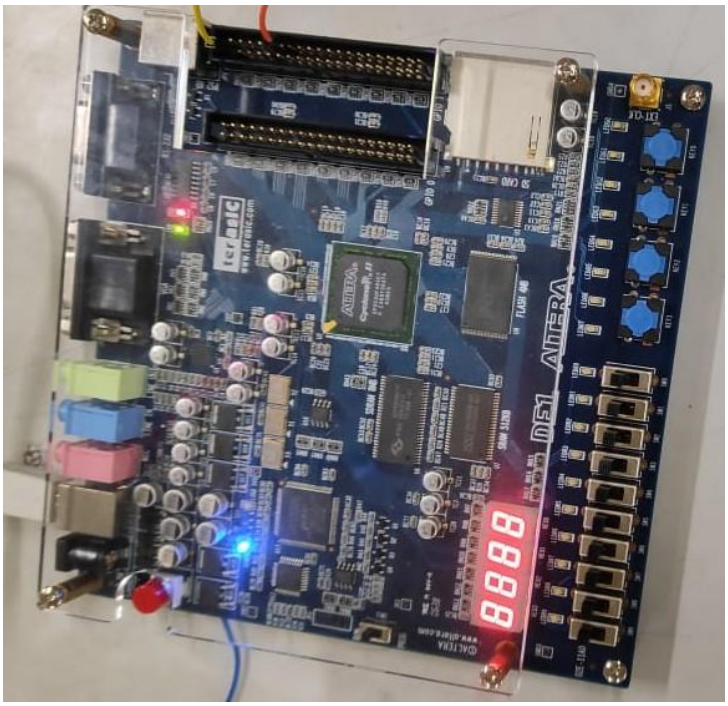


Fig. 4.10 Test setup of FPGA

Top view of alter kit and power on state of this kit in this kit we have set the 10 toggle witch to the frequency word set of NCO circuit and the last switch is reset switch in NCO circuit , we have used internal clock of the of 50MHz frequency signal

NCO testing of FPGA

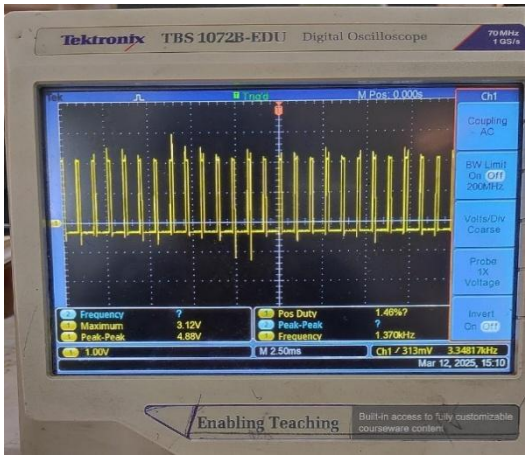


Fig. 4.11 3.3KHz signal

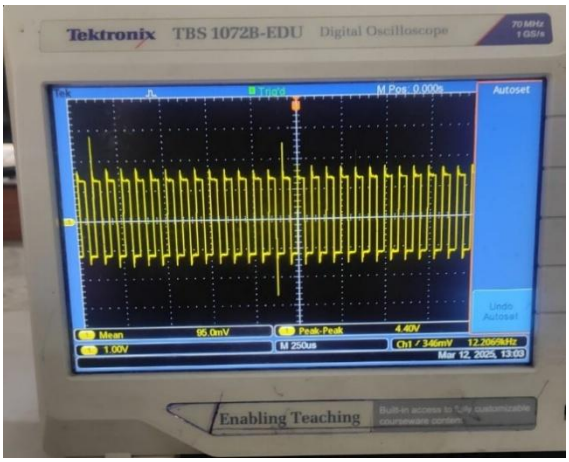


Fig. 4.12 12.2KHz signal

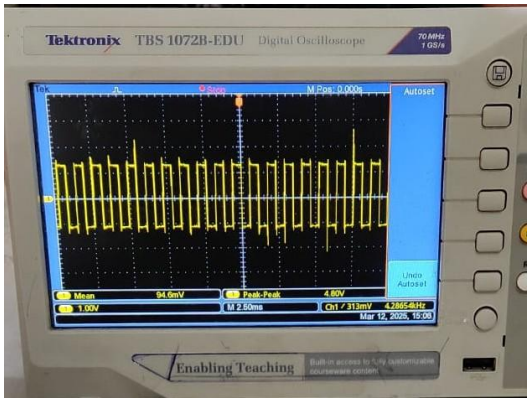


Fig. 4.13 4.5KHz signal

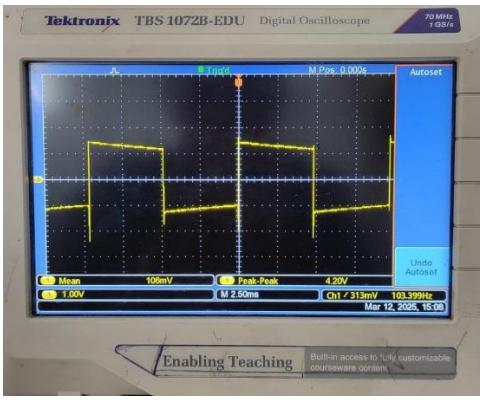


Fig. 4.14 103.3Hz signal

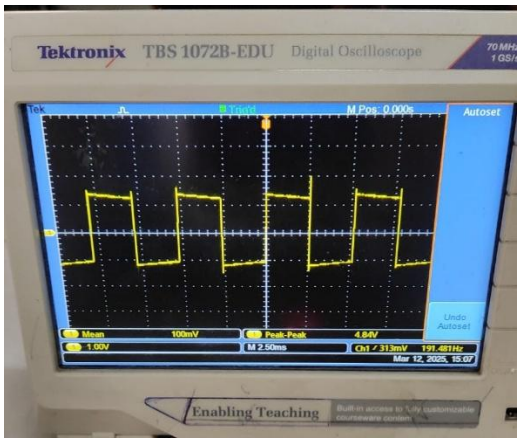


Fig. 4.15 191.4KHz signal

Different frequency response of FPGA, the high frequency nice still includes in this method as we move ahead with higher and higher frequency the response will be affected, and it will destroy the output

4.6 Hybrid CDR

A hybrid Clock and Data Recovery (CDR) module tailored for high-speed serial communication systems. The innovative design integrates a Numerically Controlled Oscillator (NCO) with a Phase-Locked Loop (PLL) to regenerate clock frequencies and recover embedded data. This hybrid approach leverages the digital precision of the NCO while benefiting from the phase-locking capabilities of a PLL, making it especially effective for modern digital applications such as FPGAs and software-defined radios (SDRs).

At the core of the module is the NCO, a digital circuit designed to generate a frequency-controlled clock signal. Unlike traditional analog PLLs, the NCO operates entirely in the digital domain, which offers superior stability and flexibility. In serial communication, an explicit clock is not transmitted along with data; instead, clock information is embedded in the data transitions. To address this, the NCO dynamically adjusts its phase and frequency to lock onto the incoming data stream. This process begins with the continuous detection of data edge transitions—changes from a logical ‘0’ to ‘1’ or vice versa. A phase detector compares these transitions with the generated NCO clock to compute the phase error, which indicates any misalignment between the two signals.

This phase error is then processed through a loop filter that smooths out rapid variations, preventing oscillatory behavior in the control system. The filtered error signal is subsequently used to update the NCO’s phase accumulator, a register that maintains the current phase state. In parallel, the Phase Increment Register (PIR) controls the step size by which the phase accumulator is updated, effectively setting the output frequency. The accumulated phase is then converted into a clock waveform by a phase-to-waveform converter. The relationship between the output frequency and these parameters is defined by the equation $f_{out} = (F_inc \times f_clk) / 2^N$, where F_inc is the phase increment, f_clk is the system clock frequency, and N is the bit-width of the phase accumulator.

The report emphasizes that the dynamic adjustment of the phase increment is crucial, allowing the NCO to lock onto the data clock and track any variations in frequency. Once the clock is accurately recovered, it is used to sample the incoming serial data at optimal points—typically at the center of each bit period—to minimize errors and jitter. The data recovery process is further refined by retiming and filtering the sampled data, thereby reducing noise

and inter-symbol interference. This step ensures that the regenerated output closely approximates the originally transmitted data, even in the presence of channel noise.

The theoretical model presented in the report represents the received signal as a combination of the original binary data and noise. Mathematically, the signal is expressed as $D(t) = A \cdot s(t) + n(t)$, where A represents the signal amplitude, s(t) is the data stream, and n(t) denotes noise. The recovered clock, generated through a function of D(t), is used to sample the noisy signal, after which a decision threshold function is applied to determine the final digital output.

In summary, the hybrid CDR module successfully merges digital clock recovery via an NCO with PLL techniques to achieve precise frequency synthesis and data regeneration. The report's detailed explanation and verification of both the design and theoretical model confirm that this approach effectively synchronizes the recovered clock with the data stream, ensuring reliable and robust communication in high-speed serial systems.

Block diagram of hybrid NCO circuit

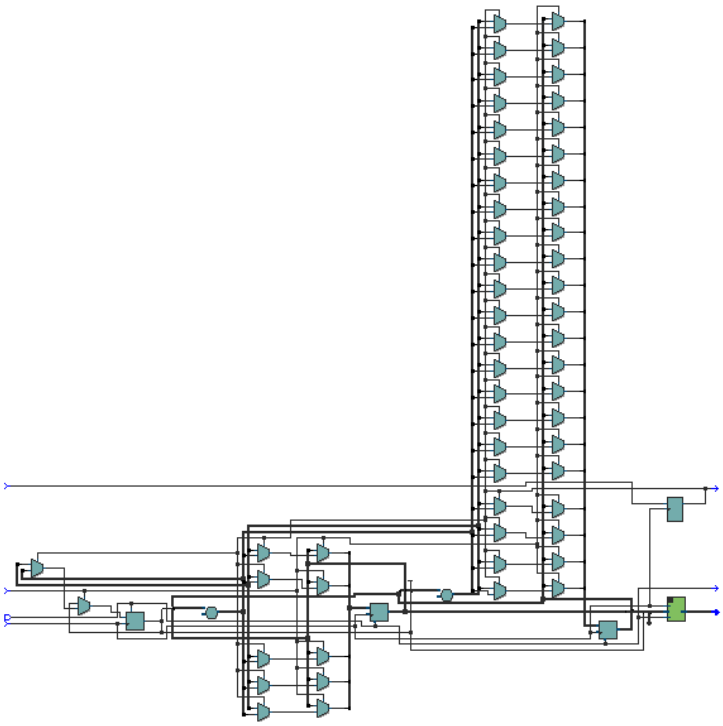


Fig. 4.16 Block Diagram of Hybrid CDR Module

Verification of the hybrid CDR module circuit:

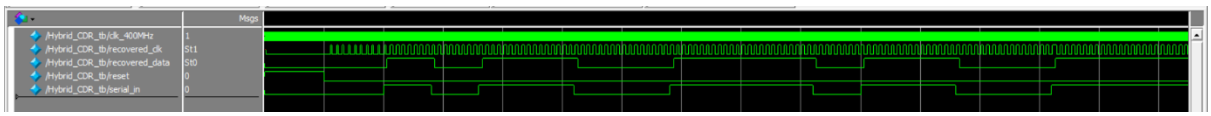


Fig. 4.17 Output of Hybrid CDR Module

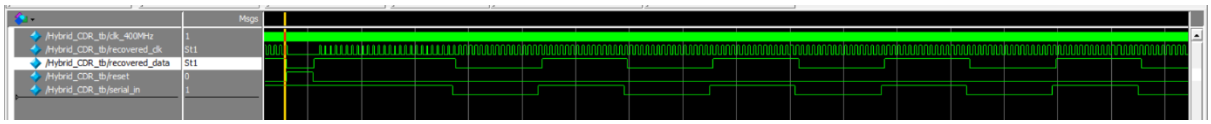


Fig. 4.18 Output of Hybrid CDR Module

- In fig. 5.17 The output of hybrid CDR module shows that the frequency is regenerated with data is recovered, in the fig. 5.15 the reset pulse is applied, and it starts regenerating the frequency from the received data

CHAPTER 5

PHYSICAL DESIGN OF TRANSMITTER AND RECEIVER CIRCUITS

Physical design is a crucial step in the ASIC design flow, translating a synthesized netlist into a layout that meets performance, power, and area (PPA) constraints. It involves key stages like floorplanning, placement, clock tree synthesis, routing, and timing closure. Proper physical design ensures minimal signal delays, optimal power distribution, and efficient utilization of silicon area, directly impacting the chip's performance and manufacturability. Poor physical design can lead to congestion, excessive power consumption, and timing violations, making the chip unreliable. Thus, careful optimization and verification during physical design are essential for achieving a functional and high-performance ASIC.

5.1introduction of ASIC design flow

ASIC design flow starts with requirement analysis and specification, where the chip's functionality, performance targets, power limits, and area constraints are defined. Next, architects develop a high-level system architecture, partitioning the design into functional blocks with clear interfaces. The design then moves to RTL coding, where engineers describe digital behavior using hardware description languages such as Verilog or VHDL. Functional verification via simulation ensures the RTL meets the intended behavior. Once verified, the RTL is synthesized into a gate-level netlist by mapping design constructs to a technology-specific standard cell library while optimizing for timing, area, and power. This netlist becomes the basis for physical design, which begins with floorplanning to allocate chip areas and determine I/O placements. Placement follows, assigning exact positions to standard cells and macros, ensuring no overlap and optimal proximity. Clock tree synthesis then distributes clock signals uniformly, minimizing skew. Routing connects the placed cells according to the netlist, with global and detailed routing stages addressing wiring and congestion. After routing, the design undergoes physical verification, including design rule checking (DRC) and layout versus schematic (LVS) analysis, to confirm manufacturability. Finally, sign-off is achieved with the generation of a GDSII file, leading to tape-out and subsequent fabrication in a semiconductor foundry.

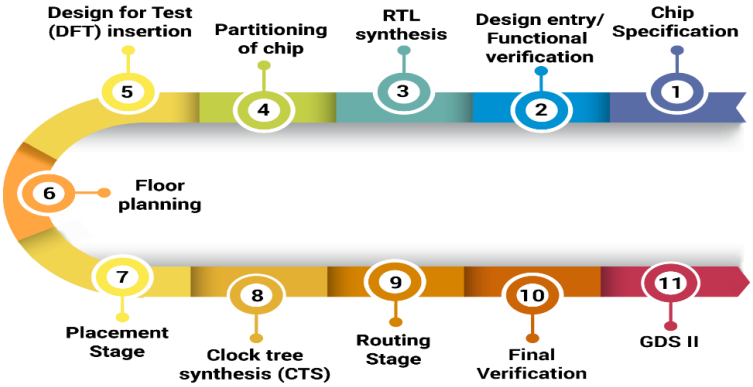


Fig. 5.1 ASIC design flow block diagram

5.2ASIC implementation using cadence EDA tools.

Cadence Design Tools form a comprehensive suite of electronic design automation (EDA) software widely used in the semiconductor industry for designing and verifying integrated circuits. These tools support the entire chip design flow—from front-end design and simulation to back-end physical implementation and sign-off. For example, Cadence's Virtuoso platform is renowned for its capabilities in custom analog and mixed-signal design, while Innovus provides advanced solutions for digital physical design including placement, routing, and timing closure. Additionally, tools like Spectre are extensively used for circuit-level simulation and verification, ensuring that designs meet performance, power, and reliability targets. Overall, Cadence's integrated environment helps engineers streamline complex design processes, reduce time-to-market, and optimize critical parameters such as area and power consumption, making it a preferred choice for high-performance and high-density chip development.

5.3Implementation of embedded clock SerDes based communication system

Verification testing of embedded clock SerDes transmitter system module

Tx module implementation

For verification we use NC lunch software by cadence

To log in NC launch use commands

- csh
- source /home/install/cshrs
- nclaunch -new&

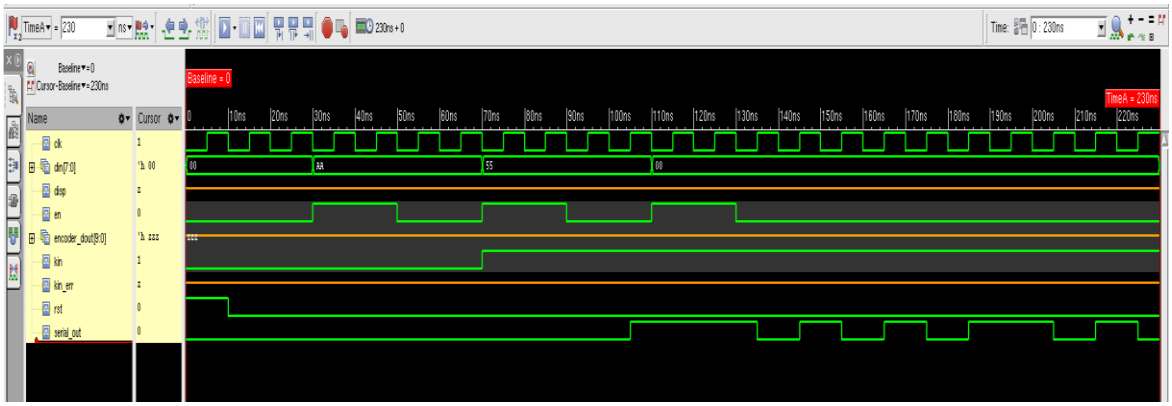


Fig. 5.2 simulation output of transmitter module

Simulation report of transmitter circuit in cadence EDA tools

Enter this command to open coverage analysis -> irun top_module_tx.v top_module_tx_tb.v -access +rwc -coverage all -gui

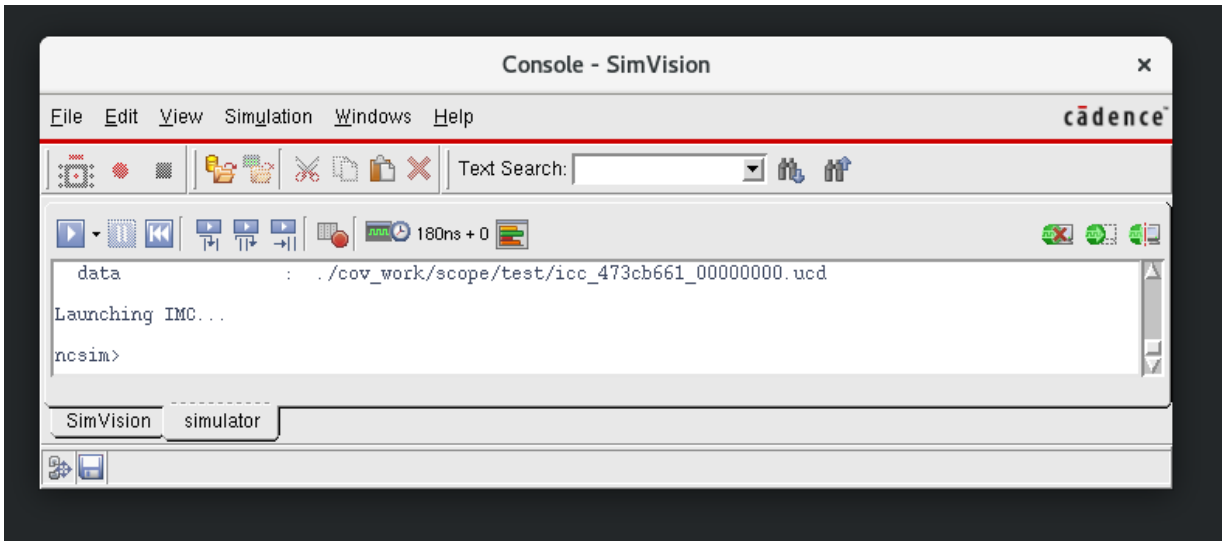


Fig. 5.3 window of NC sim

Analysis window:

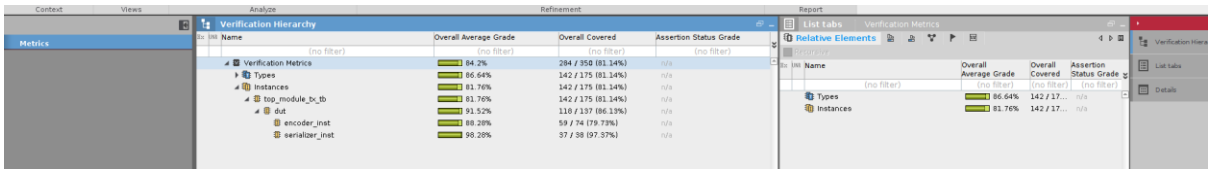


Fig. 5.4 analyzer window

The image shows a verification coverage report in a hardware design tool. It details module coverage percentages, hierarchy, and assertion status, indicating test completeness for different design instances and submodules.

