Design and implementation of 8B/10B SerDes circuit also understanding of its performance

**Project report**

In digital communication we need to convert data from serial to parallel form to transmit the data over serial line communication (wired or wireless) that’s why we use SerDes circuit which can convert data from parallel to serial and serial to parallel

**Types of SerDes architecture**

**Different type of SerDes architectures**

1. **Parallel Clock SerDes:** In this (older) architecture, the clock signal is transmitted separately and in parallel with the serial data. The PISO block within this SerDes uses this parallel clock to time the serialization process. This approach is simpler but suffers from clock skew issues at higher speeds.
2. **Embedded Clock SerDes:** Here, the clock information is embedded within the serial data stream itself. This is done using techniques like Manchester encoding or 8b/10b encoding. The receiver extracts the clock from the incoming data. The PISO in this SerDes works in conjunction with the clock recovery circuitry. This is more robust at high speeds.
3. **8b/10b SerDes:** This is a specific type of embedded clock SerDes. It uses the 8b/10b encoding scheme to embed the clock and also provides other benefits like DC balancing and run-length limitation. The PISO block is still present, but the encoding of the data before serialization (and decoding after deserialization) is the key feature. The PISO serializes the encoded 10-bit symbols.
4. **Bit Interleaved SerDes:** This focuses on how multiple serial streams are combined (interleaved) for higher bandwidth. The PISO blocks in this architecture are responsible for taking portions of the parallel data for each of the lower-speed serial streams that will be interleaved. It's less about the fundamental serialization process and more about multiplexing the serialized data.

**8B/10B SerDes circuit and its structure**

**Key Components:**

* **Parallel Data Input Buffer**: Receives multiple parallel input data lines.
* **Multiplexer (MUX):** Selects one input line at a time for transmission based on clock

signals.

* **Shift Register:** Converts parallel data into serial form by shifting bits sequentially.
* **Clock Divider/Generator:** Generates a high-speed clock signal to synchronize the bit

transmission rate.

* **Line Driver:** Amplifies and conditions the serial output signal for transmission over a communication channel.

**Advantage of converting data from parallel to serial**

* **Reduced Pin Count**: Minimizes the number of I/O pins needed.
* **High Data Rates:** Enables faster data transmission over fewer connections.
* **Long-Distance Communication:** Reliable data transmission over long distances with minimal signal degradation.

**Working of SerDes circuit**

1. Parallel data enters through multiple data lines.
2. The MUX or shift register serializes the data, sending one bit per clock cycle.
3. The output is a single high-speed serial stream.

**Design of serializer**

A serializer is a device or circuit that converts parallel data into a serial data stream. This process is crucial in digital communication systems and data transmission because it allows multiple data signals to be sent over a single data channel, reducing the number of transmission lines required.

**Effects of serialization on bit rate:**

The serialization process has a direct impact on the bit rate because it converts multiple

parallel data lines into a single high-speed serial data stream. Here's how serialization affects

the bit rate:

1. **Increased Bit Rate for Serialized Data:** In a parallel-to-serial conversion, the bit rate of the serial output must be higher to accommodate all the parallel data lines within the same time frame. If you have NNN parallel lines each operating at a clock frequency

fclockf\_{clock}fclock, the required bit rate for the serial output is:

Rserial=N×fclockR\_{serial} = N \times f\_{clock}Rserial=N×fclock

* + Example: If 8 parallel lines each carry data at 10 Mbps, the serial bit rate becomes:

Rserial=8×10 Mbps=80 MbpsR\_{serial} = 8 \times 10 \, \text{Mbps} = 8 text{Mbps}Rserial=8×10Mbps=80Mbps

1. **Transmission Bandwidth:** The higher serial bit rate demands a communication channel with a wider bandwidth to accommodate the increased data rate. Bandwidth constraints can lead to signal distortion, attenuation, or interference if not properly managed.
2. **Clock Frequency Challenges:** The serializer requires a clock that operates at a higher frequency proportional to the bit rate. This high-frequency clock can introduce challenges such as increased power consumption, electromagnetic interference (EMI), and jitter in the serial stream.
3. **Reduced Transmission Lines:** While serialization increases the bit rate, it reduces the number of physical transmission lines, which simplifies system design and lowers costs.

**Susceptibility reduction because of serialization:**

**Reduced Susceptibility to Skew Between Parallel Lines**

Skew occurs in parallel communication when signals on different lines arrive at slightly

different times due to variations in propagation delay, signal integrity, or clock alignment.

Serialization, by converting parallel data into a single serial stream, eliminates the need for

multiple parallel lines and therefore greatly reduces skew-related issues.