Timing report of the module

fig.5.5 is a timing report from Cadence Genus for transmitter module detailing a clock path's delay, including cell types, fanout, slew, setup, and slack (6365ps), ensuring proper timing closure in ASIC synthesis.

```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 21.14-s082_1
  Generated on:           Mar 19 2025  05:44:49 am
  Module:                 top_module_tx
  Technology library:     slow
  Operating conditions:   slow (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
============================================================

         Pin                Type       Fanout Load Slew Delay Arrival
                                              (fF) (ps)  (ps)   (ps)
-------------------------------------------------------------------------
(clock clk)               launch                                    0 R
serializer_inst
  bit_counter_reg[1]/CK                              80    +0       0 R
  bit_counter_reg[1]/Q    DFFRHQX1          6 14.9  120  +414     414 F
  g838__6783/B                                          +0     414
  g838__6783/Y            NOR2XL            4  8.9  284  +238     652 R
  g818__1666/B                                          +0     652
  g818__1666/Y            NAND2XL           3  7.2  227  +208     860 F
  g814__9315/B                                          +0     860
  g814__9315/Y            NOR2XL            3  5.1  194  +194    1054 R
  g811__4733/AN                                         +0    1054
  g811__4733/Y            NAND2BX1          6 14.2  142  +195    1249 R
  g843/S0                                               +0    1249
  g843/Y                  MXI2XL            1  1.7   95   +72    1322 F
  g797__6783/B                                          +0    1322
  g797__6783/Y            NOR2XL            1  1.7   84   +88    1410 R
  bit_counter_reg[2]/D    DFFRX1                        +0    1410
  bit_counter_reg[2]/CK   setup                     80  +216    1625 R
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(clock clk)               capture                               8000 R
                          uncertainty                     -10   7990 R
-------------------------------------------------------------------------
Cost Group   : 'clk' (path_group 'clk')
Timing slack :    6365ps
Start-point  : serializer_inst/bit_counter_reg[1]/CK
End-point    : serializer_inst/bit_counter_reg[2]/D
```

Fig. 5.5 timing report of transmitter module

Area report-

Fig.5.6 shows the report of the area breakdown for transmitter module listing cell counts and total area for submodules encoder_inst and serializer_inst, with a total area of 1566.026 units.

```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 21.14-s082_1
  Generated on:           Mar 19 2025  05:44:49 am
  Module:                 top_module_tx
  Technology library:     slow
  Operating conditions:   slow (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
============================================================

     Instance        Module     Cell Count  Cell Area  Net Area   Total Area   Wireload
-----------------------------------------------------------------------------------------
top_module_tx                         206   1566.026     0.000     1566.026 <none> (D)
  encoder_inst    encoder_8b10        157   1046.793     0.000     1046.793 <none> (D)
  serializer_inst serializer           49    519.233     0.000      519.233 <none> (D)
  (D) = wireload is default in technology library
```

Fig. 5.6 area report of transmitter module

Power report-

Fig.5.7 showes a power analysis report for the transmitter module instance, showing leakage, internal, and switching power across different categories. The total power consumption is 1.32146e-04 W, with registers consuming 79.12%, followed by logic (14.87%) and clock (6.01%). The report highlights that internal power dominates (82.37%), followed by switching power (11.51%) and leakage power (6.13%). Memory, latches, and other components contribute 0% to power consumption. This breakdown is crucial for optimizing power efficiency in ASIC design, particularly by targeting registers and logic for power reduction techniques.

```
Instance: /top_module_tx
Power Unit: W
PDB Frames: /stim#0/frame#0
  ----------------------------------------------------------------------
    Category         Leakage     Internal    Switching        Total     Row%
  ----------------------------------------------------------------------
      memory     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
    register     5.12078e-06  9.73226e-05  2.10904e-06  1.04552e-04  79.12%
       latch     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
       logic     2.97632e-06  1.15214e-05  5.15803e-06  1.96557e-05  14.87%
        bbox     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
       clock     0.00000e+00  0.00000e+00  7.93800e-06  7.93800e-06   6.01%
         pad     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
          pm     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
  ----------------------------------------------------------------------
    Subtotal     8.09710e-06  1.08844e-04  1.52051e-05  1.32146e-04 100.00%
  Percentage           6.13%       82.37%       11.51%      100.00% 100.00%
  ----------------------------------------------------------------------
```

Fig. 5.7 power report of transmitter module

Constrain file –

Fig. 5.8 shows SDC file defines timing constraints for top_module_tx in Cadence Genus, setting an 8 ns clock period, clock transition (0.08 ns), uncertainty (0.01 ns), and enclosed wire-load mode for accurate ASIC synthesis

```
# ####################################################################

#  Created by Genus(TM) Synthesis Solution 21.14-s082_1 on Wed Mar 19 05:44:48 EDT 2025

# ####################################################################

set sdc_version 2.0

set_units -capacitance 1000fF
set_units -time 1000ps

# Set the current design
current_design top_module_tx

create_clock -name "clk" -period 8.0 -waveform {0.0 4.0} [get_ports clk]
set_clock_transition 0.08 [get_clocks clk]
set_clock_gating_check -setup 0.0
set_wire_load_mode "enclosed"
set_clock_uncertainty -setup 0.01 [get_ports clk]
set_clock_uncertainty -hold 0.01 [get_ports clk]
```

Fig. 5.8 constrain file of transmitter module

Gui of synthesise

Fig5.9 shows the RTL view of transmitter module

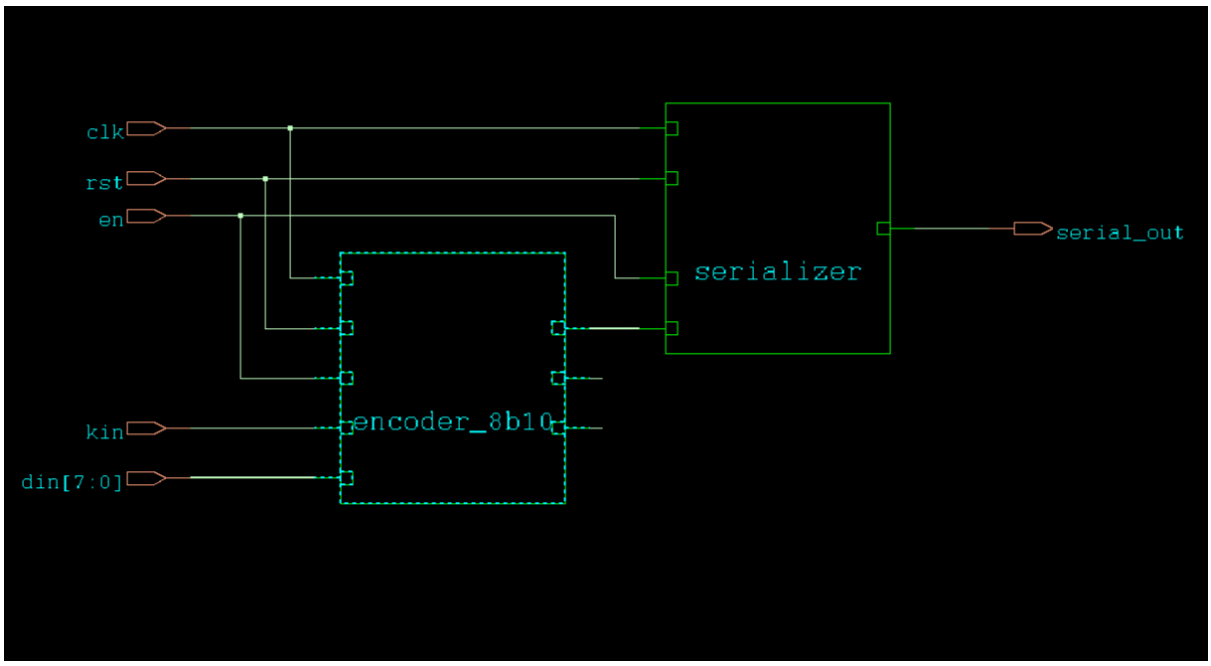


Fig. 5.9 gate level RTL simulation of transmission

Floor planning and placement output:

Fig.5.10 shows the power planning placement and floor planning of the transmission module

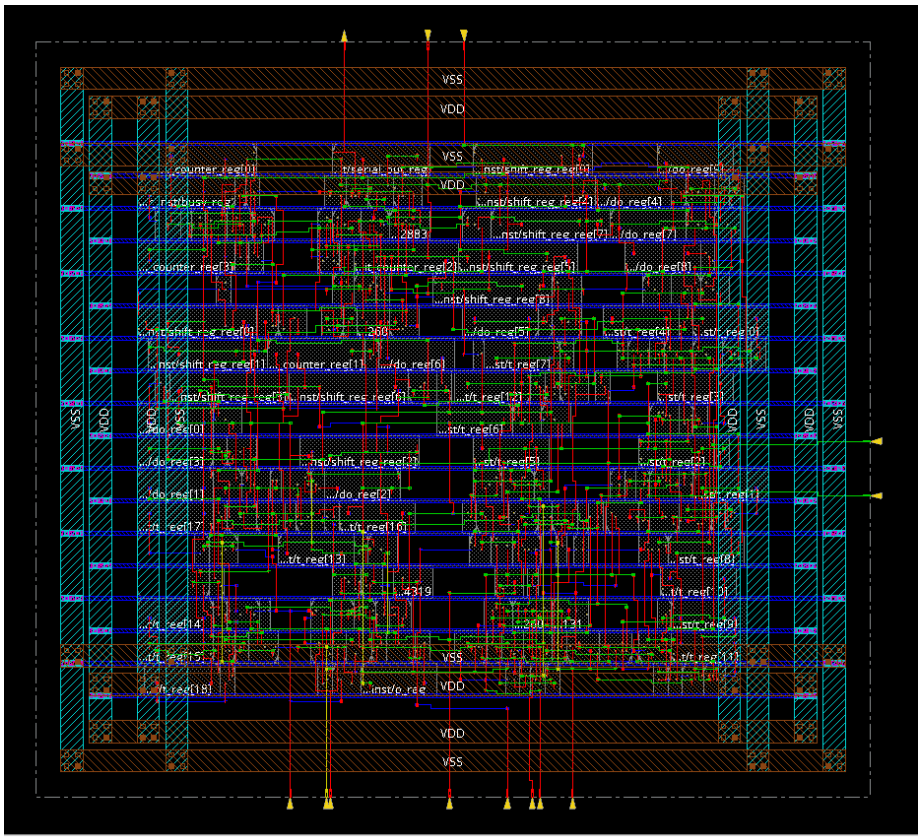


Fig. 5.10 first floor planning report of transmitter module

CTS optimization:

The fig.5.11 shows the optimized CTS debugger window of the transmitter

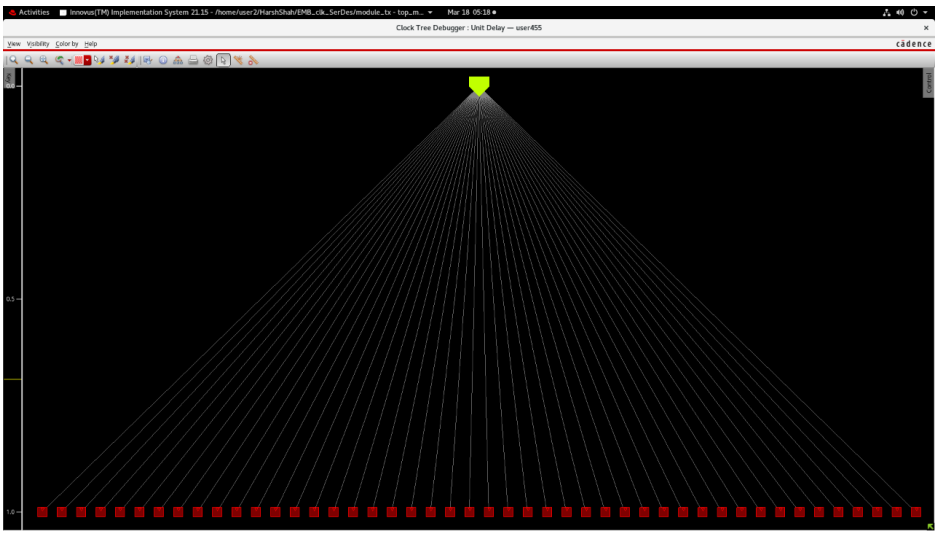


Fig. 5.11 CTS optimized output

Post CTS setup and hold report:

This timing summary shows no violations, with WNS of 5.248 ns (setup) and 0.003 ns (hold). Design density is 79.05%, with zero routing overflow and no violations in capacitance, transition, fanout, or length constraints.

```
------------------------------------------------------------------
     optDesign Final Summary
------------------------------------------------------------------

Setup views included:
 setup
Hold views included:
 hold

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  5.248  |  6.160  |  5.248  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   82    |   56    |   62    |
+--------------------+---------+---------+---------+

+--------------------+---------+---------+---------+
|     Hold mode      |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.003  |  0.041  |  0.003  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   82    |   56    |   62    |
+--------------------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 79.059%
Routing Overflow: 0.00% H and 0.00% V
```

Fig. 5.12 CTS optimized report

Post routing report:

Filler cells in physical design ensure continuous power and ground connectivity between standard cells, preventing IR drop and signal integrity issues. They maintain design rule constraints, improve manufacturability, and fill gaps left after placement. Proper filler insertion avoids open circuits in the power distribution network, ensuring reliable chip operation.

While optimizing the design some standers cells are added to reduce the negative slack in the timing report of CTS and then at time of completion of routing, we again add some filler cells to remove the errors that can accord while performing the fabrication

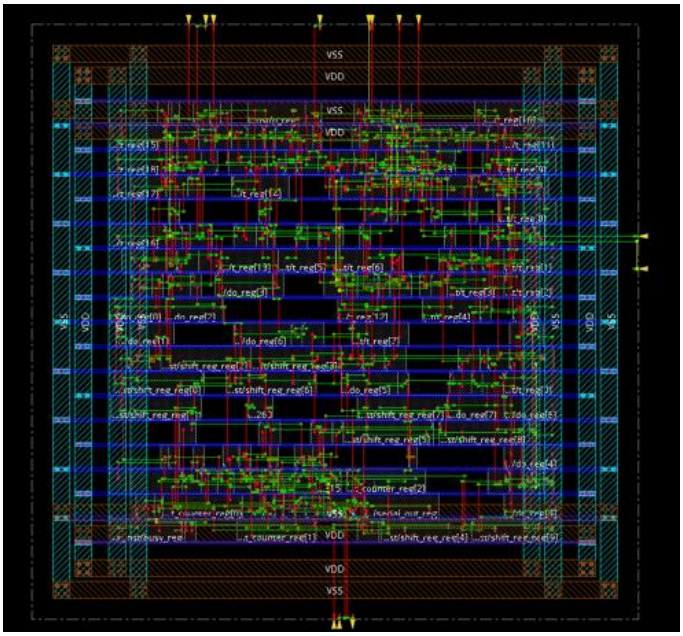


Fig. 5.13 post routing report

Post routing optimization report:

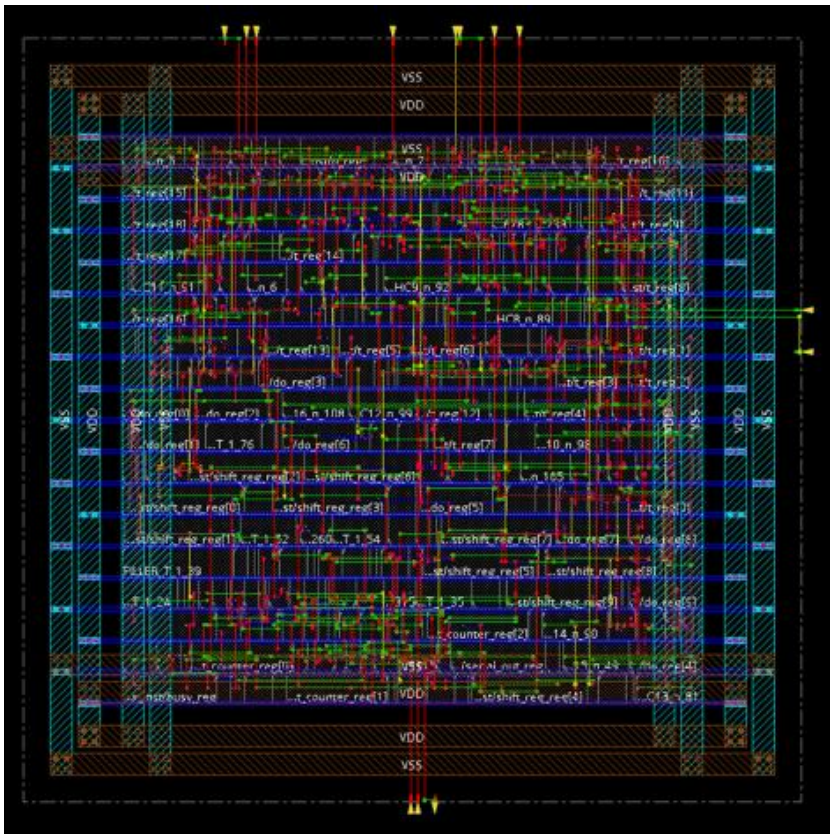


Fig. 5.14 post routing and sign off optimized output

Verification testing of embedded clock SerDes receiver system module

Tx module implementation

For verification we use NC lunch software by cadence

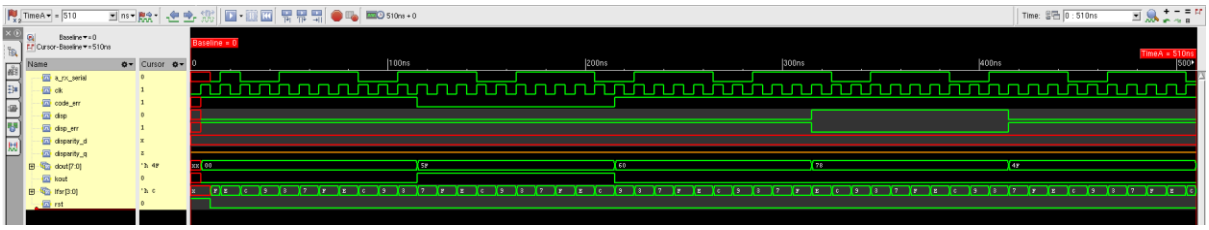


Fig. 5.15 simulation report of receiver module

Enter this command to open coverage analysis -> irun top_module_rx.v top_module_rx_tb.v -access +rwc -coverage all -gui

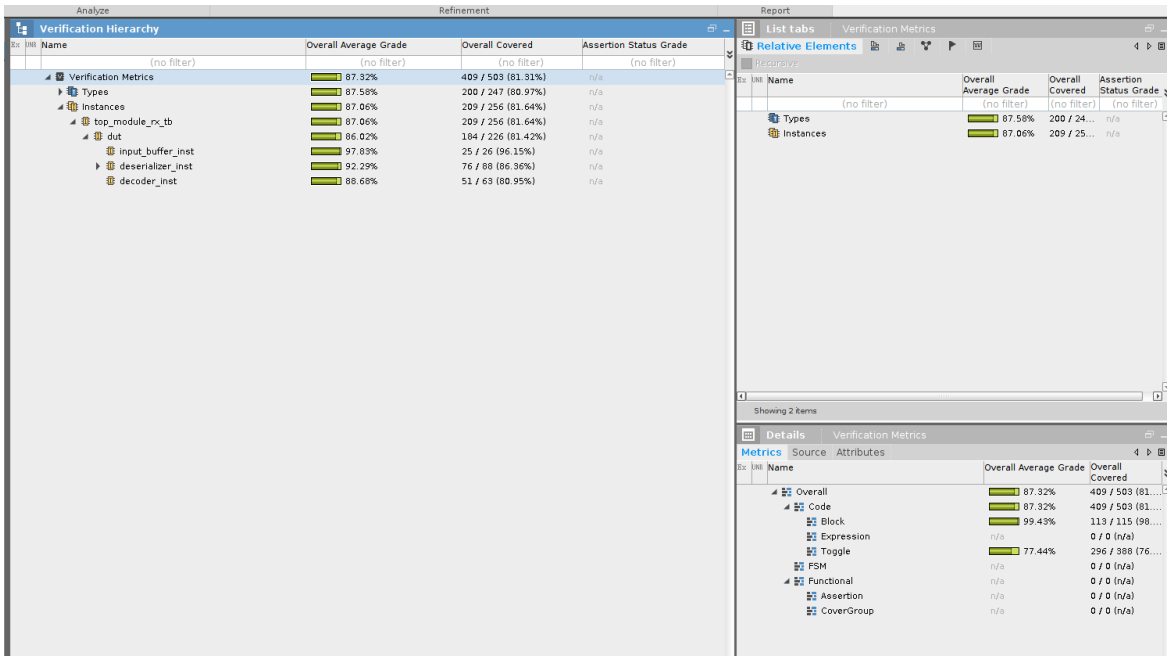


Fig. 5.16 verification coverage output

The image shows a verification coverage report for a hardware design, displaying hierarchical module coverage percentages, assertion status, and functional/code metrics, indicating test completeness for different design instances and submodules.

Timing report of the module

The fig.5.17 showes, detailing a clock path's launch-to-capture delay. It includes cell types, fanout, slew, setup time, and slack, verifying timing closure in ASIC synthesis.

```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 21.14-s082_1
  Generated on:           Mar 18 2025  05:34:18 am
  Module:                 top_module_rx
  Technology library:     slow
  Operating conditions:   slow (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
============================================================

           Pin                      Type       Fanout Load Slew Delay Arrival
                                                     (fF) (ps)  (ps)   (ps)
-------------------------------------------------------------------------------
(clock clk)                       launch                                    0 R
deserializer_inst
  c_parallel_out_reg[8]/CK                                 80    +0       0 R
  c_parallel_out_reg[8]/Q         SDFFRHQX1         5 11.4   98  +385     385 F
deserializer_inst/c_parallel_out[8]
decoder_inst/din[8]
  g8902/A                                                         +0     385
  g8902/Y                         CLKINVX1          3  7.1   77   +84     470 R
  g8891__5122/B                                                   +0     470
  g8891__5122/Y                   NAND2X1           2  4.4   83   +82     552 F
  g8883/A                                                         +0     552
  g8883/Y                         CLKINVX1          5 11.5  106  +104     656 R
  g2/AN                                                           +0     656
  g2/Y                            NAND3BX1          1  2.8   71  +119     776 R
  g8813__6161/C                                                   +0     776
  g8813__6161/Y                   NAND3X1           4 10.1  246  +194     970 F
  g8807__6131/C                                                   +0     970
  g8807__6131/Y                   NAND3BX1          4  9.1  142  +146    1115 R
  g8781__9315/B                                                   +0    1115
  g8781__9315/Y                   NOR2XL            2  4.6   90   +88    1204 F
  g8763__3680/A1                                                  +0    1204
  g8763__3680/Y                   AOI21X1           4 11.4  218  +189    1392 R
  g8737__1881/C0                                                  +0    1392
  g8737__1881/Y                   OAI211X1          1  3.7  163  +158    1550 F
  g8723__3680/S0                                                  +0    1550
  g8723__3680/Y                   MXI2XL            1  1.6   94  +157    1707 F
  g8712__5107/A0                                                  +0    1707
  g8712__5107/Y                   OAI21XL           1  1.6  180  +109    1816 R
  do_reg[4]/D                <<<  DFFQX1                          +0    1816
  do_reg[4]/CK                    setup                    80  +163    1979 R
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(clock clk)                       capture                              2000 R
                                  uncertainty                   -10    1990 R
-------------------------------------------------------------------------------
Cost Group   : 'clk' (path_group 'clk')
Timing slack :      11ps
Start-point  : deserializer_inst/c_parallel_out_reg[8]/CK
End-point    : decoder_inst/do_reg[4]/D
```

Fig. 5.17 timing report of receiver

Area report:

This image shows a synthesis report generated by Cadence Genus Synthesis Solution for the module top_module_rx. It details the cell count, cell area, and total area for different instances within the design. The total area of the design is 1775.687 units, with decoder_inst (decodePipe) consuming the most area (1015.003 units), followed by deserializer_inst (717.541 units). Other instances like cdr_inst and input_buffer_inst occupy minimal area. The wireload mode is enclosed, and the technology library operates under slow conditions. This report helps in design optimization, area reduction, and performance tuning before final implementation.

```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 21.14-s082_1
  Generated on:           Mar 18 2025  05:34:18 am
  Module:                 top_module_rx
  Technology library:     slow
  Operating conditions:   slow (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
============================================================

      Instance         Module     Cell Count  Cell Area  Net Area   Total Area   Wireload
-----------------------------------------------------------------------------------------------
top_module_rx                          235   1775.687     0.000     1775.687 <none> (D)
  decoder_inst      decodePipe         185   1015.003     0.000     1015.003 <none> (D)
  deserializer_inst deserializer        47    717.541     0.000      717.541 <none> (D)
    cdr_inst        CDR                  3     43.143     0.000       43.143 <none> (D)
  input_buffer_inst inputBuffer          3     43.143     0.000       43.143 <none> (D)
  (D) = wireload is default in technology library
```

Fig. 5.18 area report of receiver module

Power report:

Fig.5.19 shows power analysis report for receiver module showing leakage, internal, and switching power for various categories. Registers dominate power consumption (74.85%), followed by logic (20.04%), while clock switching contributes 5.11%.

```
Instance: /top_module_rx
Power Unit: W
PDB Frames: /stim#0/frame#0
  ----------------------------------------------------------------------------
    Category          Leakage     Internal    Switching        Total     Row%
  ----------------------------------------------------------------------------
       memory     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
     register     5.44101e-06  4.62157e-04  1.40645e-05  4.81662e-04  74.85%
        latch     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
        logic     3.23385e-06  8.49162e-05  4.08405e-05  1.28991e-04  20.04%
         bbox     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
        clock     0.00000e+00  0.00000e+00  3.28860e-05  3.28860e-05   5.11%
          pad     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
           pm     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
  ----------------------------------------------------------------------------
     Subtotal     8.67486e-06  5.47073e-04  8.77910e-05  6.43539e-04 100.00%
  Percentage           1.35%       85.01%       13.64%      100.00% 100.00%
  ----------------------------------------------------------------------------
```

Fig. 5.19 power report of receiver module

Constrain file:

The fig. 5.20 shows the constrain file of the receiver module which has a capacitance of 1000fF and timing of 1000ps it also shows the uncertainty of clock frequencyThis image contains a Synopsys Design Constraints (SDC) file generated by Cadence Genus Synthesis Solution for the design top_module_rx. It defines constraints for timing, clock, and signal characteristics. The clock "clk" has a 2.0 ns period with a waveform from 0.0 to 1.0 ns. The clock transition is 0.08 ns, and clock gating check is disabled. The wire load mode is set to "enclosed" for interconnect estimation. Clock setup and hold uncertainties are both set to 0.01

ns, ensuring margin for variations. These constraints help guide timing-driven synthesis and optimization for an efficient digital circuit design.

```
# ##################################################################

#  Created by Genus(TM) Synthesis Solution 21.14-s082_1 on Tue Mar 18 05:34:18 EDT 2025

# ##################################################################

set sdc_version 2.0

set_units -capacitance 1000fF
set_units -time 1000ps

# Set the current design
current_design top_module_rx

create_clock -name "clk" -period 2.0 -waveform {0.0 1.0} [get_ports clk]
set_clock_transition 0.08 [get_clocks clk]
set_clock_gating_check -setup 0.0
set_wire_load_mode "enclosed"
set_clock_uncertainty -setup 0.01 [get_ports clk]
set_clock_uncertainty -hold 0.01 [get_ports clk]
```

Fig. 5.20 constrain file of receiver module

Gui report of the module

Fig. 5.21 shows the syntheses RTL output of the receiver module

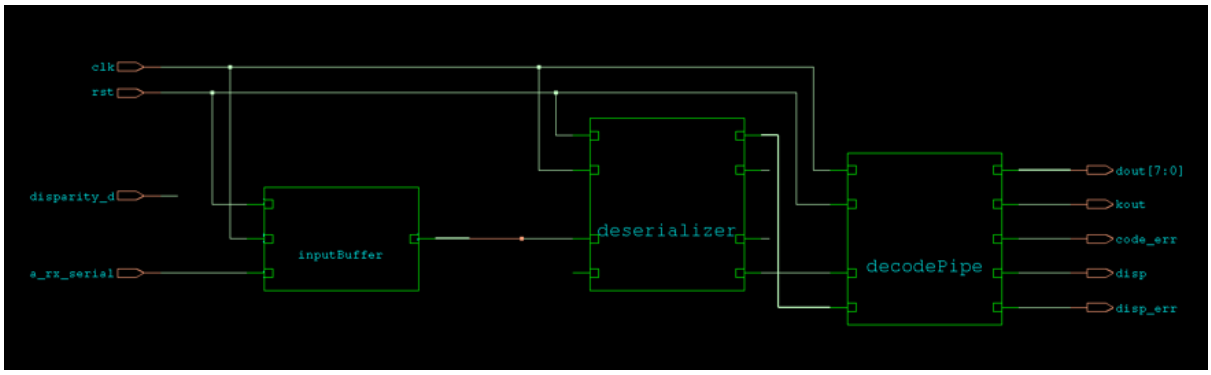


Fig. 5.21 RTL synthesis of receiver module

Power planning and floor planning report:

Fig. 5.22 shoes the power plaining placement and flower planning of the receiver module This image appears to be a standard cell layout or a custom IC layout from a VLSI physical design tool. It represents the final stage of chip implementation, where the placement and routing (P&R) of standard cells and interconnections are finalized. The metal layers, polysilicon, vias, and diffusion layers are visible, along with power (VDD) and ground (VSS) rails at the top and bottom. The colored lines indicate different routing layers used for signal interconnects. The yellow arrows at the periphery suggest I/O ports for communication. This layout is a GDSII representation, crucial for fabrication in semiconductor manufacturing.

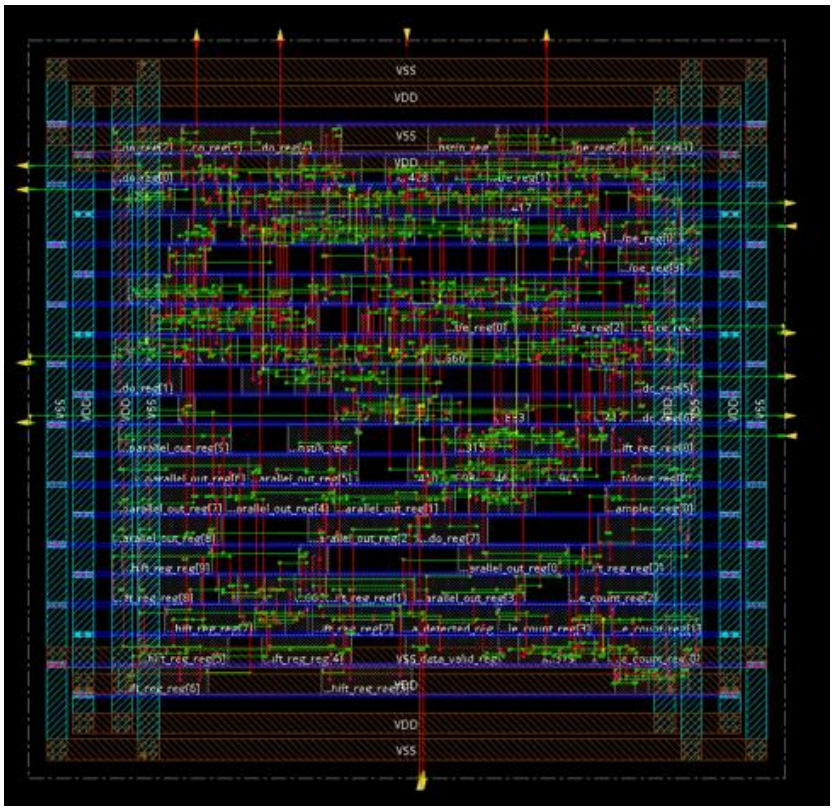


Fig. 5.22 power and floor planning of receiver

CTS graph

Fig.5.23 shows the CTS optimized graph of CTS debugger

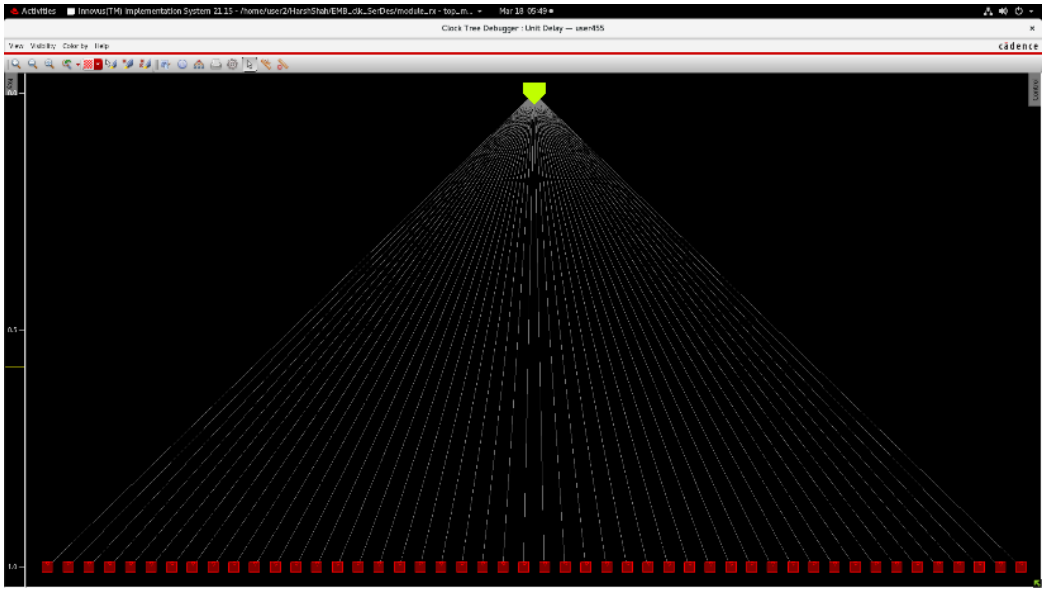


Fig. 5.23 CTS optimized report of receiver module

Post CTS report:

Fig.5.24final summary report shows setup and hold timing analysis. The worst negative slack (WNS) is positive, indicating no violations. No design rule violations (DRVs) exist. Density is 73.327%, with 0% routing overflow.

```
------------------------------------------------------------------
     optDesign Final Summary
------------------------------------------------------------------

Setup views included:
 setup
Hold views included:
 hold

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.011  |  0.011  |  0.191  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   98    |   67    |   49    |
+--------------------+---------+---------+---------+

+--------------------+---------+---------+---------+
|     Hold mode      |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.003  |  0.003  |  0.004  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   98    |   67    |   49    |
+--------------------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 73.327%
Routing Overflow: 0.00% H and 0.00% V
------------------------------------------------------------------
```

Fig. 5.24 post CTS report

Post route optimized output:

The setup timing report shows no violations, with WNS at 0.012 ns and TNS at 0.000 ns. No design rule violations (DRV) exist, and cell density is 74.368%, ensuring efficient physical design utilization.

```
Setup views included:
 setup

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.012  |  0.012  |  0.097  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   98    |   67    |   49    |
+--------------------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 74.368%
------------------------------------------------------------------
```

Fig. 5.25 post routing optimized setup report

```
------------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------------

Hold views included:
 hold

+--------------------+---------+---------+---------+
|     Hold mode      |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.002  |  0.002  |  0.002  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   98    |   67    |   49    |
+--------------------+---------+---------+---------+

Density: 74.368%
------------------------------------------------------------------
```

Fig. 5.26 post routing optimized hold report

Final propre layout presentation with filler:

Final optimized output and GDS file:

Fig 5.27 shows the final routing optimized and with filler output of the receiver block which is saved using the extenuation. GDS

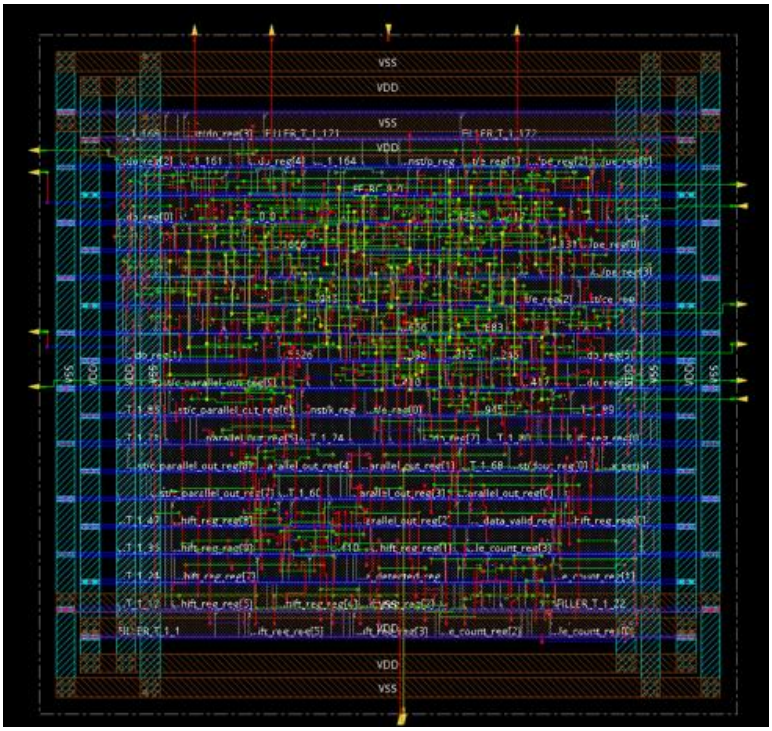


Fig. 5.27 post routing optimized report

5.4Implementation of parallel clock SerDes based communication system

Transmitter system

Verification:

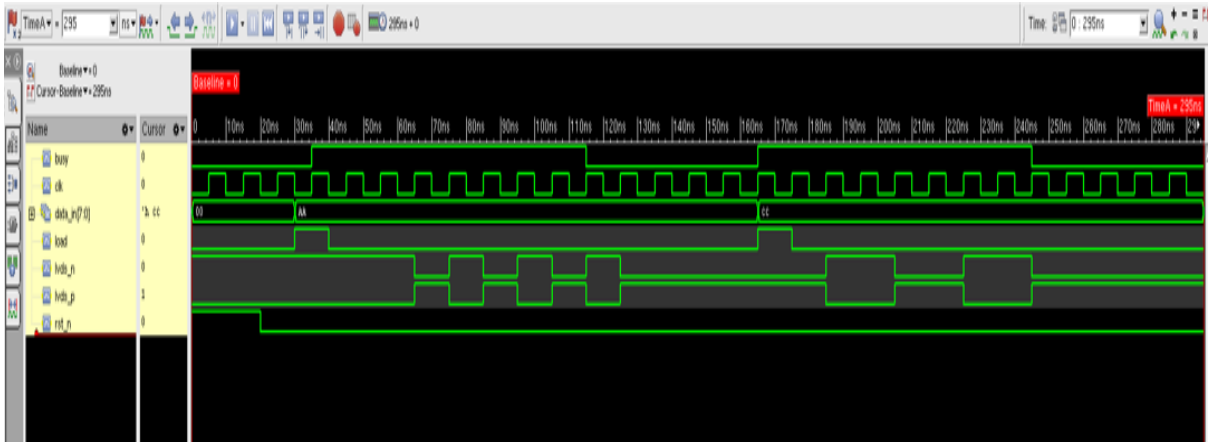


Fig. 5.28 simulation output of parallel clock transmitter module

Enter this command to open coverage analysis -> irun Parallel_Clock_Serializer.v Parallel_Clock_Serializer_tb.v -access +rwc -coverage all -gui

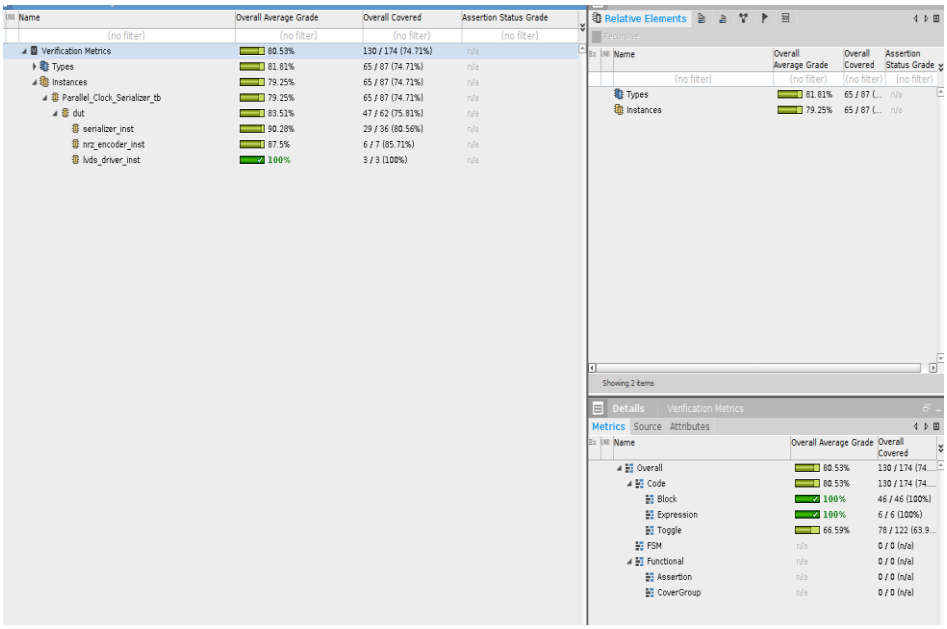


Fig. 5.29 verification coverage report

The image displays a verification coverage report for a hardware design, showing hierarchical module coverage, assertion status, and code metrics, indicating test completion percentages for different design instances and submodules.

Synthesis reports Area

Fig. 5.30 shows the araea report It operates under slow conditions with enclosed wireload mode. The design includes NRZ_Encoder (2 cells, 22.707 area) and Serializer8bit (43 cells, 416.295 area), contributing to a total cell area of 441.273. No net area is utilized.

```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 21.14-s082_1
  Generated on:           Mar 17 2025  05:50:04 am
  Module:                 Parallel_Clock_Serializer
  Technology library:     slow
  Operating conditions:   slow (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
============================================================

          Instance              Module      Cell Count  Cell Area  Net Area   Total Area   Wireload
-----------------------------------------------------------------------------------------------------
Parallel_Clock_Serializer                          46    441.273     0.000      441.273 <none> (D)
  nrz_encoder_inst        NRZ_Encoder               2     22.707     0.000       22.707 <none> (D)
  serializer_inst         Serializer8bit           43    416.295     0.000      416.295 <none> (D)
  (D) = wireload is default in technology library
```

Fig. 5.30 area report

Power report:

Fig.5. power analysis report for an instance named Parallel_Clock_Serializer in a digital circuit design. It breaks down power consumption into leakage, internal, and switching power across different circuit categories like registers, logic, clock, and memory. Registers consume 83% of total power, mainly through internal power dissipation. Logic contributes 12.47%, while clock elements use 4.53%. The total power consumption is 4.14874e-05 W, with 85.05% from internal power, 10.20% from switching, and 4.75% from leakage. This report helps in power optimization by identifying major contributors to power dissipation in the design.

```
Instance: /Parallel_Clock_Serializer
Power Unit: W
PDB Frames: /stim#0/frame#0
  ----------------------------------------------------------------------
    Category         Leakage     Internal    Switching        Total    Row%
  ----------------------------------------------------------------------
      memory     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
    register     1.48381e-06  3.18882e-05  1.06202e-06  3.44340e-05  83.00%
       latch     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
       logic     4.86464e-07  3.39806e-06  1.28966e-06  5.17419e-06  12.47%
        bbox     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
       clock     0.00000e+00  0.00000e+00  1.87920e-06  1.87920e-06   4.53%
         pad     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
          pm     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
  ----------------------------------------------------------------------
    Subtotal     1.97027e-06  3.52863e-05  4.23088e-06  4.14874e-05 100.00%
  Percentage           4.75%       85.05%       10.20%      100.00% 100.00%
  ----------------------------------------------------------------------
```

Fig. 5.31 power report

Timing report:

The fig. 5.32 shows timing analysis of the Parallel_Clock_Serializer module. It shows a timing slack of 8748ps, with the critical path ending at shift_reg_reg[0]/D, and a launch clock delay of 0ps.

```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 21.14-s082_1
  Generated on:           Mar 17 2025  05:50:04 am
  Module:                 Parallel_Clock_Serializer
  Technology library:     slow
  Operating conditions:   slow (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
============================================================

        Pin                  Type       Fanout Load Slew Delay Arrival
                                               (fF) (ps)  (ps)   (ps)
-------------------------------------------------------------------------
(clock clk)                launch                                     0 R
serializer_inst
  busy_reg/CK                                            80    +0       0 R
  busy_reg/Q               DFFRX1          14 27.9  216  +489     489 F
  g693__4733/B                                            +0     489
  g693__4733/Y             NOR2XL           8 12.9  404  +343     832 R
  g686__8246/A1                                           +0     832
  g686__8246/Y             AOI222XL         1  1.6  229  +190    1022 F
  g680/A                                                  +0    1022
  g680/Y                   INVXL            1  2.3   91   +94    1116 R
  shift_reg_reg[0]/D  <<<  DFFRHQX1                        +0    1116
  shift_reg_reg[0]/CK      setup                          80  +127    1242 R
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(clock clk)                capture                                  10000 R
                           uncertainty                            -10    9990 R
-------------------------------------------------------------------------
Cost Group   : 'clk' (path_group 'clk')
Timing slack :    8748ps
Start-point  : serializer_inst/busy_reg/CK
End-point    : serializer_inst/shift_reg_reg[0]/D
```

Fig. 5.32 timing report

Floor planning report

Fig 5.33 shows the power planning, placement and flower planning report of the parallel clock serializer

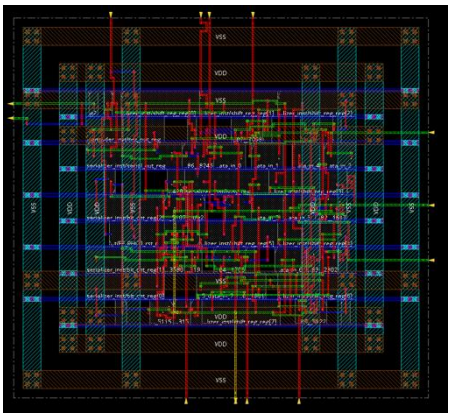


Fig. 5.33 power and floor planning report

Post CTS optimization report:

Fig.5.34 shows Final SI Timing Summary shows successful setup and hold timing with zero violations. The Worst Negative Slack (WNS) is positive, and no paths are violating timing constraints. The design has a density of 82.796%.

```
------------------------------------------------------------------
     optDesign Final SI Timing Summary
------------------------------------------------------------------

Setup views included:
 setup
Hold views included:
 hold

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  8.422  |  8.422  |  8.924  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   30    |   16    |   26    |
+--------------------+---------+---------+---------+

+--------------------+---------+---------+---------+
|     Hold mode      |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.006  |  0.011  |  0.006  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   30    |   16    |   26    |
+--------------------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 82.796%
------------------------------------------------------------------
```

Fig. 5.34 post CTS optimized output

This image represents a Final SI Timing Summary from an Static Timing Analysis (STA) tool, typically used in ASIC or FPGA design flows. It provides detailed information about setup and hold timing constraints, along with Design Rule Violations (DRVs) and overall design density.

In the setup timing section, the Worst Negative Slack (WNS) is 8.422 ns (for reg-to-reg) and 8.924 ns (for default paths), while the Total Negative Slack (TNS) is 0.000 ns, meaning there are no setup timing violations. Additionally, the number of violating paths is zero, confirming that all paths meet the setup timing constraints.

The hold timing section shows that the WNS is 0.000 ns (except for reg-to-reg, which has a small slack of 0.011 ns), and TNS remains at 0.000 ns, with no violating paths. This indicates the design does not have hold timing violations.

The DRVs (Design Rule Violations) section lists max capacitance, max transition, max fanout, and max wire length, all of which have zero violations, meaning the design adheres to physical design constraints.

Finally, the design density is 82.796%, reflecting how efficiently the available FPGA or ASIC resources are utilized. This report confirms a timing-clean design with no violations.