1. **Understanding Skew in Parallel Communication** Definition: Skew refers to the time difference between the arrival of signals on different parallel lines at the receiver. Causes:
   * Differing Line Lengths: Longer lines have greater propagation delays.
   * Signal Integrity: Crosstalk and interference can delay certain lines.
   * Clock Drift: Timing discrepancies between clock signals in the transmitter and receiver.
   * Material Variations: Differences in PCB trace impedance or routing.
   * Impact: At high data rates, even small timing mismatches can cause incorrect data interpretation. Requires complex de-skewing mechanisms to realign signals.

1. **How Serialization Eliminates Skew**

* Single Transmission Line: Serial communication uses only one data line, so there are no parallel lines to cause skew.
* Sequential Data Transfer: Bits are sent sequentially, one at a time, over the serial line, ensuring precise timing without inter-line discrepancies.
* Clock Embedding: Clock information is embedded in the serial data stream or sent as a separate line, allowing precise timing synchronization.

1. **Advantages of Reducing Skew** 
   * + **Improved Signal Integrity:** No timing mismatches between parallel lines to corrupt the data.
     + **Higher Data Rates:** Skew becomes more problematic at higher frequencies; serial communication avoids this limitation, enabling higher bit rates.
     + **Simplified Receiver Design:** Eliminates the need for de-skewing circuits, which realign parallel data lines at the receiver.
2. **Skew Challenges in High-Speed Parallel Systems** 
   * + In a parallel bus operating at high speeds, skew becomes increasingly difficult to manage:
     + **Example:** A 16-bit parallel bus operating at 500 MHz can tolerate only a few picoseconds of delay difference between lines. Beyond this, errors occur.
     + Serialization Solution: By using a single line, serialization avoids these challenges entirely.
3. **Example: Serial vs. Parallel Communication**

**Parallel Transmission (8 bits):**

* + - Data: [D7, D6, D5, D4, D3, D2, D1, D0]
    - Transmission: Each bit is sent on a separate line simultaneously.
    - Problem: If line D3 is delayed due to skew, the receiver may misinterpret the entire word.

**Serial Transmission:**

* + - Data: [D7 → D6 → D5 → D4 → D3 → D2 → D1 → D0]
    - Transmission: All bits are sent sequentially on one line.
    - Solution: No skew between lines, as there is only one line.

1. **Real-World Applications**

* **PCI Express (PCIe):** Replaced parallel PCI with serialized communication to eliminate skew issues at high data rates.
* **HDMI and DisplayPort:** Use serialization to ensure reliable high-definition video transmission without skew-related artifacts.

**Optimized Bandwidth Usage in Serialization**

Serialization optimizes the usage of bandwidth in communication channels by leveraging the full capacity of the transmission medium. Here's a theoretical explanation of how this is achieved:

1. **Sequential Bit Transmission**

Parallel Transmission:

* Transmits multiple bits simultaneously over separate lines, each consuming a portion of the total available bandwidth.
* Example: For an 8-bit bus, each line may use a fraction of the total channel bandwidth.

Serialization:

* Combines all bits into a single sequential stream that utilizes the entire channel bandwidth.
* the serialized stream is transmitted over a single line (or differential pair), fully utilizing the medium's frequency range.

1. **Avoiding Signal Redundancy**

Parallel Transmission Redundancy:

* + - * Parallel lines may introduce overhead due to skew compensation and synchronization signals, wasting bandwidth.

Serialization Efficiency:

* + - * A serializer encodes and transmits data sequentially without redundant synchronization signals for each bit line, conserving bandwidth.

1. **Efficient Encoding Techniques**

Serialization supports advanced encoding schemes to optimize bandwidth usage:

1. 8b/10b Encoding:
   * + Converts 8 data bits into 10 transmitted bits, balancing DC components and ensuring sufficient transitions for clock recovery.
     + Reduces the risk of signal degradation in high-speed channels.
2. PAM4 (Pulse Amplitude Modulation, 4 Levels):
   * Encodes two bits per symbol, doubling data throughput for the same channel bandwidth compared to NRZ (Non-Return to Zero) encoding.
3. **Reducing Crosstalk and Noise** 
   * Crosstalk and noise in parallel transmission can distort signals, forcing systems to operate at lower frequencies.
   * Serialization reduces the number of lines, minimizing interference and allowing the channel to operate at its maximum potential bandwidth.
4. **Full Utilization of Channel Capacity**

Serialization aligns with the Shannon-Hartley theorem, which defines the maximum data rate (CCC) a channel can achieve:

C=B⋅log⁡2(1+SNR)C = B \cdot \log\_2(1 + SNR)C=B⋅log2(1+SNR)

**Where:**

* + CCC = Channel capacity (in bits per second).
  + BBB = Bandwidth of the channel (in Hz).
  + SNRSNRSNR = Signal-to-Noise Ratio.