Post sign of GDS report

Fig.5.35 final optimized and filled output of parallel clock serializer system

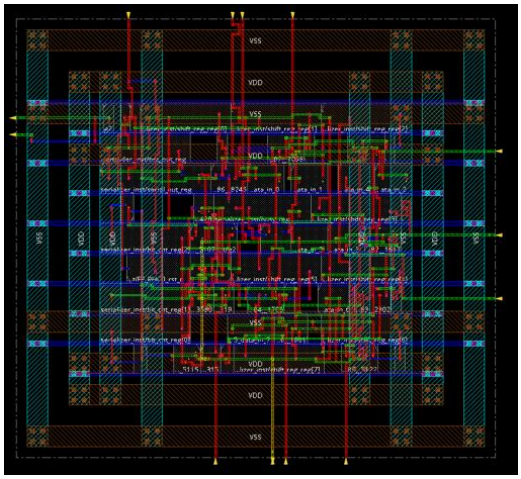


Fig. 5.35 post routing output

receiver system

testbench verification report

fig.5.36 shows the testbench verification of parallel clock deserializer system

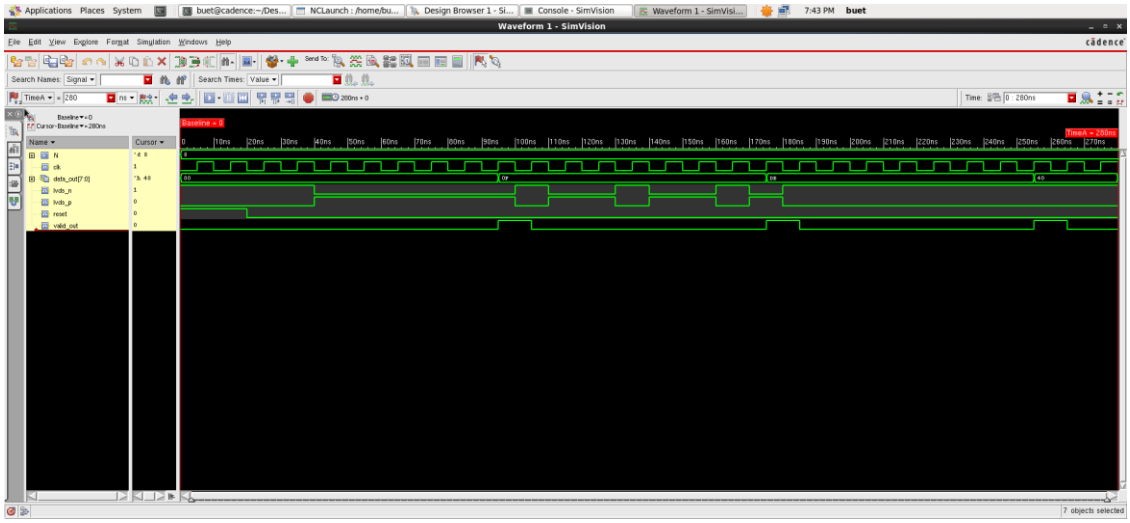


Fig. 5.36 simulation report of parallel clock SerDes based receiver

Synthesis timing report

Fig. 5.37 shows the report for Parallel_Clock_Deserializer shows a timing slack of 599ps. The critical path starts at bit_count_reg[0]/CK and ends at parallel_out_reg[6]/SE, with a launch clock delay of 0ps.

```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 21.14-s082_1
  Generated on:           Mar 17 2025  06:19:55 am
  Module:                 Parallel_Clock_Deserializer
  Technology library:     slow
  Operating conditions:   slow (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
============================================================

          Pin                    Type       Fanout Load Slew Delay Arrival
                                                   (fF) (ps)  (ps)   (ps)
-------------------------------------------------------------------------
(clock clk)                   launch                                    0 R
deserializer_inst
  bit_count_reg[0]/CK                                   80    +0       0 R
  bit_count_reg[0]/Q          DFFRX1            2  3.2   78  +361     361 R
  g124__4319/A                                                +0     361
  g124__4319/Y                NAND2XL           3  6.1  167  +141     502 F
  g118__5107/B                                                +0     502
  g118__5107/Y                NOR2X1           10 43.2  705  +539    1042 R
  parallel_out_reg[6]/SE <<<  SDFFRHQX1                       +0    1042
  parallel_out_reg[6]/CK      setup                     80  +349    1391 R
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(clock clk)                   capture                                2000 R
                              uncertainty                      -10   1990 R
-------------------------------------------------------------------------
Cost Group   : 'clk' (path_group 'clk')
Timing slack :     599ps
Start-point  : deserializer_inst/bit_count_reg[0]/CK
End-point    : deserializer_inst/parallel_out_reg[6]/SE
```

Fig. 5.37 timing report

Power report:

fig. 5.38 Parallel_Clock_Deserializer's power is dominated by registers (92.71%), primarily from internal consumption (91.05%). Leakage is low (1.12%), and switching contributes minimally (7.83%), with a total power of 2.28999e-04 W.

```
Instance: /Parallel_Clock_Deserializer
Power Unit: W
PDB Frames: /stim#0/frame#0
  ---------------------------------------------------------------------------
    Category         Leakage     Internal    Switching        Total     Row%
  ---------------------------------------------------------------------------
      memory     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
    register     2.47702e-06  2.07244e-04  2.59422e-06  2.12315e-04  92.71%
       latch     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
       logic     8.58236e-08  1.25651e-06  1.65230e-06  2.99464e-06   1.31%
        bbox     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
       clock     0.00000e+00  0.00000e+00  1.36890e-05  1.36890e-05   5.98%
         pad     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
          pm     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
  ---------------------------------------------------------------------------
    Subtotal     2.56284e-06  2.08500e-04  1.79355e-05  2.28999e-04 100.00%
  Percentage           1.12%       91.05%        7.83%      100.00% 100.00%
  ---------------------------------------------------------------------------
```

Fig. 5.38 power report

Area report:

Fig. 5. 39report for Parallel_Clock_Deserializer shows it uses 26 cells with a total area of 470.792. It includes Deserializer_N8 and NRZ_Decoder sub-modules, with default wireload settings.

```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 21.14-s082_1
  Generated on:           Mar 17 2025  06:19:56 am
  Module:                 Parallel_Clock_Deserializer
  Technology library:     slow
  Operating conditions:   slow (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
============================================================

          Instance               Module      Cell Count  Cell Area  Net Area   Total Area   Wireload
-----------------------------------------------------------------------------------------------------
Parallel_Clock_Deserializer                        26    470.792     0.000      470.792 <none> (D)
  deserializer_inst         Deserializer_N8        24    448.085     0.000      448.085 <none> (D)
  nrz_decoder_inst          NRZ_Decoder             2     22.707     0.000       22.707 <none> (D)
  (D) = wireload is default in technology library
```

Fig. 5.39 area report

Floor planning and power planning report:

Fig. 5.40 shows the power planning, placement and floor planning report of parallel clock deserializer

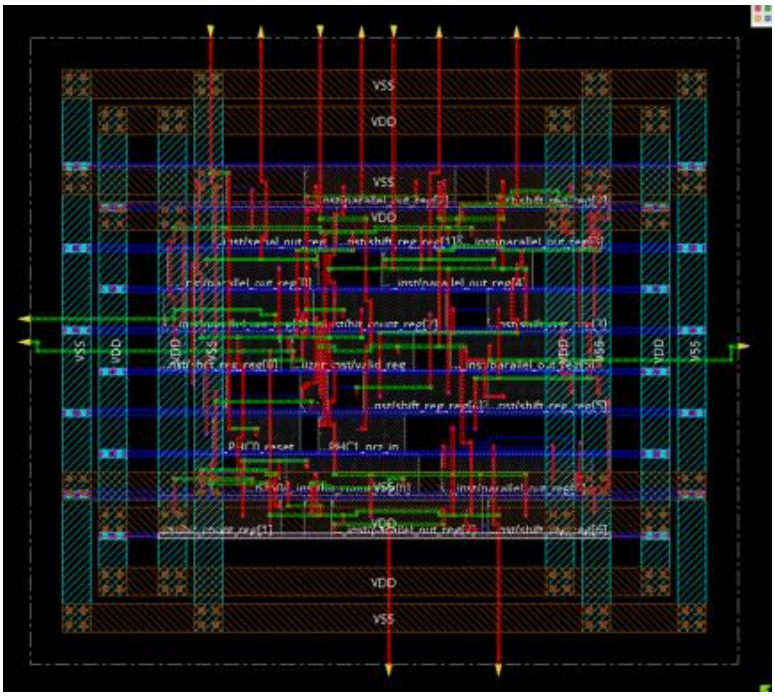


Fig. 5.40 power and floor planning report

Final SI report

Fig. 5.41shows Final SI Timing Summary shows successful setup and hold timing with positive WNS values and no violations. The design has a density of 74.747%.

```
------------------------------------------------------------------
     optDesign Final SI Timing Summary
------------------------------------------------------------------

Setup views included:
 setup
Hold views included:
 hold

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.364  |  0.364  |  1.175  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   56    |   35    |   21    |
+--------------------+---------+---------+---------+

+--------------------+---------+---------+---------+
|     Hold mode      |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.014  |  0.014  |  0.017  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   56    |   35    |   21    |
+--------------------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 74.747%
```

Fig. 5.41 final sign off report

Final optimized layout:

Fig. 5.42 shows the final layout which is optimized and added with filler

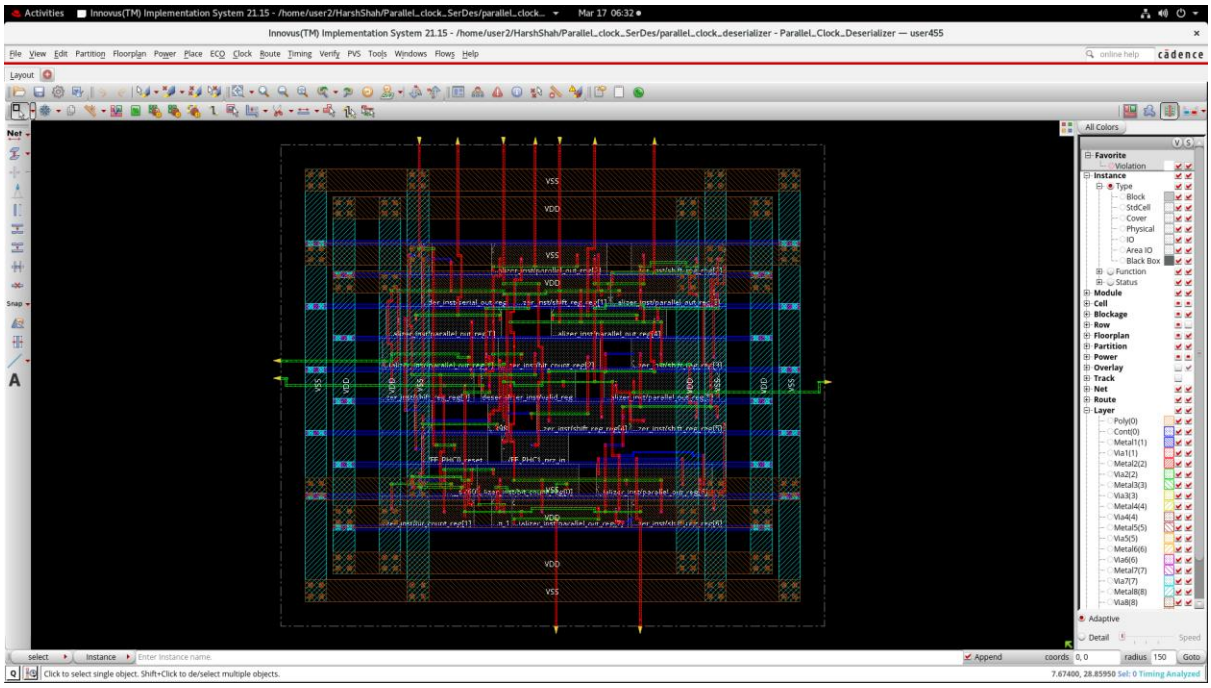


Fig. 5.42 final layout of parallel clock SerDes based receiver

CHAPTER 6

OBSERVED REPORT AND PERFORMANCE PARAMETERS

Embedded clock SerDes

Table 6.0-1 Embedded clock SerDes

Parameters	Industry	Measured
Data rate	100Mb/s-2.5Gb/s	100Mb/s
Jitter rate	< (10-5) %	6.68% (0.668 ns)
Updated jitter rate	<(10-5)%	5.6%

Physical parameters of Embedded clock SerDes based transmitter:

- **Timing:** Timing report from Genus synthesis tool shows a path delay of 1625ps, setup time of 8000ps, and timing slack of 6365ps.
- **Area:**Genus synthesis report for top_module_tx: 206 cells, total area 1566.026. Submodules: encoder_inst (157 cells, 1046.793), serializer_inst (49 cells, 519.233).
- **Power:** Power report for top_module_tx: Total power 132.146 µW. Registers dominate at 79.12%, logic contributes 14.87%, and clock 6.01%. Leakage is 6.13%.

Physical parameters of Embedded clock SerDes based receiver:

- **Timing:** Timing report for top_module_rx: Path delay 1816ps, setup time 2000ps, uncertainty 10ps, timing slack 11ps. Critical path: deserializer to decoder register.9
- **Area:** Genus synthesis report for top_module_rx: 235 cells, total area 1775.687. Largest submodules: decoder (1015.003), deserializer (717.541), CDR (43.143), input buffer (43.143).
- **Power:** Power report for top_module_rx: Total power 643.539 µW. Registers dominate 74.85%, logic 20.04%, clock 5.11%. Leakage 1.35%, internal 85.01%, switching 13.64%.

Performance Parameters of Parallel clock SerDes:

Table 6.2 parallel clock SerDes

Parameters	Industry	Measured
Data rate	100Mb/s-2.5Gb/s	100Mb/s
Jitter rate	<5%	3.5%

Physical parameters of parallel clock transmitter:

- **Timing:** Timing report for Parallel_Clock_Serializer: 8748ps timing slack. Start-point: busy_reg/CK, End-point: shift_reg_reg[0]/D. Setup time: 1242ps, Capture time: 9990ps. Critical path involves NOR2XL, AOI222XL, and INvXL. **area:** Synthesis report for Parallel_Clock_Serializer: 46 cells, 441.273 µm² total area. Major contributors: Serializer8bit (416.295 µm², 43 cells), NRZ_Encoder (22.707 µm², 2 cells).
- **Power:** Power report for Parallel_Clock_Serializer: Total power 41.48 µW. Registers dominate (83%), followed by logic (12.47%). Internal power 85.05%, leakage 4.75%, and switching 10.20%.

Physical parameters of parallel clock transmitter:

- **Timing:** Timing report for Parallel_Clock_Deserializer: 599ps timing slack. Start-point: bit_count_reg[0]/CK, End-point: parallel_out_reg[6]/SE. Critical path involves NAND2XL, NOR2X1, and SDFFRHQX1.
- **Area:** Power report for Parallel_Clock_Deserializer: Total power 2.2899e-4 W. Registers dominate (92.71%), followed by logic (1.31%) and clock (5.98%). Internal power is highest (91.05%).
- **Power:** Synthesis report for Parallel_Clock_Deserializer: 26 cells, total area 470.792. Deserializer_N8 dominates (448.085 area), with NRZ_Decoder (22.707 area). Wireload mode: enclosed.

Conclusion

This project have provided us an in depth understanding and analysis of clocked based SerDes system, firstly we understood that the SerDes systems are very important in this era of high speed communication explain that the SerDes system helps to convert data from serial to parallel and parallel to serial, this system helps in inter device communication and wired and wireless communication, for wired we can offer USB,I2C,SPI,PCIe and various other types of serial communication systems. With smaller modifications you can use parallel clock SerDes system in small distance communication systems like microcontroller to microcontroller, FPGA to FPGA, microcontroller to any small sensors.

For embedded clock SerDes system we have learned that we need to design a system which can regenerate clock frequency for receivers we have used an approach called clock divider system which divides higher clock frequency to the clock frequency which is used to transmit the data and uses that frequency to deserialize and further more decode the data with your more modifications we can use this embedded clock SerDes system in wireless and wired communication we have faced a clock jitter rate problem while designing the clock divider system to overcome that clock jitter rate we have created an NCO module which helps to regenerate the clock frequency and reduce noises which can be added also it is way more feasible in terms of frequency generation and it reduces the jitter rates.

After the design process we have completed the physical design part of this system, and we have achieved a proper and optimum physical design layout of these circuits using cadence EDA tools we have achieved area power and timing report and while performing the PNR flow we have ensured that the timing summaries should be always in positive numbers.

We also understand the importance of fillers in physical design it adds a protective layer, while we fabricate the chip micro dust particles can settle down on the chip to avoid this, we added the fillers in the design chip

For further advancement I have put a few floating pins in the embedded clock receiver layout so that I can add error detection and varies disparities in the transmutation in the system

References

Books: -

1. Fundamentals of Digital Logic withVerilog Design, THIRD EDITION, By Stephen Brown and Zvonko Vranesic
2. Introduction to Industrial Physical Design Flow.by Kunal Ghosh

Research papers: -

1. Mixed Signal Serializer/Deserializer (SerDes) Design using Sky130 and eSim by Ayush Gupta
2. SerDes Architectures and Applications by Dave Lewis, National Semiconductor Corporation
3. Design Methodology of Numerically Controlled Oscillator For Digital Waveform Generation By Shivangi Tandan 1, Dr. Shravan Kumar Sable
4. Design of CMOS Based Numerical Control Oscillator with Better Performance Parameter in 45nm CMOS Process, by Rudradatta Dhoke, Minal Kharbikar
5. Design and Implementation of Numerical Controlled Design and Implementation of Numerical Controlled Design and Implementation of Numerical Controlled Oscillator on FPGA by Nehal.A.Ranabhatt , Priyesh.P.Gandhi, Sudhir Agarwal, Raghunadh.K.Bhattar
6. Realization of FPGA based numerically Controlled Oscillator Gopal D. Ghiwala1, Pinakin P. Thaker2, Gireeja D.Amin
7. A Numerically controlled oscillator for all Digital Phase Locked Loop Anupama.Patil ;Dr P .H .Tandel*
8. A Review on Design and Implementation of Numerically Controlled Oscillator Arti D. Gaikwad, Govind U. Kharat, Shekhar H. Bodake
9. HIGH SPEED CLOCK AND DATA RECOVERY FOR SERDES APPLICATIONS Krishna G Namboothiri, Ashok Kumar, Sandip Paul, R. M. Parmar
10. Design of a New Serializer and Deserializer Architecture for On-Chip SerDes Transceivers Nivedita Jaiswal, Radheshyam Gamad
11. DESIGN AND OPTIMIZATION OF SERDES IN CMOS 45NM TECHNOLOGY 1Ravi Kumar M, 2Sanjai S, 3Shilpashree S, 4Shreyas N Kulal, 5Tinu M
12. DESIGN AND OPTIMIZATION OF SERDES IN CMOS 45NM TECHNOLOGY 1Ravi Kumar M, 2Sanjai S, 3Shilpashree S, 4Shreyas N Kulal, 5Tinu M

13. Mixed Signal Serializer/Deserializer (SerDes) Design using Sky130 and eSim Ayush Gupta
14. Article Design and Implementation of Fast Locking All-Digital Duty Cycle Corrector Circuit with Wide Range Input Frequency Shao-Ku Ka
15. An All-Digital Fast-Locking Programmable DLL-Based Clock Generator Chuan-Kang Liang,StudentMember,IEEE, Rong-Jyi Yang,Member,IEEE,and Shen-Iuan Liu,SeniorMember,IEEE
16. A Novel MUX-FF Circuit for Low Power and High Speed Serial Link Interfaces Wei-Yu Tsai, Ching-Te Chiu, Jen-Ming Wu, Shuo-Hung Hsu, Yar-Sun Hsu
17. AN ALL-DIGITAL PHASE-LOCKED LOOP FOR HIGH-SPEED CLOCK GENERATION Ching-Che Chung and Chen-Yi Lee
18. A Digitally Controlled PLL for Digital SOCs Thomas Olsson and Peter Nilsson

Website links: -

To understand the types if SerDes**:-**https://resources.pcb.cadence.com/blog/2020-serdes-design-high-speed-electronic-challenges

To theoretically understand the SerDes circuit: -https://www.synopsys.com/glossary/what-is-serdes.html

To understand the different parameters of SerDes: -
https://www.ti.com/lit/ds/slls574d/slls574d.pdf

To understand the different SerDes circuit:https://www.ti.com/lit/ug/spruho3a/spruho3a.pdf?ts=1743080024306&ref_url=https%253A%252F%252Fwww.google.com%252F

Appendix A ABBREVATIONS

USB	Universal Serial Bus
PCIe	Peripheral Component Interconnect Express
I2C	Inter-Integrated Circuit
SPI	Serial Peripheral Interface
CDR	Clock Data recovery
NCO	numerically controlled oscillator
DAC	digital to analog convertor
HDMI	High-Definition Multimedia Interface
SATA	Serial Advanced Technology Attachment.
FPGA	field programable gate array
ASIC	application specific integrated circuit

Appendix B CODES

Design Verilog code

top_module_tx.v

```
module top_module_tx (
   input wire clk,
   input wire rst,
   input wire en,
   input wire kin,
   input wire [7:0] din,
   output wire serial_out
);
   wire [9:0] encoder_dout;
   wire disp;
   wire kin_err;
   encoder_8b10 encoder_inst (
      .clk(clk),
      .rst(rst),
      .en(en),
      .kin(kin),
      .din(din),
      .dout(encoder_dout),
      .disp(disp),
      .kin_err(kin_err)
   );
   serializer serializer_inst (
   .clk(clk),
   .reset(rst),
   .load(en),              // Map "en" to "load"
   .parallel_data(encoder_dout), // Map "encoder_dout" to "parallel_data"
   .serial_out(serial_out)
);
endmodule
module encoder_8b10
(
         input wire clk,
         input wire rst,
         input wire en,
         input wire kin,
         input wire [7:0]din,
         output wire [9:0]dout,
         output wire disp,
         output wire kin_err
);
reg p;
reg ke;
reg [18:0]t;
reg [9:0]do;
wire [7:0]d;
wire k;
assign d = din;
assign k = kin;
assign dout = do;
```

```
assign disp = p;
assign kin_err = ke;
always @(posedge clk) begin
	if (rst) begin
		p <= 1'b0;
		ke <= 1'b0;
		do <= 10'b0;
	end else begin
		if (en == 1'b1) begin
			p <=
((d[5]&d[6]&d[7])|(!d[5]&!d[6]))^(p^(((d[4]&d[3]&!d[2]&!d[1]&!d[0])|(!d[4]&!((d[0]&d[1]&!d[2]
&!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!d[2]&!d[3]))))&!((!(
(d[0]&d[1])|(!d[0]&!d[1]))&d[2]&d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))&d[0]&d[1]))))|(k|(d[4]&!((d[0]
&d[1]&!d[2]&!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!d[2]&!
d[3]))))&!((!((d[0]&d[1])|(!d[0]&!d[1]))&!d[2]&!d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))&!d[0]&!d[1]))))
));
			ke <=
(k&(d[0]|d[1]|!d[2]|!d[3]|!d[4])&(!d[5]|!d[6]|!d[7]|!d[4]|!((!((d[0]&d[1])|(!d[0]&!d[1]))&d[2]&d[3])|(!
((d[2]&d[3])|(!d[2]&!d[3]))&d[0]&d[1]))));
			do[9] <= t[12]^t[0];
			do[8] <= t[12]^(t[1]|t[2]);
			do[7] <= t[12]^(t[3]|t[4]);
			do[6] <= t[12]^t[5];
			do[5] <= t[12]^(t[6]&t[7]);
			do[4] <= t[12]^(t[8]|t[9]|t[10]|t[11]);
			do[3] <= t[13]^(t[15]&!t[14]);
			do[2] <= t[13]^t[16];
			do[1] <= t[13]^t[17];
			do[0] <= t[13]^(t[18]|t[14]);
		end
	end
end
always @(posedge clk) begin
	if(rst) begin
		t <= 0;
	end else begin
		if (en == 1'b1) begin
			t[0] <= d[0];
			t[1] <= d[1]&!(d[0]&d[1]&d[2]&d[3]);
			t[2] <= (!d[0]&!d[1]&!d[2]&!d[3]);
			t[3] <= (!d[0]&!d[1]&!d[2]&!d[3])|d[2];
			t[4] <= d[4]&d[3]&!d[2]&!d[1]&!d[0];
			t[5] <= d[3]&!(d[0]&d[1]&d[2]);
			t[6] <=
d[4]|((!((d[0]&d[1])|(!d[0]&!d[1]))&!d[2]&!d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))&!d[0]&!d[1]));
			t[7] <= !(d[4]&d[3]&!d[2]&!d[1]&!d[0]);
			t[8] <=
(((d[0]&d[1]&!d[2]&!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!
d[2]&!d[3]))))&!d[4])|(d[4]&(d[0]&d[1]&d[2]&d[3]));
			t[9] <= d[4]&!d[3]&!d[2]&!(d[0]&d[1]);
			t[10] <= k&d[4]&d[3]&d[2]&!d[1]&!d[0];
			t[11] <= d[4]&!d[3]&d[2]&!d[1]&!d[0];
			t[12] <=
(((d[4]&d[3]&!d[2]&!d[1]&!d[0])|(!d[4]&!((d[0]&d[1]&!d[2]&!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[
0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!d[2]&!d[3]))))&!((!((d[0]&d[1])|(!d[0]&!d[1]))&d[2]&d[3])
```

```
|(!((d[2]&d[3])|(!d[2]&!d[3]))&d[0]&d[1]))))&!p)|((k|(d[4]&!((d[0]&d[1]&!d[2]&!d[3])|(d[2]&d[3]&
!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!d[2]&!d[3]))))&!((!((d[0]&d[1])|(!d[0]&!
d[1]))&!d[2]&!d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))&!d[0]&!d[1])))|(!d[4]&!d[3]&d[2]&d[1]&d[0]))&
p);
                        t[13] <=
(((!d[5]&!d[6])|(k&((d[5]&!d[6])|(!d[5]&d[6]))))&!(p^(((d[4]&d[3]&!d[2]&!d[1]&!d[0])|(!d[4]&!((d[
0]&d[1]&!d[2]&!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!d[2]
&!d[3]))))&!((!((d[0]&d[1])|(!d[0]&!d[1]))&d[2]&d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))&d[0]&d[1])))|
(k|(d[4]&!((d[0]&d[1]&!d[2]&!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]
&d[3])|(!d[2]&!d[3]))))&!((!((d[0]&d[1])|(!d[0]&!d[1]))&!d[2]&!d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))
&!d[0]&!d[1])))))))|((d[5]&d[6])&(p^(((d[4]&d[3]&!d[2]&!d[1]&!d[0])|(!d[4]&!((d[0]&d[1]&!d[2]&
!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!d[2]&!d[3]))))&!((!((d
[0]&d[1])|(!d[0]&!d[1]))&d[2]&d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))&d[0]&d[1])))|(k|(d[4]&!((d[0]&
d[1]&!d[2]&!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!d[2]&!d[
3]))))&!((!((d[0]&d[1])|(!d[0]&!d[1]))&!d[2]&!d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))&!d[0]&!d[1])))))
);
                        t[14] <=
d[5]&d[6]&d[7]&(k|(p?(!d[4]&d[3]&((!((d[0]&d[1])|(!d[0]&!d[1]))&d[2]&d[3])|(!((d[2]&d[3])|(!d[2]
&!d[3]))&d[0]&d[1])):(d[4]&!d[3]&((!((d[0]&d[1])|(!d[0]&!d[1]))&!d[2]&!d[3])|(!((d[2]&d[3])|(!d[
2]&!d[3]))&!d[0]&!d[1])))));
                        t[15] <= d[5];
                        t[16] <= d[6]|(!d[5]&!d[6]&!d[7]);
                        t[17] <= d[7];
                        t[18] <= !d[7]&(d[6]^d[5]);
                    end
            end
end

endmodule
module serializer(
   input wire clk,              // Clock signal
   input wire reset,            // Asynchronous reset signal
   input wire load,             // Load signal to load parallel data
   input wire [9:0] parallel_data,// Parallel input data (10-bit)
   output reg serial_out        // Serial output
);
   reg [9:0] shift_reg;         // Shift register to hold data
   reg [3:0] bit_counter;       // Counter to track bit position (0 to 9)
   reg busy;                    // Flag to indicate active serialization
   always @(posedge clk or posedge reset) begin
      if (reset) begin
         // Reset all registers
         shift_reg <= 10'b0;
         serial_out <= 1'b0;
         bit_counter <= 4'd0;
         busy <= 1'b0;          // Not busy after reset
      end else if (load) begin
         // Load parallel data into shift register
         shift_reg <= parallel_data;
         bit_counter <= 4'd9;   // Initialize bit counter to MSB
         busy <= 1'b1;          // Start serialization
      end else if (busy) begin
         // Perform serialization
         serial_out <= shift_reg[bit_counter]; // Output current bit
         if (bit_counter == 4'd0) begin
```

```
            busy <= 1'b0;      // Serialization complete
        end else begin
            bit_counter <= bit_counter - 1; // Decrement bit counter
        end
    end
  end
endmodule
```

top_module_rx.v

```
module top_module_rx (
  input wire rst,
  input wire clk,
  input wire a_rx_serial,
  input wire disparity_d,
  output wire [7:0] dout,
  output wire code_err,
  output wire kout,
  output wire disp,
  output wire disp_err
);
  // Instantiate inputBuffer module
  wire [9:0] dout_buffer_input;
  inputBuffer input_buffer_inst (
    .rst(rst),
    .clk(clk),
    .a_rx_serial(a_rx_serial),
    .dout(dout_buffer_input)
  );
  // Deserializer buffer
  wire [9:0] dout_buffer_deserializer;
  // Instantiate deserializer module (modify to use dout_buffer_deserializer)
  deserializer deserializer_inst (
    .rst(rst),
    .clk(clk),
    .a_rx(dout_buffer_input[0]),
    .disparity_d(disparity_d),
    .c_parallel_out(dout_buffer_deserializer), // Use dout_buffer_deserializer
    .clk_out(clk_out),
    .disparity_q(disparity_q),
    .c_data_valid(c_data_valid)
  );
  // Decoder buffer
  wire [9:0] dout_buffer_decoder;
  // Modify decoder instantiation to use dout_buffer_decoder
  decodePipe decoder_inst (
    .clk(clk),
    .rst(rst),
    .en(c_data_valid),
    .din(dout_buffer_deserializer), // Use dout_buffer_deserializer
    .dout(dout), // Connect decoder output to module output
    .kout(kout),
    .code_err(code_err),
    .disp(disp),
    .disp_err(disp_err)
```

```
    );
endmodule
// ... (rest of the modules: deserializer, CDR, inputBuffer, decodePipe)
// remain unchanged ...
module deserializer (
    input wire rst,
    input wire clk,           // Single clock input
    input wire a_rx,          // Serial data input
    input wire disparity_d,   // Running disparity input
    output reg [9:0] c_parallel_out, // 10-bit parallel data output
    output reg clk_out,       // Recovered clock output
    output reg disparity_q,   // Running disparity output
    output reg c_data_valid   // Data valid flag
);
    reg [9:0] shift_reg;      // Shift register for serial-to-parallel conversion
    reg [3:0] cycle_count;    // Cycle counter
    wire sampled_data;        // Sampled data from CDR
    reg comma_detected;       // Comma detection flag
    // CDR instance for clock and data recovery
    wire bit_clock;
    CDR cdr_inst (
        .rst(rst),
        .clks_in({clk, clk, clk, clk}), // Replace with 4-phase clock signals
        .a_rx(a_rx),
        .bit_clock(bit_clock),
        .samp_test(sampled_data)
    );
    // Serial-to-parallel shift register logic
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            shift_reg <= 10'b0;
        end else begin
            shift_reg <= {shift_reg[8:0], sampled_data};
        end
    end
    // Running disparity logic
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            disparity_q <= 1'b0;
        end else begin
            disparity_q <= disparity_d;
        end
    end
    // Comma detection logic (for 10-bit comma patterns)
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            comma_detected <= 1'b0;
        end else begin
            comma_detected <= (shift_reg == 10'b0011111100) || (shift_reg == 10'b1100000011);
        end
    end
    // Cycle counter and clk_out generation
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            cycle_count <= 4'd9;
```

```
            c_data_valid <= 1'b0;
            clk_out <= 1'b0;
        end else begin
            if (comma_detected) begin
                cycle_count <= 4'd9;
                c_data_valid <= 1'b0;
            end else if (cycle_count == 4'd0) begin
                cycle_count <= 4'd9;
                c_data_valid <= 1'b1;
            end else begin
                cycle_count <= cycle_count - 1;
                c_data_valid <= 1'b0;
            end
            clk_out <= (cycle_count == 4'd5);
        end
    end
    // Parallel data output logic
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            c_parallel_out <= 10'b0;
        end else if (c_data_valid) begin
            c_parallel_out <= shift_reg;
        end
    end
endmodule

// Updated CDR Module
module CDR (
    input wire rst,
    input wire [3:0] clks_in, // Multi-phase clock input
    input wire a_rx,          // Serial data input
    output reg bit_clock,     // Recovered clock output
    output reg samp_test      // Sampled data output
);
    reg [3:0] c_rx_upsampled;
    reg [1:0] best_samp;
    // Upsampling logic using 4-phase clock
    generate
        genvar i;
        for (i = 0; i < 4; i = i + 1) begin : upsample
            always @(posedge clks_in[i] or posedge rst) begin
                if (rst) begin
                    c_rx_upsampled[i] <= 1'b0;
                end else begin
                    c_rx_upsampled[i] <= a_rx;
                end
            end
        end
    endgenerate
    // Transition detection and phase selection logic
    always @(posedge clks_in[0] or posedge rst) begin
        if (rst) begin
            best_samp <= 2'b00;
        end else begin
            // Simple phase selection (placeholder logic)
```

```
            best_samp <= 2'b01;
        end
    end

    // Recovered clock and sampled data output
    always @(posedge clks_in[best_samp] or posedge rst) begin
        if (rst) begin
            bit_clock <= 1'b0;
            samp_test <= 1'b0;
        end else begin
            bit_clock <= clks_in[best_samp];
            samp_test <= c_rx_upsampled[best_samp];
        end
    end
endmodule
module inputBuffer (
    input wire rst,           // Reset signal
    input wire clk,           // Clock signal
    input wire a_rx_serial,   // Serial data input
    output reg [9:0] dout     // Parallel data output
);
    // Shift register to hold serial data
    reg [9:0] shift_reg;
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            // Reset the shift register and output
            shift_reg <= 10'b0;
            dout <= 10'b0;
        end else begin
            // Shift in new serial data and update the output
            shift_reg <= {shift_reg[8:0], a_rx_serial};
            dout <= shift_reg;
        end
    end

endmodule
module decodePipe (
            input wire clk,
            input wire rst,
            input wire en,
            input wire [9:0]din,
            output wire [7:0]dout,
            output wire kout,
            output wire code_err,
            output wire disp,
            output wire disp_err
);
reg [7:0]do;
reg k;
reg ce;
reg [2:0]e;
reg p;
reg [3:0]pe;
wire [9:0]d;
assign d = din;
```

```
assign disp_err = pe?1'b1:1'b0;
assign dout = do;
assign kout = k;
assign code_err = ce;
assign disp = p;

always @(posedge clk) begin
	if (rst) begin
		k <= 0;
		do <= 8'b0;
	end else begin
		if (en == 1'b1) begin
			k <=
(((d[7]&d[6]&d[5]&d[4])|(!d[7]&!d[6]&!d[5]&!d[4]))|((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(
!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&(!d[5]&d[4]&d[2]&d[1]&d[0]))|((((!((d[9]&d[8])|(!d[9]
&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&(d[5]&!d[4]&!d[2]&!d[1]&!d[0]))
);
			do[7] <=
((d[0]^d[1])&!((!d[3]&d[2]&!d[1]&d[0]&!(!(d[7]|d[6]|d[5]|d[4])))|(!d[3]&d[2]&d[1]&!d[0]&(!(d[7]|d
[6]|d[5]|d[4])))|(d[3]&!d[2]&!d[1]&d[0]&!(!(d[7]|d[6]|d[5]|d[4])))|(d[3]&!d[2]&d[1]&!d[0]&(!(d[7]|d
[6]|d[5]|d[4]))))|(!d[3]&d[2]&d[1]&d[0])|(d[3]&!d[2]&!d[1]&!d[0]);
			do[6] <=
(d[0]&!d[3]&(d[1]|!d[2]|!(!(d[7]|d[6]|d[5]|d[4]))))|(d[3]&!d[0]&(!d[1]|d[2]|(!(d[7]|d[6]|d[5]|d[4]))))|(!(
!(d[7]|d[6]|d[5]|d[4]))&d[2]&d[1])|((!(d[7]|d[6]|d[5]|d[4]))&!d[2]&!d[1]);
			do[5] <=
(d[0]&!d[3]&(d[1]|!d[2]|(!(d[7]|d[6]|d[5]|d[4]))))|(d[3]&!d[0]&(!d[1]|d[2]|!(!(d[7]|d[6]|d[5]|d[4]))))|((!
(d[7]|d[6]|d[5]|d[4]))&d[2]&d[1])|(!(!(d[7]|d[6]|d[5]|d[4]))&!d[2]&!d[1]);
			do[4] <=
d[5]^(((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d
[5])|(!d[7]&!d[6]&!d[5]&!d[4])|(!d[9]&!d[8]&!d[5]&!d[4]))|((((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]
&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[9]&!d[7]&(!(d[5]^d[
4])))|(((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6])) & !d[9] &
!d[8]))&!d[4]))|((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!
d[8]))&d[6]&d[5]&d[4])|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&
!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[8]&!d[7]&(!(d[5]^d[4]))));
			do[3] <=
d[6]^(((d[9]&d[8]&d[5]&d[4])|(!d[7]&!d[6]&!d[5]&!d[4])|(((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6
])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&d[4]))|((((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&
!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&d[9]&d[7]&(!(d[5]^d[4])))|(((!((
d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d[5]))|((((!((d
[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&d[6]&d[5]&d[4
])|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|
(!d[7]&!d[6]))))&!d[8]&!d[7]&(!(d[5]^d[4]))));
			do[2] <=
d[7]^(((((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d
[6])|(!d[7]&!d[6]))))&!d[9]&!d[7]&(!(d[5]^d[4])))|(((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d
[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d[5]))|((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]
&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&d[4])|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d
[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&d[8]&d[7]&(!(d[5]^d[4])))|(((!((d[9]&d[8])
|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&d[6]&d[5]&d[4]))|(((((!((
d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d[5])|(!d[7]&
!d[6]&!d[5]&!d[4])|(!d[9]&!d[8]&!d[5]&!d[4])));
			do[1] <=
d[8]^(((d[9]&d[8]&d[5]&d[4])|(!d[7]&!d[6]&!d[5]&!d[4])|(((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6
])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&d[4]))|((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((
```

```
d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&d[4])|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(
!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&d[8]&d[7]&(!(d[5]^d[4])))|(((!((d[9]&d
[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&d[6]&d[5]&d[4]))|(((
(d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[
7]&!d[6]))))&d[9]&d[7]&(!(d[5]^d[4])))|(((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])
|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d[5])));
                    do[0] <=
d[9]^(((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&d[
6]&d[5]&d[4])|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!
((d[7]&d[6])|(!d[7]&!d[6]))))&!d[8]&!d[7]&(!(d[5]^d[4]))))|((((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]
&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[9]&!d[7]&(!(d[5]^d[
4])))|(((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d[5
]))|((d[9]&d[8]&d[5]&d[4])|(!d[7]&!d[6]&!d[5]&!d[4])|(((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(
!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&d[4])));
            end
        end
end
always @(posedge clk) begin
        if (rst) begin
            p <= 1'b0;
            pe <= 4'hF;
            ce <= 1'b1;
            e <= 3'b000;
        end else begin
            if (en == 1'b1) begin
                    p <=
(((!((d[3]&d[2])|(!d[3]&!d[2])))&d[1]&d[0])|(!((d[1]&d[0])|(!d[1]&!d[0]))&d[3]&d[2]))|(((d[5]&d[4]
&!(((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!p))|((
((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))|(((d[9]&d[8]&
!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))
&p))&(d[5]|d[4]))|(((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d
[8]))&p))&((d[3]&d[2]&!d[1]&!d[0])|(!d[3]&!d[2]&d[1]&d[0])|(!((d[3]&d[2])|(!d[3]&!d[2]))&!((d[1
]&d[0])|(!d[1]&!d[0])))) ;
                    pe[0] <=
((p&((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&(d[5]|d
[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[
6])|(!d[7]&!d[6]))))&d[5]&d[4])))|(((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[
7]&!d[6]))&!d[9]&!d[8]))&!(d[5]&d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[
9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[5]&!d[4]))&!p))|((p&!((((!((d[9]&d[8])|(
!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!(d[5]&d[4]))|(((d[9]&d[8
]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))
))&!d[5]&!d[4]))&d[3]&d[2]));
                    pe[1] <=
((p&d[9]&d[8]&d[7]))|((p&!((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[
6]))&!d[9]&!d[8]))&!(d[5]&d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8
])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[5]&!d[4]))&(((!((d[3]&d[2])|(!d[3]&!d[2])))&d[
1]&d[0])|(!((d[1]&d[0])|(!d[1]&!d[0]))&d[3]&d[2]))));
                    pe[2] <=
((!p&!((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&(d[5]
|d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d
[6])|(!d[7]&!d[6]))))&d[5]&d[4]))&!d[3]&!d[2]))|((!p&!d[9]&!d[8]&!d[7]));
                    pe[3] <=
((!p&!((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&(d[5]
|d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d
[6])|(!d[7]&!d[6]))))&d[5]&d[4]))&((!((d[3]&d[2])|(!d[3]&!d[2]))&!d[1]&!d[0])|(!((d[1]&d[0])|(!d[1]
```

```
&!d[0]))&!d[3]&!d[2]))))|((((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6])
)&d[9]&d[8]))&(d[5]|d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[
9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&d[5]&d[4]))&(((!((d[3]&d[2])|(!d[3]&!d[2])))&d[1]&d[0
])|(!((d[1]&d[0])|(!d[1]&!d[0]))&d[3]&d[2])))|(((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]
&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!(d[5]&d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!
d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[5]&!d[4]))&(((!((d[3]&d[2])|(!
d[3]&!d[2]))&!d[1]&!d[0])|(!((d[1]&d[0])|(!d[1]&!d[0]))&!d[3]&!d[2]))));
                    e[0] <=
((d[9]&d[8]&d[7]&d[6])|(!d[9]&!d[8]&!d[7]&!d[6]))|((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(
!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d[5]&!d[4]))|((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[
6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&d[5]&d[4]))|((d[3]&d[2]&d[1]&d[0])|(!d[3]&!d[2]&!
d[1]&!d[0]))|((d[5]&d[4]&d[3]&d[2]&d[1])|(!d[5]&!d[4]&!d[3]&!d[2]&!d[1]))|((d[5]&!d[4]&d[2]&
d[1]&d[0])|(!d[5]&d[4]&!d[2]&!d[1]&!d[0]))|((((d[5]&d[4]&!d[2]&!d[1]&!d[0])|(!d[5]&!d[4]&d[2]
&d[1]&d[0]))&!((d[7]&d[6]&d[5])|(!d[7]&!d[6]&!d[5]))))|((!((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[
6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&d[5]&!d[4]&!d[2]&!d[1]&!d[0]))|((!((!((d[9]&d[8])|(
!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d[5]&d[4]&d[2]&d[1]&d
[0]));
                    e[1] <=
((((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&(d[5]|d[4]
))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|
(!d[7]&!d[6]))))&d[5]&d[4]))&(((!((d[3]&d[2])|(!d[3]&!d[2])))&d[1]&d[0])|(!((d[1]&d[0])|(!d[1]&!d
[0]))&d[3]&d[2])))|(((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[
9]&!d[8]))&!(d[5]&d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]
&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[5]&!d[4]))&(((!((d[3]&d[2])|(!d[3]&!d[2]))&!d[1]&!d[0]
)|(!((d[1]&d[0])|(!d[1]&!d[0]))&!d[3]&!d[2]))))|((d[3]&d[2]&!d[1]&!d[0]&((((!((d[9]&d[8])|(!d[9]&!
d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&(d[5]|d[4]))|(((d[9]&d[8]&!d[7]&!d[6
])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&d[5]&d[4]
))))|((!d[3]&!d[2]&d[1]&d[0]&((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!
d[6]))&!d[9]&!d[8]))&!(d[5]&d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d
[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[5]&!d[4]))));
                    e[2] <=
((d[9]&d[8]&d[7]&!d[5]&!d[4]&((!d[3]&!d[2])|((!((d[3]&d[2])|(!d[3]&!d[2]))&!d[1]&!d[0])|(!((d[1]
&d[0])|(!d[1]&!d[0]))&!d[3]&!d[2])))))|((!d[9]&!d[8]&!d[7]&d[5]&d[4]&((d[3]&d[2])|((((!((d[3]&d[
2])|(!d[3]&!d[2])))&d[1]&d[0])|(!((d[1]&d[0])|(!d[1]&!d[0]))&d[3]&d[2])))))|((d[7]&d[6]&d[5]&d[4
]&!d[3]&!d[2]&!d[1]))|((!d[7]&!d[6]&!d[5]&!d[4]&d[3]&d[2]&d[1]));
                    ce <= e ? 1'b1 : 1'b0;
                end
        end
end
endmodule
```