Serialization ensures that the entire available BBB is used for a single data stream, achieving higher CCC compared to spreading data across multiple lines with lowerBBB**.**

1. **Synchronization Overhead Reduction** 
   * Parallel communication requires separate clocks or alignment mechanisms for each line, consuming bandwidth for clock signals.
   * Serialization embeds clock information within the data stream (e.g., via encoding), optimizing bandwidth for actual data transmission.
2. **Scalability with Higher Data Rates** 
   * As data rates increase, serialization scales effectively by using the full bandwidth of high-speed transmission technologies such as:
     + - **Fiber optics:** Wide bandwidth allows multi-gigabit serial streams.
       - **Copper cables:** Serialization optimizes limited bandwidth by avoiding parallel-induced distortions.
3. **Application in Multiplexing** 
   * **Time-Division Multiplexing (TDM):** 
     + - Serialization enables multiple data sources to share a single channel, optimizing bandwidth usage by interleaving bits or symbols from different sources.

Before serialization we need to encode the data and then we must convert the data to serial data string, We have many encoding techniques to encode data

**Different Encoding and decoding techniques for digital communications**

In this project the circuit is converting digital data from “parallel to serial” and “serial to parallel” so we must create a circuit which will help generated data to encode and decode

**Different type of coding techniques for digital communications**

There are different types of coding techniques in digital communications like   
source coding

**1. Source Coding**

**Goal:** To represent information with fewer bits while minimizing information loss.

**Examples:**

* **Huffman Coding:** Assigns shorter codes to more frequent symbols (e.g., text compression).
* **Arithmetic Coding:** More efficient than Huffman, represents data as a single number within an interval.
* **JPEG, MPEG:** Image and video compression standards using transform coding (DCT).

**2. Channel Coding**

**Goal:** To introduce redundancy to the data to enable error detection and correction during transmission.

**Examples:**

* **Hamming Codes:** Simple linear block codes for single-bit error correction.
* **Convolutional Codes:** More complex codes with good error correction capabilities, often used in deep space communication.
* **Reed-Solomon Codes:** Widely used for error correction in CDs, DVDs, and data storage systems.

**3. Line Coding**

**Goal:** To represent digital data as Analog signals for transmission over communication channels.

**Examples:**

* **Unipolar NRZ:** Simple encoding with one voltage level for '1' and no voltage for '0'.
* **Polar NRZ:** Uses two voltage levels (e.g., positive and negative) to represent '1' and '0'.
* **Manchester Coding:** Self-clocking encoding with a transition in the middle of each bit period.
* **8b/10b:** Maps 8-bit words to 10-bit symbols for DC balance and clock recovery.

**4. Modulation**

**Goal:** To transmit digital data by modulating a carrier signal (e.g., frequency, amplitude, phase).

**Examples:**

* **Amplitude Shift Keying (ASK):** Changes the amplitude of the carrier signal.
* **Frequency Shift Keying (FSK):** Changes the frequency of the carrier signal.
* **Phase Shift Keying (PSK):** Changes the phase of the carrier signal.
* **Quadrature Amplitude Modulation (QAM):** Combines amplitude and phase shifts for higher data rates.

**5. Spread Spectrum**

**Goal:** To spread the signal over a wider bandwidth to improve resistance to interference and jamming.

**Examples:**

* **Direct Sequence Spread Spectrum (DSSS):** Multiplies the data signal with a pseudo-random noise sequence.
* **Frequency Hopping Spread Spectrum (FHSS):** Transmits the signal over different frequency bands.

**Coding techniques for wired communication**

For **wired digital communication**, the most suitable type of encoding is **Line Encoding** because it ensures reliable data transmission over physical cables while minimizing errors, DC bias, and synchronization issues. The best line encoding scheme depends on the application and system requirements.

**Recommended Line Encoding Techniques for Wired Digital Communication:**

1. **Non-Return-to-Zero (NRZ) Encoding**
   * Used in **serial communication** (e.g., UART, USB).
   * Simple and efficient but may have synchronization issues.
2. **Manchester Encoding**
   * Used in **Ethernet (10BASE-T networks)**.
   * Ensures synchronization but requires more bandwidth.
3. **Differential Manchester Encoding**
   * Used in **Token Ring networks** and **MIL-STD-1553 (military bus systems)**.
   * More resistant to noise and phase shifts than standard Manchester encoding.
4. **4B/5B Encoding**
   * Used in **Fast Ethernet (100BASE-TX)**.
   * Increases efficiency while maintaining synchronization.
5. **8b/10b Encoding**
   * Used in **PCI Express, SATA, and USB 3.0**.
   * Ensures DC balance and error detection.
6. **Scrambling Techniques (e.g., 64b/66b Encoding)**
   * Used in **Gigabit Ethernet, optical fiber communication**.
   * Helps reduce long sequences of zeros and ones to prevent clock recovery issues.