decodePipe.v

```
module decodePipe (
        input wire clk,
        input wire rst,
        input wire en,
        input wire [9:0]din,
        output wire [7:0]dout,
        output wire kout,
        output wire code_err,
        output wire disp,
        output wire disp_err
);
```

```
reg [7:0]do;
reg k;
reg ce;
reg [2:0]e;
reg p;
reg [3:0]pe;
wire [9:0]d;

assign d = din;
assign disp_err = pe?1'b1:1'b0;
assign dout = do;
assign kout = k;
assign code_err = ce;
assign disp = p;

always @(posedge clk) begin
	if (rst) begin
		k <= 0;
		do <= 8'b0;
	end else begin
		if (en == 1'b1) begin
			k <=
(((d[7]&d[6]&d[5]&d[4])|(!d[7]&!d[6]&!d[5]&!d[4]))|((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(
!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&(!d[5]&d[4]&d[2]&d[1]&d[0]))|((((!((d[9]&d[8])|(!d[9]
&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&(d[5]&!d[4]&!d[2]&!d[1]&!d[0]))
);
			do[7] <=
((d[0]^d[1])&!((!d[3]&d[2]&!d[1]&d[0]&!(!(d[7]|d[6]|d[5]|d[4])))|(!d[3]&d[2]&d[1]&!d[0]&(!(d[7]|d
[6]|d[5]|d[4])))|(d[3]&!d[2]&!d[1]&d[0]&!(!(d[7]|d[6]|d[5]|d[4])))|(d[3]&!d[2]&d[1]&!d[0]&(!(d[7]|d
[6]|d[5]|d[4])))))|(!d[3]&d[2]&d[1]&d[0])|(d[3]&!d[2]&!d[1]&!d[0]);
			do[6] <=
(d[0]&!d[3]&(d[1]|!d[2]|!(!(d[7]|d[6]|d[5]|d[4]))))|(d[3]&!d[0]&(!d[1]|d[2]|(!(d[7]|d[6]|d[5]|d[4]))))|(!(
!(d[7]|d[6]|d[5]|d[4]))&d[2]&d[1])|((!(d[7]|d[6]|d[5]|d[4]))&!d[2]&!d[1]);
			do[5] <=
(d[0]&!d[3]&(d[1]|!d[2]|(!(d[7]|d[6]|d[5]|d[4]))))|(d[3]&!d[0]&(!d[1]|d[2]|!(!(d[7]|d[6]|d[5]|d[4]))))|((!
(d[7]|d[6]|d[5]|d[4]))&d[2]&d[1])|(!(!(d[7]|d[6]|d[5]|d[4]))&!d[2]&!d[1]);
			do[4] <=
d[5]^(((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d
[5])|(!d[7]&!d[6]&!d[5]&!d[4])|(!d[9]&!d[8]&!d[5]&!d[4]))|((((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]
&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[9]&!d[7]&(!(d[5]^d[
4])))|(((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6])) & !d[9] &
!d[8]))&!d[4]))|((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!
d[8]))&d[6]&d[5]&d[4])|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&
!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[8]&!d[7]&(!(d[5]^d[4]))));
			do[3] <=
d[6]^(((d[9]&d[8]&d[5]&d[4])|(!d[7]&!d[6]&!d[5]&!d[4])|((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6
])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&d[4]))|((((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&
!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&d[9]&d[7]&(!(d[5]^d[4])))|((((!((
d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d[5]))|((((!((d
[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&d[6]&d[5]&d[4
])|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|
(!d[7]&!d[6]))))&!d[8]&!d[7]&(!(d[5]^d[4]))));
			do[2] <=
d[7]^(((((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d
```

```
[6])|(!d[7]&!d[6]))))&!d[9]&!d[7]&(!(d[5]^d[4])))|(((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d
[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d[5]))|((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]
&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&d[4])|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d
[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&d[8]&d[7]&(!(d[5]^d[4])))|(((!((d[9]&d[8])
|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&d[6]&d[5]&d[4]))|((((!((
d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d[5])|(!d[7]&
!d[6]&!d[5]&!d[4])|(!d[9]&!d[8]&!d[5]&!d[4]));
                    do[1] <=
d[8]^(((d[9]&d[8]&d[5]&d[4])|(!d[7]&!d[6]&!d[5]&!d[4])|((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6
])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&d[4]))|((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((
d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&d[4])|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(
!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&d[8]&d[7]&(!(d[5]^d[4])))|(((!((d[9]&d
[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&d[6]&d[5]&d[4]))|(((
(d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[
7]&!d[6]))))&d[9]&d[7]&(!(d[5]^d[4])))|(((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])
|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d[5])));
                    do[0] <=
d[9]^(((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&d[
6]&d[5]&d[4])|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!
((d[7]&d[6])|(!d[7]&!d[6]))))&!d[8]&!d[7]&(!(d[5]^d[4]))))|((((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]
&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[9]&!d[7]&(!(d[5]^d[
4])))|(((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d[5
]))|((d[9]&d[8]&d[5]&d[4])|(!d[7]&!d[6]&!d[5]&!d[4])|((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(
!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&d[4])));
                end
        end
end

always @(posedge clk) begin
        if (rst) begin
                p <= 1'b0;
                pe <= 4'hF;
                ce <= 1'b1;
                e = 3'b000;
        end else begin
                if (en == 1'b1) begin
                        p <=
(((!((d[3]&d[2])|(!d[3]&!d[2])))&d[1]&d[0])|(!((d[1]&d[0])|(!d[1]&!d[0]))&d[3]&d[2]))|(((d[5]&d[4]
&!(((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!p))|((
(((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))|(((d[9]&d[8]&
!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))
&p))&(d[5]|d[4]))|(((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d
[8]))&p))&((d[3]&d[2]&!d[1]&!d[0])|(!d[3]&!d[2]&d[1]&d[0])|(!((d[3]&d[2])|(!d[3]&!d[2]))&!((d[1
]&d[0])|(!d[1]&!d[0])))) ;
                        pe[0] <=
((p&((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&(d[5]|d
[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[
6])|(!d[7]&!d[6]))))&d[5]&d[4])))|(((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[
7]&!d[6]))&!d[9]&!d[8]))&!(d[5]&d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[
9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[5]&!d[4]))&!p))|((p&!(((((!((d[9]&d[8])|(
!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!(d[5]&d[4]))|(((d[9]&d[8
]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))
))&!d[5]&!d[4]))&d[3]&d[2]));
                        pe[1] <=
((p&d[9]&d[8]&d[7]))|((p&!(((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[
```

```
6]))&!d[9]&!d[8]))&!(d[5]&d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8
])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[5]&!d[4]))&(((!((d[3]&d[2])|(!d[3]&!d[2])))&d[
1]&d[0])|(!((d[1]&d[0])|(!d[1]&!d[0]))&d[3]&d[2]))));
                        pe[2] <=
((!p&!((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&(d[5]
|d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d
[6])|(!d[7]&!d[6]))))&d[5]&d[4]))&!d[3]&!d[2]))|((!p&!d[9]&!d[8]&!d[7]));
                        pe[3] <=
((!p&!((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&(d[5]
|d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d
[6])|(!d[7]&!d[6]))))&d[5]&d[4]))&((!((d[3]&d[2])|(!d[3]&!d[2]))&!d[1]&!d[0])|(!((d[1]&d[0])|(!d[1]
&!d[0]))&!d[3]&!d[2]))))|((((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6])
)&d[9]&d[8]))&(d[5]|d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[
9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&d[5]&d[4]))&(((!((d[3]&d[2])|(!d[3]&!d[2]))&d[1]&d[0
])|(!((d[1]&d[0])|(!d[1]&!d[0]))&d[3]&d[2]))))|(((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]
&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!(d[5]&d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!
d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[5]&!d[4]))&((!((d[3]&d[2])|(!
d[3]&!d[2]))&!d[1]&!d[0])|(!((d[1]&d[0])|(!d[1]&!d[0]))&!d[3]&!d[2]))));
                        e[0] <=
((d[9]&d[8]&d[7]&d[6])|(!d[9]&!d[8]&!d[7]&!d[6]))|((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(
!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d[5]&!d[4]))|((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[
6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&d[5]&d[4]))|((d[3]&d[2]&d[1]&d[0])|(!d[3]&!d[2]&!
d[1]&!d[0]))|((d[5]&d[4]&d[3]&d[2]&d[1])|(!d[5]&!d[4]&!d[3]&!d[2]&!d[1]))|((d[5]&!d[4]&d[2]&
d[1]&d[0])|(!d[5]&d[4]&!d[2]&!d[1]&!d[0]))|((((d[5]&d[4]&!d[2]&!d[1]&!d[0])|(!d[5]&!d[4]&d[2]
&d[1]&d[0]))&!((d[7]&d[6]&d[5])|(!d[7]&!d[6]&!d[5]))))|((!((((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[
6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&d[5]&!d[4]&!d[2]&!d[1]&!d[0]))|((!((!((d[9]&d[8])|(
!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[9]&!d[8]))&!d[5]&d[4]&d[2]&d[1]&d
[0]));
                        e[1] <=
((((((!((d[9]&d[8])|(!d[9]&!d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&(d[5]|d[4]
))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|
(!d[7]&!d[6]))))&d[5]&d[4]))&(((!((d[3]&d[2])|(!d[3]&!d[2])))&d[1]&d[0])|(!((d[1]&d[0])|(!d[1]&!d
[0]))&d[3]&d[2]))))|(((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&!d[
9]&!d[8]))&!(d[5]&d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]
&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[5]&!d[4]))&((!((d[3]&d[2])|(!d[3]&!d[2]))&!d[1]&!d[0]
)|(!((d[1]&d[0])|(!d[1]&!d[0]))&!d[3]&!d[2]))))|((d[3]&d[2]&!d[1]&!d[0]&((((!((d[9]&d[8])|(!d[9]&!
d[8]))&d[7]&d[6])|(!((d[7]&d[6])|(!d[7]&!d[6]))&d[9]&d[8]))&(d[5]|d[4]))|(((d[9]&d[8]&!d[7]&!d[6
])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&d[5]&d[4]
))))|((!d[3]&!d[2]&d[1]&d[0]&((((!((d[9]&d[8])|(!d[9]&!d[8]))&!d[7]&!d[6])|(!((d[7]&d[6])|(!d[7]&!
d[6]))&!d[9]&!d[8]))&!(d[5]&d[4]))|(((d[9]&d[8]&!d[7]&!d[6])|(d[7]&d[6]&!d[9]&!d[8])|(!((d[9]&d
[8])|(!d[9]&!d[8]))&!((d[7]&d[6])|(!d[7]&!d[6]))))&!d[5]&!d[4]))));
                        e[2] <=
((d[9]&d[8]&d[7]&!d[5]&!d[4]&((!d[3]&!d[2])|(((!((d[3]&d[2])|(!d[3]&!d[2]))&!d[1]&!d[0])|(!((d[1]
&d[0])|(!d[1]&!d[0]))&!d[3]&!d[2])))))|((!d[9]&!d[8]&!d[7]&d[5]&d[4]&((d[3]&d[2])|((((!((d[3]&d[
2])|(!d[3]&!d[2])))&d[1]&d[0])|(!((d[1]&d[0])|(!d[1]&!d[0]))&d[3]&d[2])))))|((d[7]&d[6]&d[5]&d[4
]&!d[3]&!d[2]&!d[1]))|((!d[7]&!d[6]&!d[5]&!d[4]&d[3]&d[2]&d[1]));
                        ce <= e ? 1'b1 : 1'b0;
                end
        end
end

endmodule
```

encoder_8b10.v

```
module encoder_8b10
(
	input wire clk,
	input wire rst,
	input wire en,
	input wire kin,
	input wire [7:0]din,
	output wire [9:0]dout,
	output wire disp,
	output wire kin_err
);

reg p;
reg ke;
reg [18:0]t;
reg [9:0]do;
wire [7:0]d;
wire k;

assign d = din;
assign k = kin;

assign dout = do;
assign disp = p;
assign kin_err = ke;

always @(posedge clk) begin
	if (rst) begin
		p <= 1'b0;
		ke <= 1'b0;
		do <= 10'b0;
	end else begin
		if (en == 1'b1) begin
			p <=
((d[5]&d[6]&d[7])|(!d[5]&!d[6]))^(p^(((d[4]&d[3]&!d[2]&!d[1]&!d[0])|(!d[4]&!((d[0]&d[1]&!d[2]
&!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!d[2]&!d[3]))))&!((!(
(d[0]&d[1])|(!d[0]&!d[1]))&d[2]&d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))&d[0]&d[1])))))|(k|(d[4]&!((d[0]
&d[1]&!d[2]&!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!d[2]&!
d[3]))))&!((!((d[0]&d[1])|(!d[0]&!d[1]))&!d[2]&!d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))&!d[0]&!d[1]))))
));
			ke <=
(k&(d[0]|d[1]|!d[2]|!d[3]|!d[4])&(!d[5]|!d[6]|!d[7]|!d[4]|!((!((d[0]&d[1])|(!d[0]&!d[1]))&d[2]&d[3])|(!
((d[2]&d[3])|(!d[2]&!d[3]))&d[0]&d[1]))));
			do[9] <= t[12]^t[0];
			do[8] <= t[12]^(t[1]|t[2]);
			do[7] <= t[12]^(t[3]|t[4]);
			do[6] <= t[12]^t[5];
			do[5] <= t[12]^(t[6]&t[7]);
			do[4] <= t[12]^(t[8]|t[9]|t[10]|t[11]);
			do[3] <= t[13]^(t[15]&!t[14]);
			do[2] <= t[13]^t[16];
			do[1] <= t[13]^t[17];
			do[0] <= t[13]^(t[18]|t[14]);
```

```
		end
	end
end

always @(posedge clk) begin
	if(rst) begin
		t <= 0;
	end else begin
		if (en == 1'b1) begin
			t[0] <= d[0];
			t[1] <= d[1]&!(d[0]&d[1]&d[2]&d[3]);
			t[2] <= (!d[0]&!d[1]&!d[2]&!d[3]);
			t[3] <= (!d[0]&!d[1]&!d[2]&!d[3])|d[2];
			t[4] <= d[4]&d[3]&!d[2]&!d[1]&!d[0];
			t[5] <= d[3]&!(d[0]&d[1]&d[2]);
			t[6] <=
d[4]|((!((d[0]&d[1])|(!d[0]&!d[1]))&!d[2]&!d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))&!d[0]&!d[1]));
			t[7] <= !(d[4]&d[3]&!d[2]&!d[1]&!d[0]);
			t[8] <=
(((d[0]&d[1]&!d[2]&!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!
d[2]&!d[3]))))&!d[4])|(d[4]&(d[0]&d[1]&d[2]&d[3]));
			t[9] <= d[4]&!d[3]&!d[2]&!(d[0]&d[1]);
			t[10] <= k&d[4]&d[3]&d[2]&!d[1]&!d[0];
			t[11] <= d[4]&!d[3]&d[2]&!d[1]&!d[0];
			t[12] <=
(((d[4]&d[3]&!d[2]&!d[1]&!d[0])|(!d[4]&!((d[0]&d[1]&!d[2]&!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[
0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!d[2]&!d[3]))))&!((!((d[0]&d[1])|(!d[0]&!d[1]))&d[2]&d[3])
|(!((d[2]&d[3])|(!d[2]&!d[3]))&d[0]&d[1]))))&!p)|((k|(d[4]&!((d[0]&d[1]&!d[2]&!d[3])|(d[2]&d[3]&
!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!d[2]&!d[3]))))&!((!((d[0]&d[1])|(!d[0]&!
d[1]))&!d[2]&!d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))&!d[0]&!d[1]))))|(!d[4]&!d[3]&d[2]&d[1]&d[0]))&
p);
			t[13] <=
(((!d[5]&!d[6])|(k&((d[5]&!d[6])|(!d[5]&d[6]))))&!(p^(((d[4]&d[3]&!d[2]&!d[1]&!d[0])|(!d[4]&!((d[
0]&d[1]&!d[2]&!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!d[2]
&!d[3]))))&!((!((d[0]&d[1])|(!d[0]&!d[1]))&d[2]&d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))&d[0]&d[1]))))|
(k|(d[4]&!((d[0]&d[1]&!d[2]&!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]
&d[3])|(!d[2]&!d[3]))))&!((!((d[0]&d[1])|(!d[0]&!d[1]))&!d[2]&!d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))
&!d[0]&!d[1])))))))|((d[5]&d[6])&(p^(((d[4]&d[3]&!d[2]&!d[1]&!d[0])|(!d[4]&!((d[0]&d[1]&!d[2]&
!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!d[2]&!d[3]))))&!((!((d
[0]&d[1])|(!d[0]&!d[1]))&d[2]&d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))&d[0]&d[1]))))|(k|(d[4]&!((d[0]&
d[1]&!d[2]&!d[3])|(d[2]&d[3]&!d[0]&!d[1])|(!((d[0]&d[1])|(!d[0]&!d[1]))&!((d[2]&d[3])|(!d[2]&!d[
3]))))&!((!((d[0]&d[1])|(!d[0]&!d[1]))&!d[2]&!d[3])|(!((d[2]&d[3])|(!d[2]&!d[3]))&!d[0]&!d[1]))))))
);
			t[14] <=
d[5]&d[6]&d[7]&(k|(p?(!d[4]&d[3]&((!((d[0]&d[1])|(!d[0]&!d[1]))&d[2]&d[3])|(!((d[2]&d[3])|(!d[2]
&!d[3]))&d[0]&d[1])):(d[4]&!d[3]&((!((d[0]&d[1])|(!d[0]&!d[1]))&!d[2]&!d[3])|(!((d[2]&d[3])|(!d[
2]&!d[3]))&!d[0]&!d[1]))))));
			t[15] <= d[5];
			t[16] <= d[6]|(!d[5]&!d[6]&!d[7]);
			t[17] <= d[7];
			t[18] <= !d[7]&(d[6]^d[5]);
		end
	end
end
```

```
endmodule
```

Parallel_clock_serializer.v

```
module Parallel_Clock_Serializer (
  input       clk,      // System clock
    input       rst_n,     // Active-low reset
    input       load,      // Signal to load new 8-bit data
    input  [7:0] data_in,   // 8-bit parallel data input
    output      lvds_p,    // LVDS positive output
    output      lvds_n,    // LVDS negative output
    output      busy       // Indicates that transmission is in progress
);
    // Internal nets connecting the submodules.
    wire serializer_out;
    wire nrz_out;
    // Instantiate the 8-bit Serializer.
    Serializer8bit serializer_inst (
        .clk(clk),
        .rst_n(rst_n),
        .load(load),
        .data_in(data_in),
        .serial_out(serializer_out),
        .busy(busy)
    );
    // Instantiate the NRZ Encoder.
    NRZ_Encoder nrz_encoder_inst (
        .clk(clk),
        .rst_n(rst_n),
        .serial_in(serializer_out),
        .nrz_out(nrz_out)
    );
    // Instantiate the LVDS Driver.
    LVDS_Driver lvds_driver_inst (
        .data(nrz_out),
        .lvds_p(lvds_p),
        .lvds_n(lvds_n)
    );
endmodule
//=====================================================
// 8-bit Serializer Module
// Converts 8-bit parallel data into a serial bit stream.
//=====================================================
module Serializer8bit(
    input       clk,        // System clock
    input       rst_n,      // Active-low reset
    input       load,       // Load signal to capture new parallel data
    input  [7:0] data_in,     // 8-bit parallel data input
    output reg   serial_out,  // Serialized data output (LSB first)
    output reg   busy         // High when serialization is in progress
);
    reg [7:0] shift_reg;      // Shift register holding the data
    reg [2:0] bit_cnt;        // 3-bit counter to track transmitted bits
    always @(posedge clk or posedge rst_n) begin
        if (rst_n) begin
```

```
            shift_reg  <= 8'b0;
            bit_cnt    <= 3'b0;
            serial_out <= 1'b0;
            busy       <= 1'b0;
        end
        else if (load && !busy) begin
            // Load the 8-bit data and start the serialization process.
            shift_reg  <= data_in;
            bit_cnt    <= 3'b0;
            busy       <= 1'b1;
        end
        else if (busy) begin
            // Output the LSB first and shift the register.
            serial_out <= shift_reg[0];
            shift_reg  <= shift_reg >> 1;
            bit_cnt    <= bit_cnt + 1;

            // After transmitting 8 bits, end the busy state.
            if (bit_cnt == 3'b111)
                busy <= 1'b0;
        end
    end
endmodule
//=====================================================
// NRZ Encoder Module
// For NRZ, the output is simply the input data without any
// additional encoding. This module registers the input
// (and could be expanded for further processing if needed).
//=====================================================
module NRZ_Encoder(
    input  clk,
    input  rst_n,
    input  serial_in,  // Input serial data (from the serializer)
    output reg nrz_out // NRZ-encoded output (unchanged in this case)
);
    always @(posedge clk or posedge rst_n) begin
        if (rst_n)
            nrz_out <= 1'b0;
        else
            nrz_out <= serial_in;
    end
endmodule
//=====================================================
// LVDS Driver Module
// Converts a single-ended signal into a differential pair.
// In this simple behavioral model, the positive output is the
// data and the negative output is its complement.
//=====================================================
module LVDS_Driver(
    input  wire data,       // Single-ended input data (NRZ encoded)
    output wire lvds_p,     // LVDS positive output
    output wire lvds_n      // LVDS negative output (complement of data)
);
    assign lvds_p = data;
    assign lvds_n = ~data;
```

```
endmodule
```

Parallel_clock_deserializer.v

```
// Top-level module to integrate NRZ Decoder and Parallel Clock Deserializer
module Parallel_Clock_Deserializer # (
    parameter N = 8  // Number of bits in parallel output
)(
    input  wire clk,        // Parallel clock
    input  wire reset,      // Active-high reset
    input  wire nrz_in,     // NRZ encoded input
    output wire  [N-1:0] data_out, // Parallel data output
    output wire  valid_out      // Data valid signal
);
    wire decoded_serial;
    // Instantiate NRZ Decoder
    NRZ_Decoder nrz_decoder_inst (
        .clk(clk),
        .reset(reset),
        .nrz_in(nrz_in),
        .serial_out(decoded_serial)
    );

    // Instantiate Deserializer (Fixed recursive instantiation issue)
    Deserializer #(.N(N)) deserializer_inst (
        .clk(clk),
        .reset(reset),
        .serial_in(decoded_serial),
        .parallel_out(data_out),
        .valid(valid_out)
    );
endmodule
// Parallel Clock Deserializer Module (Fixed bit shift timing and counter logic)
module Deserializer # (
    parameter N = 8  // Number of bits in parallel output
```

```
)(
    input  wire clk,        // Parallel clock
    input  wire reset,      // Active-high reset
    input  wire serial_in,  // Serial data input
    output reg  [N-1:0] parallel_out, // Parallel data output
    output reg  valid       // Data valid signal
);
    reg [N-1:0] shift_reg;
    reg [3:0] bit_count;  // Supports up to 16-bit serialization
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            shift_reg  <= 0;
            bit_count  <= 0;
            valid      <= 0;
            parallel_out <= 0;
        end else begin
            // Shift in new serial data bit
            shift_reg <= {shift_reg[N-2:0], serial_in};
            // Update bit counter with correct reset condition
            if (bit_count == (N-1)) begin
                bit_count <= 0;
                parallel_out <= {shift_reg[N-2:0], serial_in}; // Correct final bit capture
                valid <= 1;
            end else begin
                bit_count <= bit_count + 1;
                valid <= 0;
            end
        end
    end
endmodule
// NRZ Decoder Module (No changes needed)
module NRZ_Decoder (
```

```
    input  wire clk,       // Clock signal
    input  wire reset,     // Active-high reset
    input  wire nrz_in,    // NRZ encoded input
    output reg  serial_out // Decoded serial output
);
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            serial_out <= 0;
        end else begin
            // Directly assign the NRZ input to the serial output
            serial_out <= nrz_in;
        end
    end
endmodule
```

NCO.v

```
module NCO (
    input wire clk,
    input wire rst,
    input wire [31:0] freq_word, // Reduced to 10-bit
    output reg signed [7:0] sine_wave
);
    reg [31:0] phase_accumulator;
    wire [8:0] lut_index; // 9-bit index for 512 samples
    // Phase accumulator logic
    always @(posedge clk or posedge rst) begin
        if (rst)
            phase_accumulator <= 32'd0;
        else
            phase_accumulator <= phase_accumulator + freq_word; // Scale to 32-bit
    end
    // Extract upper bits to use as LUT index
    assign lut_index = phase_accumulator[31:23]; // 9-bit index for 512 samples
    // Sine LUT (512-point LUT)
```

```
reg signed [7:0] sine_LUT [0:511];

initial begin

sine_LUT[0] = 8'h00; sine_LUT[1] = 8'h01; sine_LUT[2] = 8'h03; sine_LUT[3] = 8'h04; sine_LUT[4] = 8'h06;
sine_LUT[5] = 8'h07; sine_LUT[6] = 8'h09; sine_LUT[7] = 8'h0A;

sine_LUT[8] = 8'h0C; sine_LUT[9] = 8'h0D; sine_LUT[10] = 8'h0F; sine_LUT[11] = 8'h11; sine_LUT[12] =
8'h12; sine_LUT[13] = 8'h14; sine_LUT[14] = 8'h15; sine_LUT[15] = 8'h17;

sine_LUT[16] = 8'h18; sine_LUT[17] = 8'h1A; sine_LUT[18] = 8'h1B; sine_LUT[19] = 8'h1D; sine_LUT[20] =
8'h1E; sine_LUT[21] = 8'h20; sine_LUT[22] = 8'h21; sine_LUT[23] = 8'h23;

sine_LUT[24] = 8'h24; sine_LUT[25] = 8'h26; sine_LUT[26] = 8'h27; sine_LUT[27] = 8'h29; sine_LUT[28] =
8'h2A; sine_LUT[29] = 8'h2C; sine_LUT[30] = 8'h2D; sine_LUT[31] = 8'h2F;

sine_LUT[32] = 8'h30; sine_LUT[33] = 8'h32; sine_LUT[34] = 8'h33; sine_LUT[35] = 8'h34; sine_LUT[36] =
8'h36; sine_LUT[37] = 8'h37; sine_LUT[38] = 8'h39; sine_LUT[39] = 8'h3A;

sine_LUT[40] = 8'h3B; sine_LUT[41] = 8'h3D; sine_LUT[42] = 8'h3E; sine_LUT[43] = 8'h3F; sine_LUT[44] =
8'h41; sine_LUT[45] = 8'h42; sine_LUT[46] = 8'h43; sine_LUT[47] = 8'h45;

sine_LUT[48] = 8'h46; sine_LUT[49] = 8'h47; sine_LUT[50] = 8'h49; sine_LUT[51] = 8'h4A; sine_LUT[52] =
8'h4B; sine_LUT[53] = 8'h4C; sine_LUT[54] = 8'h4E; sine_LUT[55] = 8'h4F;

sine_LUT[56] = 8'h50; sine_LUT[57] = 8'h51; sine_LUT[58] = 8'h52; sine_LUT[59] = 8'h54; sine_LUT[60] =
8'h55; sine_LUT[61] = 8'h56; sine_LUT[62] = 8'h57; sine_LUT[63] = 8'h58;

sine_LUT[64] = 8'h59; sine_LUT[65] = 8'h5A; sine_LUT[66] = 8'h5B; sine_LUT[67] = 8'h5D; sine_LUT[68] =
8'h5E; sine_LUT[69] = 8'h5F; sine_LUT[70] = 8'h60; sine_LUT[71] = 8'h61;

sine_LUT[72] = 8'h62; sine_LUT[73] = 8'h63; sine_LUT[74] = 8'h64; sine_LUT[75] = 8'h65; sine_LUT[76] =
8'h66; sine_LUT[77] = 8'h66; sine_LUT[78] = 8'h67; sine_LUT[79] = 8'h68;

sine_LUT[80] = 8'h69; sine_LUT[81] = 8'h6A; sine_LUT[82] = 8'h6B; sine_LUT[83] = 8'h6C; sine_LUT[84] =
8'h6C; sine_LUT[85] = 8'h6D; sine_LUT[86] = 8'h6E; sine_LUT[87] = 8'h6F;

sine_LUT[88] = 8'h70; sine_LUT[89] = 8'h70; sine_LUT[90] = 8'h71; sine_LUT[91] = 8'h72; sine_LUT[92] =
8'h72; sine_LUT[93] = 8'h73; sine_LUT[94] = 8'h74; sine_LUT[95] = 8'h74;

sine_LUT[96] = 8'h75; sine_LUT[97] = 8'h75; sine_LUT[98] = 8'h76; sine_LUT[99] = 8'h77; sine_LUT[100] =
8'h77; sine_LUT[101] = 8'h78; sine_LUT[102] = 8'h78; sine_LUT[103] = 8'h79;

sine_LUT[104] = 8'h79; sine_LUT[105] = 8'h79; sine_LUT[106] = 8'h7A; sine_LUT[107] = 8'h7A;
sine_LUT[108] = 8'h7B; sine_LUT[109] = 8'h7B; sine_LUT[110] = 8'h7B; sine_LUT[111] = 8'h7C;

sine_LUT[112] = 8'h7C; sine_LUT[113] = 8'h7C; sine_LUT[114] = 8'h7D; sine_LUT[115] = 8'h7D;
sine_LUT[116] = 8'h7D; sine_LUT[117] = 8'h7D; sine_LUT[118] = 8'h7E; sine_LUT[119] = 8'h7E;

sine_LUT[120] = 8'h7E; sine_LUT[121] = 8'h7E; sine_LUT[122] = 8'h7E; sine_LUT[123] = 8'h7E;
sine_LUT[124] = 8'h7E; sine_LUT[125] = 8'h7E; sine_LUT[126] = 8'h7E; sine_LUT[127] = 8'h7E;

sine_LUT[128] = 8'h7F; sine_LUT[129] = 8'h7E; sine_LUT[130] = 8'h7E; sine_LUT[131] = 8'h7E;
sine_LUT[132] = 8'h7E; sine_LUT[133] = 8'h7E; sine_LUT[134] = 8'h7E; sine_LUT[135] = 8'h7E;

sine_LUT[136] = 8'h7E; sine_LUT[137] = 8'h7E; sine_LUT[138] = 8'h7E; sine_LUT[139] = 8'h7D;
sine_LUT[140] = 8'h7D; sine_LUT[141] = 8'h7D; sine_LUT[142] = 8'h7D; sine_LUT[143] = 8'h7C;

sine_LUT[144] = 8'h7C; sine_LUT[145] = 8'h7C; sine_LUT[146] = 8'h7B; sine_LUT[147] = 8'h7B;
sine_LUT[148] = 8'h7B; sine_LUT[149] = 8'h7A; sine_LUT[150] = 8'h7A; sine_LUT[151] = 8'h79;

sine_LUT[152] = 8'h79; sine_LUT[153] = 8'h79; sine_LUT[154] = 8'h78; sine_LUT[155] = 8'h78;
sine_LUT[156] = 8'h77; sine_LUT[157] = 8'h77; sine_LUT[158] = 8'h76; sine_LUT[159] = 8'h75;
```

```
sine_LUT[160] = 8'h75; sine_LUT[161] = 8'h74; sine_LUT[162] = 8'h74; sine_LUT[163] = 8'h73;
sine_LUT[164] = 8'h72; sine_LUT[165] = 8'h72; sine_LUT[166] = 8'h71; sine_LUT[167] = 8'h70;

sine_LUT[168] = 8'h70; sine_LUT[169] = 8'h6F; sine_LUT[170] = 8'h6E; sine_LUT[171] = 8'h6D;
sine_LUT[172] = 8'h6C; sine_LUT[173] = 8'h6C; sine_LUT[174] = 8'h6B; sine_LUT[175] = 8'h6A;

sine_LUT[176] = 8'h69; sine_LUT[177] = 8'h68; sine_LUT[178] = 8'h67; sine_LUT[179] = 8'h66;
sine_LUT[180] = 8'h66; sine_LUT[181] = 8'h65; sine_LUT[182] = 8'h64; sine_LUT[183] = 8'h63;

sine_LUT[184] = 8'h62; sine_LUT[185] = 8'h61; sine_LUT[186] = 8'h60; sine_LUT[187] = 8'h5F;
sine_LUT[188] = 8'h5E; sine_LUT[189] = 8'h5D; sine_LUT[190] = 8'h5B; sine_LUT[191] = 8'h5A;

sine_LUT[192] = 8'h59; sine_LUT[193] = 8'h58; sine_LUT[194] = 8'h57; sine_LUT[195] = 8'h56;
sine_LUT[196] = 8'h55; sine_LUT[197] = 8'h54; sine_LUT[198] = 8'h52; sine_LUT[199] = 8'h51;

sine_LUT[200] = 8'h50; sine_LUT[201] = 8'h4F; sine_LUT[202] = 8'h4E; sine_LUT[203] = 8'h4C;
sine_LUT[204] = 8'h4B; sine_LUT[205] = 8'h4A; sine_LUT[206] = 8'h49; sine_LUT[207] = 8'h47;

sine_LUT[208] = 8'h46; sine_LUT[209] = 8'h45; sine_LUT[210] = 8'h43; sine_LUT[211] = 8'h42;
sine_LUT[212] = 8'h41; sine_LUT[213] = 8'h3F; sine_LUT[214] = 8'h3E; sine_LUT[215] = 8'h3D;

sine_LUT[216] = 8'h3B; sine_LUT[217] = 8'h3A; sine_LUT[218] = 8'h39; sine_LUT[219] = 8'h37;
sine_LUT[220] = 8'h36; sine_LUT[221] = 8'h34; sine_LUT[222] = 8'h33; sine_LUT[223] = 8'h32;

sine_LUT[224] = 8'h30; sine_LUT[225] = 8'h2F; sine_LUT[226] = 8'h2D; sine_LUT[227] = 8'h2C;
sine_LUT[228] = 8'h2A; sine_LUT[229] = 8'h29; sine_LUT[230] = 8'h27; sine_LUT[231] = 8'h26;

sine_LUT[232] = 8'h24; sine_LUT[233] = 8'h23; sine_LUT[234] = 8'h21; sine_LUT[235] = 8'h20;
sine_LUT[236] = 8'h1E; sine_LUT[237] = 8'h1D; sine_LUT[238] = 8'h1B; sine_LUT[239] = 8'h1A;

sine_LUT[240] = 8'h18; sine_LUT[241] = 8'h17; sine_LUT[242] = 8'h15; sine_LUT[243] = 8'h14;
sine_LUT[244] = 8'h12; sine_LUT[245] = 8'h11; sine_LUT[246] = 8'h0F; sine_LUT[247] = 8'h0D;

sine_LUT[248] = 8'h0C; sine_LUT[249] = 8'h0A; sine_LUT[250] = 8'h09; sine_LUT[251] = 8'h07;
sine_LUT[252] = 8'h06; sine_LUT[253] = 8'h04; sine_LUT[254] = 8'h03; sine_LUT[255] = 8'h01;

sine_LUT[256] = 8'h00; sine_LUT[257] = 8'hFF; sine_LUT[258] = 8'hFD; sine_LUT[259] = 8'hFC;
sine_LUT[260] = 8'hFA; sine_LUT[261] = 8'hF9; sine_LUT[262] = 8'hF7; sine_LUT[263] = 8'hF6;

sine_LUT[264] = 8'hF4; sine_LUT[265] = 8'hF3; sine_LUT[266] = 8'hF1; sine_LUT[267] = 8'hEF;
sine_LUT[268] = 8'hEE; sine_LUT[269] = 8'hEC; sine_LUT[270] = 8'hEB; sine_LUT[271] = 8'hE9;

sine_LUT[272] = 8'hE8; sine_LUT[273] = 8'hE6; sine_LUT[274] = 8'hE5; sine_LUT[275] = 8'hE3;
sine_LUT[276] = 8'hE2; sine_LUT[277] = 8'hE0; sine_LUT[278] = 8'hDF; sine_LUT[279] = 8'hDD;

sine_LUT[280] = 8'hDC; sine_LUT[281] = 8'hDA; sine_LUT[282] = 8'hD9; sine_LUT[283] = 8'hD7;
sine_LUT[284] = 8'hD6; sine_LUT[285] = 8'hD4; sine_LUT[286] = 8'hD3; sine_LUT[287] = 8'hD1;

sine_LUT[288] = 8'hD0; sine_LUT[289] = 8'hCE; sine_LUT[290] = 8'hCD; sine_LUT[291] = 8'hCC;
sine_LUT[292] = 8'hCA; sine_LUT[293] = 8'hC9; sine_LUT[294] = 8'hC7; sine_LUT[295] = 8'hC6;

sine_LUT[296] = 8'hC5; sine_LUT[297] = 8'hC3; sine_LUT[298] = 8'hC2; sine_LUT[299] = 8'hC1;
sine_LUT[300] = 8'hBF; sine_LUT[301] = 8'hBE; sine_LUT[302] = 8'hBD; sine_LUT[303] = 8'hBB;

sine_LUT[304] = 8'hBA; sine_LUT[305] = 8'hB9; sine_LUT[306] = 8'hB7; sine_LUT[307] = 8'hB6;
sine_LUT[308] = 8'hB5; sine_LUT[309] = 8'hB4; sine_LUT[310] = 8'hB2; sine_LUT[311] = 8'hB1;

sine_LUT[312] = 8'hB0; sine_LUT[313] = 8'hAF; sine_LUT[314] = 8'hAE; sine_LUT[315] = 8'hAC;
sine_LUT[316] = 8'hAB; sine_LUT[317] = 8'hAA; sine_LUT[318] = 8'hA9; sine_LUT[319] = 8'hA8;

sine_LUT[320] = 8'hA7; sine_LUT[321] = 8'hA6; sine_LUT[322] = 8'hA5; sine_LUT[323] = 8'hA3;
sine_LUT[324] = 8'hA2; sine_LUT[325] = 8'hA1; sine_LUT[326] = 8'hA0; sine_LUT[327] = 8'h9F;
```

```
sine_LUT[328] = 8'h9E; sine_LUT[329] = 8'h9D; sine_LUT[330] = 8'h9C; sine_LUT[331] = 8'h9B;
sine_LUT[332] = 8'h9A; sine_LUT[333] = 8'h9A; sine_LUT[334] = 8'h99; sine_LUT[335] = 8'h98;

sine_LUT[336] = 8'h97; sine_LUT[337] = 8'h96; sine_LUT[338] = 8'h95; sine_LUT[339] = 8'h94;
sine_LUT[340] = 8'h94; sine_LUT[341] = 8'h93; sine_LUT[342] = 8'h92; sine_LUT[343] = 8'h91;

sine_LUT[344] = 8'h90; sine_LUT[345] = 8'h90; sine_LUT[346] = 8'h8F; sine_LUT[347] = 8'h8E;
sine_LUT[348] = 8'h8E; sine_LUT[349] = 8'h8D; sine_LUT[350] = 8'h8C; sine_LUT[351] = 8'h8C;

sine_LUT[352] = 8'h8B; sine_LUT[353] = 8'h8B; sine_LUT[354] = 8'h8A; sine_LUT[355] = 8'h89;
sine_LUT[356] = 8'h89; sine_LUT[357] = 8'h88; sine_LUT[358] = 8'h88; sine_LUT[359] = 8'h87;

sine_LUT[360] = 8'h87; sine_LUT[361] = 8'h87; sine_LUT[362] = 8'h86; sine_LUT[363] = 8'h86;
sine_LUT[364] = 8'h85; sine_LUT[365] = 8'h85; sine_LUT[366] = 8'h85; sine_LUT[367] = 8'h84;

sine_LUT[368] = 8'h84; sine_LUT[369] = 8'h84; sine_LUT[370] = 8'h83; sine_LUT[371] = 8'h83;
sine_LUT[372] = 8'h83; sine_LUT[373] = 8'h83; sine_LUT[374] = 8'h82; sine_LUT[375] = 8'h82;

sine_LUT[376] = 8'h82; sine_LUT[377] = 8'h82; sine_LUT[378] = 8'h82; sine_LUT[379] = 8'h82;
sine_LUT[380] = 8'h82; sine_LUT[381] = 8'h82; sine_LUT[382] = 8'h82; sine_LUT[383] = 8'h82;

sine_LUT[384] = 8'h81; sine_LUT[385] = 8'h82; sine_LUT[386] = 8'h82; sine_LUT[387] = 8'h82;
sine_LUT[388] = 8'h82; sine_LUT[389] = 8'h82; sine_LUT[390] = 8'h82; sine_LUT[391] = 8'h82;

sine_LUT[392] = 8'h82; sine_LUT[393] = 8'h82; sine_LUT[394] = 8'h82; sine_LUT[395] = 8'h83;
sine_LUT[396] = 8'h83; sine_LUT[397] = 8'h83; sine_LUT[398] = 8'h83; sine_LUT[399] = 8'h84;

sine_LUT[400] = 8'h84; sine_LUT[401] = 8'h84; sine_LUT[402] = 8'h85; sine_LUT[403] = 8'h85;
sine_LUT[404] = 8'h85; sine_LUT[405] = 8'h86; sine_LUT[406] = 8'h86; sine_LUT[407] = 8'h87;

sine_LUT[408] = 8'h87; sine_LUT[409] = 8'h87; sine_LUT[410] = 8'h88; sine_LUT[411] = 8'h88;
sine_LUT[412] = 8'h89; sine_LUT[413] = 8'h89; sine_LUT[414] = 8'h8A; sine_LUT[415] = 8'h8B;

sine_LUT[416] = 8'h8B; sine_LUT[417] = 8'h8C; sine_LUT[418] = 8'h8C; sine_LUT[419] = 8'h8D;
sine_LUT[420] = 8'h8E; sine_LUT[421] = 8'h8E; sine_LUT[422] = 8'h8F; sine_LUT[423] = 8'h90;

sine_LUT[424] = 8'h90; sine_LUT[425] = 8'h91; sine_LUT[426] = 8'h92; sine_LUT[427] = 8'h93;
sine_LUT[428] = 8'h94; sine_LUT[429] = 8'h94; sine_LUT[430] = 8'h95; sine_LUT[431] = 8'h96;

sine_LUT[432] = 8'h97; sine_LUT[433] = 8'h98; sine_LUT[434] = 8'h99; sine_LUT[435] = 8'h9A;
sine_LUT[436] = 8'h9A; sine_LUT[437] = 8'h9B; sine_LUT[438] = 8'h9C; sine_LUT[439] = 8'h9D;

sine_LUT[440] = 8'h9E; sine_LUT[441] = 8'h9F; sine_LUT[442] = 8'hA0; sine_LUT[443] = 8'hA1;
sine_LUT[444] = 8'hA2; sine_LUT[445] = 8'hA3; sine_LUT[446] = 8'hA5; sine_LUT[447] = 8'hA6;

sine_LUT[448] = 8'hA7; sine_LUT[449] = 8'hA8; sine_LUT[450] = 8'hA9; sine_LUT[451] = 8'hAA;
sine_LUT[452] = 8'hAB; sine_LUT[453] = 8'hAC; sine_LUT[454] = 8'hAE; sine_LUT[455] = 8'hAF;

sine_LUT[456] = 8'hB0; sine_LUT[457] = 8'hB1; sine_LUT[458] = 8'hB2; sine_LUT[459] = 8'hB4;
sine_LUT[460] = 8'hB5; sine_LUT[461] = 8'hB6; sine_LUT[462] = 8'hB7; sine_LUT[463] = 8'hB9;

sine_LUT[464] = 8'hBA; sine_LUT[465] = 8'hBB; sine_LUT[466] = 8'hBD; sine_LUT[467] = 8'hBE;
sine_LUT[468] = 8'hBF; sine_LUT[469] = 8'hC1; sine_LUT[470] = 8'hC2; sine_LUT[471] = 8'hC3;

sine_LUT[472] = 8'hC5; sine_LUT[473] = 8'hC6; sine_LUT[474] = 8'hC7; sine_LUT[475] = 8'hC9;
sine_LUT[476] = 8'hCA; sine_LUT[477] = 8'hCC; sine_LUT[478] = 8'hCD; sine_LUT[479] = 8'hCE;

sine_LUT[480] = 8'hD0; sine_LUT[481] = 8'hD1; sine_LUT[482] = 8'hD3; sine_LUT[483] = 8'hD4;
sine_LUT[484] = 8'hD6; sine_LUT[485] = 8'hD7; sine_LUT[486] = 8'hD9; sine_LUT[487] = 8'hDA;

sine_LUT[488] = 8'hDC; sine_LUT[489] = 8'hDD; sine_LUT[490] = 8'hDF; sine_LUT[491] = 8'hE0;
sine_LUT[492] = 8'hE2; sine_LUT[493] = 8'hE3; sine_LUT[494] = 8'hE5; sine_LUT[495] = 8'hE6;
```

```
sine_LUT[496] = 8'hE8; sine_LUT[497] = 8'hE9; sine_LUT[498] = 8'hEB; sine_LUT[499] = 8'hEC;
sine_LUT[500] = 8'hEE; sine_LUT[501] = 8'hEF; sine_LUT[502] = 8'hF1; sine_LUT[503] = 8'hF3;

sine_LUT[504] = 8'hF4; sine_LUT[505] = 8'hF6; sine_LUT[506] = 8'hF7; sine_LUT[507] = 8'hF9;
sine_LUT[508] = 8'hFA; sine_LUT[509] = 8'hFC; sine_LUT[510] = 8'hFD; sine_LUT[511] = 8'hFF;

end



    always @(posedge clk) begin

        sine_wave <= sine_LUT[lut_index]; // Output corresponding sine value

    end

endmodule
```

Hybride_CDR.v

```
module Hybrid_CDR (

    input  wire clk_400MHz,      // 400 MHz reference clock

    input  wire reset,           // Active-high reset

    input  wire serial_in,       // Incoming serial data stream

    output reg  recovered_clk,   // Recovered clock

    output reg  recovered_data   // Recovered serial data

);

    //-----------------------------------------------------------------------

    // Parameters for NCO and Phase Detector

    //-----------------------------------------------------------------------

    parameter NOM_FREQ_WORD = 32'd67108864;  // Adjusted for 400 MHz reference clock

    parameter CORR_DELTA     = 32'd10000;    // Fine-tuning adjustment step

    //-----------------------------------------------------------------------

    // Internal Signals

    //-----------------------------------------------------------------------

    reg [31:0] freq_word;         // NCO frequency word (tunable)

    reg serial_in_d1, serial_in_d2; // Synchronization registers for edge detection

    reg edge_detect;              // Edge detection flag

    reg signed [31:0] phase_error;// Signed phase error output from BBPD

    wire signed [15:0] sine_wave; // NCO sine wave output
```

```
reg recovered_clk_delayed;    // Delayed version of recovered clock for better sampling
reg recovered_clk_sync;       // Extra sync stage to stabilize recovered clock
//-----------------------------------------------------------------------------
// Edge Detection Logic (Fixing Glitch Issues)
//-----------------------------------------------------------------------------
always @(posedge clk_400MHz) begin
    serial_in_d1 <= serial_in;
    serial_in_d2 <= serial_in_d1;
    edge_detect <= (serial_in_d1 != serial_in_d2);
end
//-----------------------------------------------------------------------------
// NCO Instance (Clock Recovery using LUT-based NCO)
//-----------------------------------------------------------------------------
NCO nco_inst (
    .clk(clk_400MHz),
    .rst(reset),
    .freq_word(freq_word),
    .sine_wave(sine_wave)
);
// Generate recovered clock from NCO output
always @(posedge clk_400MHz) begin
    recovered_clk <= sine_wave[15]; // Use MSB of sine wave as recovered clock
    recovered_clk_sync <= recovered_clk; // Sync stage to stabilize clock
    recovered_clk_delayed <= recovered_clk_sync; // Additional delay for better sampling
end
//-----------------------------------------------------------------------------
// Bang-Bang Phase Detector (BBPD) Logic
//-----------------------------------------------------------------------------
always @(posedge clk_400MHz or posedge reset) begin
    if (reset) begin
        freq_word    <= NOM_FREQ_WORD;
        recovered_data <= 0;
```

```
            phase_error  <= 0;
        end else begin
            // Adjust frequency word based on detected phase error
            if (edge_detect) begin
                if (recovered_clk_delayed)
                    freq_word <= freq_word + CORR_DELTA; // Small increase
                else
                    freq_word <= freq_word - CORR_DELTA; // Small decrease
            end
            // Data Recovery: Sample serial data at the falling edge of recovered clock
            if (!recovered_clk_delayed)
                recovered_data <= serial_in_d2;
        end
    end
endmodule
//-----------------------------------------------------------------------
// Numerically Controlled Oscillator (NCO) Module
//-----------------------------------------------------------------------
module NCO (
    input wire clk,
    input wire rst,
    input wire [31:0] freq_word,
    output reg signed [15:0] sine_wave
);
    reg [31:0] phase_accumulator;
    wire [11:0] lut_index;
    // Phase accumulator logic
    always @(posedge clk or posedge rst) begin
        if (rst)
            phase_accumulator <= 32'd0;
        else
            phase_accumulator <= phase_accumulator + freq_word;
```

```
    end
    // Extract upper bits to use as LUT index
    assign lut_index = phase_accumulator[31:20]; // 12-bit index
    // Sine LUT (Example: 4096-point LUT)
    reg signed [15:0] sine_LUT [0:4095];
    initial begin
        $readmemh("C:\\Users\\HP\\sine_lut.hex", sine_LUT); // Load sine values from LUT file
    end
    always @(posedge clk) begin
        sine_wave <= sine_LUT[lut_index]; // Output corresponding sine value
    end
endmodule
```

sigma_delta_dac.v

```
module sigma_delta_dac (
    input wire clk,                 // System clock
    input wire rst,                 // Active-low reset
    input wire signed [7:0] din,    // 8-bit input (digital sine wave)
    output reg dout                 // 1-bit DAC output
);
    // 9-bit integrator (8-bit input plus one extra bit)
    reg signed [8:0] integrator = 0;
    // 9-bit feedback (scaled from 8-bit range)
    reg signed [8:0] feedback = 0;
    always @(posedge clk or negedge rst) begin
        if (!rst) begin
            integrator <= 0;
            feedback <= 0;
            dout <= 0;
        end else begin
            // Integrator: sum input and subtract feedback
            integrator <= integrator + din - feedback;
            // Comparator: if the sign bit of the integrator (bit 8) is 0 (non-negative), output 1
            dout <= (integrator[8] == 0) ? 1'b1 : 1'b0;
```

```
            // Feedback: convert the 1-bit output back to 8-bit scale (extended to 9 bits)
            feedback <= dout ? 9'sd127 : -9'sd128;
        end
    end
endmodule
```

sigma_delta_filter.v

```
module sigma_delta_filter (
    input clk,
    input rst,
    input x_in,       // 1-bit input
    output reg y_out  // 1-bit output
);
    reg signed [7:0] integrator;  // 8-bit accumulator
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            integrator <= 0;
            y_out <= 0;
        end else begin
            // Integrator (accumulate input)
            integrator <= integrator + (x_in ? 1 : -1);
            // Comparator (1-bit output)
            y_out <= (integrator > 0) ? 1 : 0;
        end
    end
endmodule
```

Testbench codes

top_module_tx_tb.v

```
module top_module_tx_tb;
    // Inputs
    reg clk;
    reg rst;
    reg en;
    reg kin;
    reg [7:0] din;
    // Outputs
    wire serial_out;
    // Internal signals
    wire [9:0] encoder_dout;
    wire disp;
    wire kin_err;
    // Instantiate the DUT
    top_module_tx dut (
        .clk(clk),
        .rst(rst),
        .en(en),
        .kin(kin),
        .din(din),
        .serial_out(serial_out)
    );
    // Clock generation
    always #5 clk = ~clk;
    // Testbench logic
    initial begin
        // Initialize signals
        clk = 0;
        rst = 1;
        en = 0;
        kin = 0;
        din = 8'h00;
        // Apply reset
        #10 rst = 0;
        // Test case 1: Normal operation
        #20 en = 1;
        din = 8'hAA;
        kin = 0;
        #20 en = 0;
        // Test case 2: With kin input
        #20 en = 1;
        din = 8'h55;
        kin = 1;
        #20 en = 0;
        // Test case 3: With kin error
        #20 en = 1;
        din = 8'h00;
```

```
        kin = 1;
        #20 en = 0;
        // Finish simulation
        #100 $finish;
    end

    // Monitor signals
    always @(posedge clk) begin
        if (en) begin
            $display("Time: %t, din: %h, encoder_dout: %h, serial_out: %b", $time, din,
encoder_dout, serial_out);
        end
    end
endmodule
```

top_module_rx_tb.v

```
module top_module_rx_tb;

    // Inputs
    reg rst;
    reg clk;
    reg a_rx_serial;
    reg disparity_d;

    // Outputs
    wire [7:0] dout;
    wire disparity_q;
    wire code_err;
    wire kout;
    wire disp;
    wire disp_err;

    // Instantiate the DUT
    top_module_rx dut (
        .rst(rst),
        .clk(clk),
        .a_rx_serial(a_rx_serial),
        .disparity_d(disparity_d),
        .dout(dout),
        .code_err(code_err),
        .kout(kout),
        .disp(disp),
        .disp_err(disp_err)
    );

    // Clock generation
    always begin
        clk = 0;
        #5;
        clk = 1;
        #5;
    end

    // Declare lfsr outside the always block
    reg [3:0] lfsr;

    initial begin
```

```
        // Reset
        rst = 1;
        #10;
        rst = 0;

        // Initial value for a_rx_serial
        a_rx_serial = 1'b0;

        // Initialize lfsr
        lfsr = 4'b1111;

        // Monitor and display results
        $monitor("Time: %t, a_rx_serial: %b, dout: %b, code_err: %b, kout: %b, disp: %b, disp_err: %b", $time,
a_rx_serial, dout, code_err, kout, disp, disp_err);

        #500; // Run for some time
        $finish;
    end

    // Generate random data on the rising edge of the clock
    always @(posedge clk) begin
        if (~rst) begin
            lfsr <= {lfsr[2:0], lfsr[3] ^ lfsr[1]};
            a_rx_serial <= lfsr[0];
        end
    end

endmodule
```

Parallel_Clock_Serializer_tb.v

```
module Parallel_Clock_Serializer_tb;

    reg       clk;
    reg       rst_n;
    reg       load;
    reg  [7:0] data_in;
    wire      lvds_p;
    wire      lvds_n;
    wire      busy;

    // Instantiate the DUT (Device Under Test)
    Parallel_Clock_Serializer dut (
        .clk(clk),
        .rst_n(rst_n),
        .load(load),
        .data_in(data_in),
        .lvds_p(lvds_p),
        .lvds_n(lvds_n),
        .busy(busy)
    );

    // Clock generation
    always #5 clk = ~clk; // 10ns period (100MHz)

    // Test sequence
    initial begin
        // Initialize signals
        clk = 0;
        rst_n = 1;
        load = 0;
```

```
        data_in = 8'b00000000;

        // Apply reset
        #20 rst_n = 0;

        // Load first data
        #10 data_in = 8'b10101010;
        load = 1;
        #10 load = 0;

        // Wait for serialization to complete
        wait (!busy);

        // Load second data
        #50 data_in = 8'b11001100;
        load = 1;
        #10 load = 0;

        // Wait for serialization to complete
        wait (!busy);

        // End simulation
        #50 $stop;
    end

    // Monitor output
    initial begin
        $monitor("Time=%0t | Data_in=%b | LVDS_p=%b | LVDS_n=%b | Busy=%b",
                 $time, data_in, lvds_p, lvds_n, busy);
    end

endmodule
```

tb_Parallel_Clock_Deserializer.v

```
module tb_Parallel_Clock_Deserializer;
    parameter N = 8;

    reg clk;
    reg reset;
    reg nrz_in;
    wire [N-1:0] data_out;
    wire valid_out;

    // Instantiate the Parallel Clock Deserializer module
    Parallel_Clock_Deserializer #(.N(N)) uut (
        .clk(clk),
        .reset(reset),
        .nrz_in(nrz_in),
        .data_out(data_out),
        .valid_out(valid_out)
    );

    // Clock generation
    always #5 clk = ~clk; // 10ns period clock

    initial begin
        // Initialize signals
        clk = 0;
        reset = 1;
        nrz_in = 0;
```

```
        // Apply reset
        #20;
        reset = 0;

        // Send NRZ-encoded serial data
        send_serial_data(8'b10110011);
        send_serial_data(8'b11001100);
        send_serial_data(8'b11110000);

        // Wait for outputs to stabilize
        #100;

        // End simulation
        $stop;
    end

    // Task to send serial data
    task send_serial_data(input [N-1:0] serial_data);
        integer i;
        for (i = N-1; i >= 0; i = i - 1) begin
            nrz_in = serial_data[i];
            #10; // Wait for one clock cycle per bit
        end
    endtask

endmodule
```

NCO_tb.v

```
`timescale 1ns / 1ps
module NCO_tb;
    reg clk;
    reg rst;
    reg [31:0] freq_word;
    wire signed [7:0] sine_wave;
    // Instantiate the NCO module
    NCO uut (
        .clk(clk),
        .rst(rst),
        .freq_word(freq_word),
        .sine_wave(sine_wave)
    );
    // Clock generation
    always #5 clk = ~clk; // 100MHz clock (10ns period)
    initial begin
        // Initialize signals
        clk = 0;
        rst = 1;
        freq_word = 32'd100000; // Example frequency tuning word
        // Apply reset
        #10 rst = 0;
        // Run simulation for a sufficient duration
        #10000;
        // Finish simulation
        $stop;
    end
```

```
    // Monitor output
    initial begin
        $monitor("Time: %0t | Sine Wave Output: %d", $time, sine_wave);
    end
endmodule
```

Hybrid_CDR_tb.v

```
`timescale 1ns / 1ps
module Hybrid_CDR_tb;
    // Testbench Signals
    reg clk_400MHz;      // 400 MHz reference clock
    reg reset;           // Active-high reset
    reg serial_in;       // Simulated serial data input
    wire recovered_clk;  // Recovered clock output
    wire recovered_data; // Recovered data output
    // Instantiate the Hybrid_CDR module
    Hybrid_CDR uut (
        .clk_25MHz(clk_400MHz), // Change reference clock to 400 MHz
        .reset(reset),
        .serial_in(serial_in),
        .recovered_clk(recovered_clk),
        .recovered_data(recovered_data)
    );
    // Clock Generation (400 MHz)
    always #1.25 clk_400MHz = ~clk_400MHz; // 2.5 ns period -> 400 MHz
    // Serial Data Generation Task
    task send_serial_data(input [31:0] data, input integer bit_period);
        integer i;
        for (i = 31; i >= 0; i = i - 1) begin
            serial_in = data[i]; // Send each bit
            #bit_period;
        end
    endtask
    // Test Procedure
    initial begin
        // Initialize signals
        clk_400MHz = 0;
        reset = 1;
        serial_in = 0;
        // Apply Initial Reset Pulse
        #100;
        reset = 0;
        #100;
        // Send Serial Data (Example: 10110011101100111011001110110011)
        send_serial_data(32'b10110011101100111011001110110011, 80); // Simulated serial bits
        // Wait and Observe Output
        #5000;
        // Apply Another Reset Pulse
        reset = 1;
        #50;
        reset = 0;
        #100;
        // Send More Test Data
        send_serial_data(32'b11001100110011001100110011001100, 80);
```

```
      send_serial_data(32'b11110000111100001111000011110000, 80);
      // Wait and Observe Output
      #5000;
      // End Simulation
      $stop;
    end
endmodule
```

tb_sigma_delta_dac.v

```
`timescale 1ns / 1ps
module tb_sigma_delta_dac;
  // Clock and reset signals
  reg clk;
  reg rst;    // Active-high reset for the NCO
  reg rst_n;  // Active-low reset for the sigma_delta_dac_8bit
  // Frequency word for the NCO (adjust as needed)
  reg [31:0] freq_word;
  // Wires connecting the NCO and DAC
  wire signed [7:0] sine_wave; // Output of the NCO
  wire dac_out;              // Output of the sigma-delta DAC
  // Instantiate the NCO (assumes NCO has active-high reset)
  NCO nco_inst (
    .clk(clk),
    .rst(rst),
    .freq_word(freq_word),
    .sine_wave(sine_wave)
  );
  // Instantiate the sigma_delta_dac_8bit (active-low reset)
  sigma_delta_dac dac_inst (
    .clk(clk),
    .rst(rst_n),
    .din(sine_wave),
    .dout(dac_out)
  );
  // Clock generation: 100 MHz clock (10 ns period)
  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end

  // Test stimulus
  initial begin
    // Set frequency word; here 4294967 is chosen as an example (adjust as needed)
    freq_word = 32'd4294967;
    // Apply resets:
    // For the NCO: active-high reset, so set rst = 1.
    // For the DAC: active-low reset, so set rst_n = 0.
    rst   = 1;
    rst_n = 0;
    #20;  // Hold resets for 20 ns
    // Release resets:
```

```
    // NCO reset becomes inactive (0) and DAC reset becomes inactive (1)
    rst   = 0;
    rst_n = 1;
    // Run simulation for enough time to observe behavior
    #1000000;
    $finish;
  end
endmodule
```

testbench_decodePipe.v

```
`timescale 1ns / 1ps

module testbench_decodePipe;
  // Inputs
  reg clk;
  reg rst;
  reg en;
  reg [9:0] din;
  // Outputs
  wire [7:0] dout;
  wire kout;
  wire code_err;
  wire disp;
  wire disp_err;
  // Instantiate the Unit Under Test (UUT)
  decodePipe uut (
    .clk(clk),
    .rst(rst),
    .en(en),
    .din(din),
    .dout(dout),
    .kout(kout),
    .code_err(code_err),
    .disp(disp),
    .disp_err(disp_err)
  );
  // Clock generation
  always begin
    clk = 0;
    #5;
    clk = 1;
    #5;
  end
  // Test sequence
  initial begin
    // Initialize signals
    clk = 0;
    rst = 1;
    en = 0;
    din = 10'b0000000000;
    // Apply reset
    #10;
    rst = 0;
```

```
    // Apply enable and test vectors
    #10;
    en = 1;
    din = 10'b1001110100; // Test vector 1
    #20;
    din = 10'b1010110001; // Test vector 2
    #20;
    din = 10'b0101011010; // Test vector 3
    #20;
    din = 10'b1010100101; // Test vector 4
    #20;
    din = 10'b0011010110; // Test vector 5
    #20;
    din = 10'b1010010101; // Test vector 6
    // Disable module
    #20;
    en = 0;
    // Monitor outputs
    $monitor("Time=%t, din=%b, dout=%b, kout=%b, code_err=%b, disp=%b, disp_err=%b", $time,
din, dout, kout, code_err, disp, disp_err);
    // Finish simulation
    #100;
    $finish;
  end
endmodule
```

testbench_encoder_8b10.v

```
`timescale 1ns / 1ps

module testbench_encoder_8b10;

  // Inputs
  reg clk;
  reg rst;
  reg en;
  reg kin;
  reg [7:0] din;
  // Outputs
  wire [9:0] dout;
  wire disp;
  wire kin_err;
  // Instantiate the Unit Under Test (UUT)
  encoder_8b10 uut (
    .clk(clk),
    .rst(rst),
    .en(en),
    .kin(kin),
    .din(din),
    .dout(dout),
    .disp(disp),
    .kin_err(kin_err)
  );
  // Clock generation
```

```
  always begin
    clk = 0;
    #5;
    clk = 1;
    #5;
  end
  // Test sequence
  initial begin
    // Initialize signals
    clk = 0;
    rst = 1;
    en = 0;
    kin = 0;
    din = 8'b00000000;
    // Apply reset
    #10;
    rst = 0;
    // Apply enable and test vectors
    #10;
    en = 1;
    din = 8'b00000000; // Test vector 1
    #20;
    din = 8'b11111111; // Test vector 2
    #20;
    din = 8'b10101010; // Test vector 3
    #20;
    din = 8'b01010101; // Test vector 4
    #20;
    din = 8'b11001100; // Test vector 5
    #20;
    din = 8'b01000101; // Test vector 6
    // Disable module
    #20;
    en = 0;
    // Monitor outputs
    $monitor("Time=%t, din=%b, dout=%b, disp=%b, kin_err=%b", $time, din, dout, disp, kin_err);
    // Finish simulation
    #100;
    $finish;
  end
endmodule
```

Appendix C

Software setup

NC launch steps

Quartus-II-> just normal RTL coding and compilation

Modelsim-> functional simulation and verification

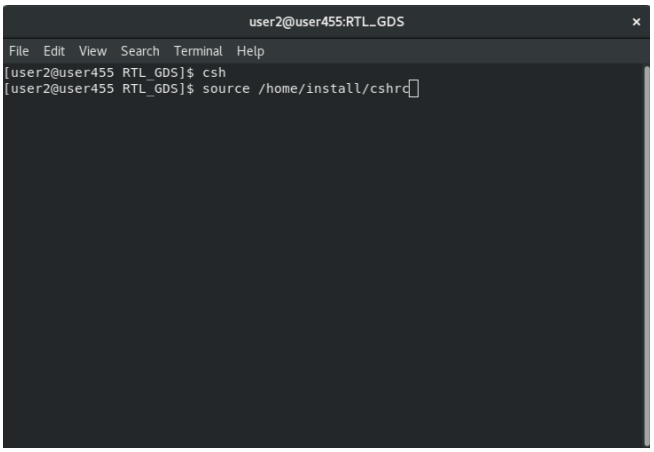
To launch the cadence software, follow few steps

Type these two commands

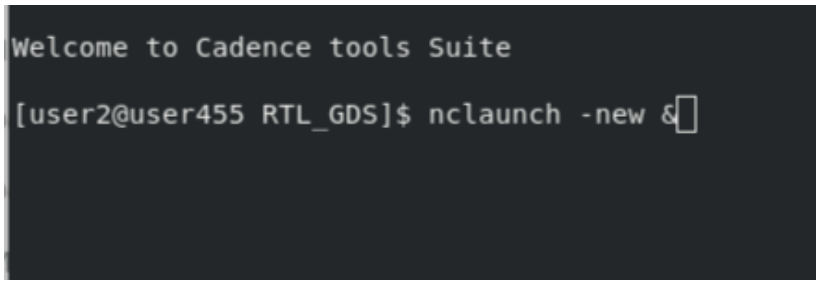
Csh

Source /home/install/cshrc

`

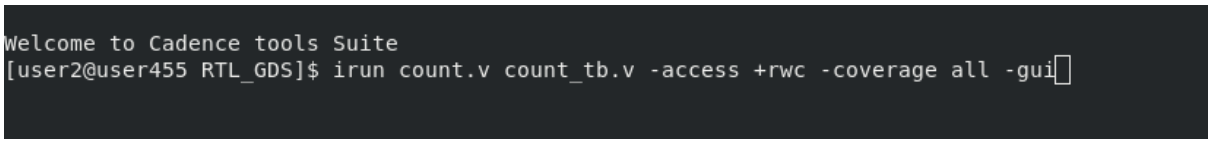


NC launch-> this software is also for functional simulation and verification, but it is provided by cadence



NC sim-> to analysis the verification coverage report

To open NC sim software type this command



Geneus-> **To complete the gate level synthesis and generate functional netlist file and constrain file which has the report of capacitance and timing of circuit and also for RTL synthesis**

Geneus steps

In cadence terminal-> geneus

And then type these commands to complete the prosses of synthesis and report generation:

@genus:root: 1> set_db init_lib_search_path /home/install/FOUNDRY/digital/90nm/dig/lib/

@genus:root: 2> read_libs slow.lib

@genus:root: 3> set_db init_hdl_search_path ./

@genus:root: 4> read_hdl count.v

@genus:root: 5> elaborate

@genus:root: 6> create_clock -name clk -period 10 -waveform {0 5} [get_ports "clk"]

@genus:root: 8> syn_generic

@genus:root: 9> write_hdl

@genus:root: 10> syn_map

@genus:root: 11> write_hdl

@genus:root: 12> report_area

@genus:root: 13> syn_opt

@genus:root: 14> report_area

@genus:root: 15> report_timing

@genus:root: 16> report_power

@genus:root: 17> write_hdl > count_nl.v

@genus:root: 18> write_sdc > count_tool.sdc

@genus:root: 19> report_area>area.rep

@genus:root: 20> report_timing > timing.rep

@genus:root: 21> report_power > power.rep

@genus:root: 22> gui_show

Innovus steps

Design Import (specify input files)

1. Make sure that you are in your main separate directory (e.g., Innovus)
2. At the Unix prompt, type: **innovus**
3. When the Innovus tool window appears, go to the menu bar and select File **Import Design** to get the Design Import window.
4. For the **Verilog Netlist**, click on the box with the dots **[…]** to open the **Netlist Files window.**
5. Click on the **">>"** button to expand the window to show the directories:
6. select the file **count_nl.v** which is generated by genus and click the **Add** button to add it to the Netlist Files list. Click Close to the Netlist Files window.
7. In the main window, for **Top Cell**, select "**Auto Assign**".
8. For **Technology/Physical Libraries,** select "**LEF File**". Click on the **[…]** button open the LEF Files window.
9. Select the path(home/install/FOUNDRY/digital/90nm/dig/lef) for LEF File and select third file
10. For **Power,** enter the following:
    a. Power Nets: **VDD**
    b. Ground Nets: **VSS**
11. Select **Create Analysis Configuration**
12. Select and right click at **Library sets** and select **New**
13. Give name **"max_timing"** and click button **"Add"**
14. For max timing select path (home/install/FOUNDRY/digital/90nm/dig/lib/slow.lib) click on **ADD**
15. Repeat step 12 to 14 with change name **"min_timing**" and select path for **fast.lib**
16. Select and right click at **Delay Corners** and select **New**
17. Give name **"max_delay"** and select library set with **max_timing,** click ok
18. Select and right click at **Delay Corners** and select **New**
19. Give name **"min_delay"** and select library set with **min_timing,** click
20. Select and right click at **Constraint Modes** and select **New**
21. Give name **"top"** and click button **"Add"**
22. Select .sdc file which is generated by genus rom directory, click ok
23. Select and right click at **Analysis Views** and select **New**
24. Give name **"setup"** and select delay corner **max_delay**
25. Select and right click at **Analysis Views** and select **New**
26. Give name **"hold"** and select delay corner **min_delay**
27. Select and right click at **Setup Analysis Views** and select **New**
28. Select analysis view **setup**
29. Select and right click at **Setup Analysis Views** and select **New**
30. Select analysis view **setup**
31. Select **save&close**
32. In import design window, select save keep name default and click ok
33. After click ok, check terminal(if any error is there then correct it before go ahead)

2 Floor-planning

2.1 Specify Floorplan

In Innovus tool menu bar, select **Floorplan**, **Specify Floorplan** to get the **Specify Floorplan** window.

1. Select core size **"aspect ratio"**1, 0.7, 0.5 ,2
2. Select core utilization: 0.6 to 0.8(industry standardize value)
3. in the **Basic** tab, select the following options:
    a. **Core Margins** – select Core to IO Boundary and set all margins to **7.**
4. Click on **OK**.

3 Power Planning

3.1 Connect Global Nets

In Innovus tool menu bar, select **Power, Connect Global Nets** to get the **Global Net Connections** Window.

1. In **Power Ground Connection**

    a. In the **Connect** area, select **Pin**

    b. In the **Scope** area, select **Apply All**

2. For each net **VDD** and **VSS**, do the following:

    a. Enter the net name (VDD) in the following boxes:

        i. "To Global Net"
        ii. "Pin Name(s)"

    a. Click on the "Add to List" button
    b. Repeat (a) and (b) for **VSS**

3. Click Apply, then click Cancel

3.2 Power Rings

In Innovus tool menu bar, select **Power, Power Planning, Add Ring** to get the **Add Rings** window.

1. For Net(s), enter **VDD** and **VSS** nets as follows:
    a. Click on folder icon to the right of the **Net(s)** box to get Net Selection window
    b. Select **VDD** and **VSS** from Possible Nets column
    c. Click **Add** to copy to Chosen Nets column
    d. Click OK

2. In **Ring Configuration,** select **metal9(9)H** for Top and Bottom, **metal8(8)V** for Left and Right.
    a. Keep width default and select button **update** for **spacing.**

b. Offset should be **“ Center in channel”**

3.3 Power Stripes

In Innovus tool menu bar, select **Power, Power Planning, Add Stripes** to get the **Add Stripes** window.

1. **Basic Tab**
   a. For Net(s), enter **VDD** and **VSS** nets as follows:
      i. Click on folder icon to the right of the **Net(s)** box to get Net Selection window
      ii. Select **VDD** and **VSS** from Possible Nets column
      iii. Click **Add** to copy to Chosen Nets column
      iv. Click OK

   b. In **Set Configuration,** select Layer **metal9(9)** for horizontal. Select **update** for spacing.

   c. Select **Number of sets** according to circuit(design) area
   (less area and higher number of set will be create violation in DRC and connectivity)

   d. keep other setting default and press button **Apply**

   e. Repeat (a) to (d) steps with change in layer **Metal8(8)** for vertical.

   f. ok.

3.4 Connect Power to Standard Cell Rows

In Innovus tool menu bar, select **Rout→Special Route** ,for nets enter **VDD** and **VSS** and click OK. (This will create power (VDD) and ground (VSS) rails for your standard cell rows)

4 Placing the Standard Cells

In Innovus tool menu bar, select **Place→ Place Standard Cell** to get the Place window.

1. Select **“Run Full Placement”** and “Include Pre-Place Optimization”
2. Select **Mode** option and the select **“placement with IO pins”**
3. Click OK

5 Timing reports

In Innovus toll menu bar, select **Timing → Timing reports**

(There are multiple stages where timing reports will different.)

a. Pre-CTS with setup and Pre-CTS with Hold
b. Post-CTS with setup and Post-CTS with Hold
c. Post-Route with setup and Post-Route with Hold

6 Routing

In Innovus tool menu bar, select **Route→NanoRoute→Route** to get the NanoRoute window.

Keep all option as default

6. **Optimization**

In Innovous go to tools→ **mode→ specify analysis mode→change to on chip variation and CPPR**

In Innovus tool bar, select **ECO → optimize Design**

There are different ways to optimization

a. Pre-CTS with setup
b. Post-CTS with setup and Post-CTS with Hold and both together

Post-Route with setup and Post-Route with Hold and both together