For normal communication protocol-based implementation line coding we can implement scrambling techniques else for study purpose we can implement 8B/10B encoding and decoding techniques

**Best Encoding Techniques for Serial Data Transmission:**

1. **NRZ (Non-Return-to-Zero)**
   * **Best for:** Simple designs (UART, SPI, RS-232, RS-485)
   * **Pros:**
     + Easy to implement in hardware (just shift registers).
     + No extra overhead.
   * **Cons:**
     + No inherent clock synchronization.
2. **NRZI (Non-Return-to-Zero Inverted)**
   * **Best for:** USB, Fiber Channel, SSD interfaces
   * **Pros:**
     + Avoids long sequences of 0s or 1s.
     + Reduces DC bias.
   * **Cons:**
     + Needs additional scrambling for synchronization.
3. **Manchester Encoding**
   * **Best for:** Ethernet (10BASE-T), RFID, Infrared communication.
   * **Pros:**
     + Self-clocking (no need for a separate clock line).
     + Reliable for long-distance communication.
   * **Cons:**
     + Requires **twice the bandwidth**.
4. **8b/10b Encoding**
   * **Best for:** PCI Express, SATA, High-speed serial links.
   * **Pros:**
     + Ensures DC balance and error detection.
     + Used in **high-speed** applications.
   * **Cons:**
     + Requires a **20% overhead**.
     + More complex decoder.

The **overhead** in **8b/10b encoding** refers to the extra bits added to the transmitted data for better signal integrity, synchronization, and error detection.

**1. What Causes Overhead?**

* **8b/10b encoding** converts each **8-bit data word** into a **10-bit code word**.
* This results in a **20% overhead** because: 10 bits−8 bits8 bits×100%=25%\frac{10 \text{ bits} - 8 \text{ bits}}{8 \text{ bits}} \times 100\% = 25\%8 bits10 bits−8 bits​×100%=25% However, since the transmission rate is higher, the effective overhead is around **20%** in practical systems.

**2. Why is this Overhead Necessary?**

* **DC Balance**: Prevents long sequences of 1s or 0s to maintain signal integrity.
* **Clock Synchronization**: Ensures transitions in the signal for easier clock recovery.
* **Error Detection**: Some invalid 10-bit sequences help detect transmission errors.

**3. Impact of Overhead**

* **Bandwidth Consumption**: Since each **8-bit data byte** is transmitted as **10 bits**, the actual transmission rate is **higher** than the raw data rate.
* **Complex Decoding**: The receiver must decode the **10-bit symbols** back into **8-bit data**, increasing hardware complexity.
* With this information we can conclude that we can use 8B/10B scrambling method to encode and decode the data bits before serialization of data because with this method we can encode and decode data parallelly

**8b/10b Encoding and Decoding**

**8b/10b encoding** is a line code used in telecommunications and data storage to improve the reliability and performance of high-speed serial data transmission. It maps 8-bit bytes to 10-bit symbols, ensuring several key properties:

* **DC balance:** The long-term average of the transmitted signal is zero, which is important for some transmission media.
* **Bounded disparity:** The difference between the number of ones and zeros in a sequence is limited, preventing long strings of identical bits.
* **Sufficient transitions:** The encoded signal has enough transitions to allow for reliable clock recovery at the receiver.

**Encoding Process:**

1. **Input:** An 8-bit byte is received.
2. **Disparity calculation:** The encoder calculates the disparity of the byte, which is the difference between the number of ones and zeros.
3. **Code selection:** Based on the disparity, the encoder selects one of two possible 10-bit codewords from a lookup table. The selected codeword has a disparity that complements the input byte's disparity, maintaining DC balance.
4. **Output:** The 10-bit codeword is transmitted.

**Decoding Process:**

1. **Input:** A 10-bit symbol is received.
2. **Code lookup:** The decoder uses the received codeword to look up the corresponding 8-bit byte in the lookup table.
3. **Disparity update:** The decoder updates the running disparity based on the received codeword.
4. **Output:** The decoded 8-bit byte is delivered.

**Advantages of 8b/10b Encoding:**

* **Improved signal quality:** DC balance and bounded disparity reduce signal distortion and improve clock recovery.
* **Error detection:** Some codewords are reserved for special purposes, such as control signals or error detection.
* **Flexibility:** The encoding scheme can be adapted to different transmission rates and media.

**Applications of 8b/10b Encoding:**

* **Fibre Channel:** Widely used in high-speed storage networks.
* **Gigabit Ethernet:** Employed in some Gigabit Ethernet implementations.
* **Serial ATA:** Used in high-speed hard disk drives.

**A diagram of a computer program

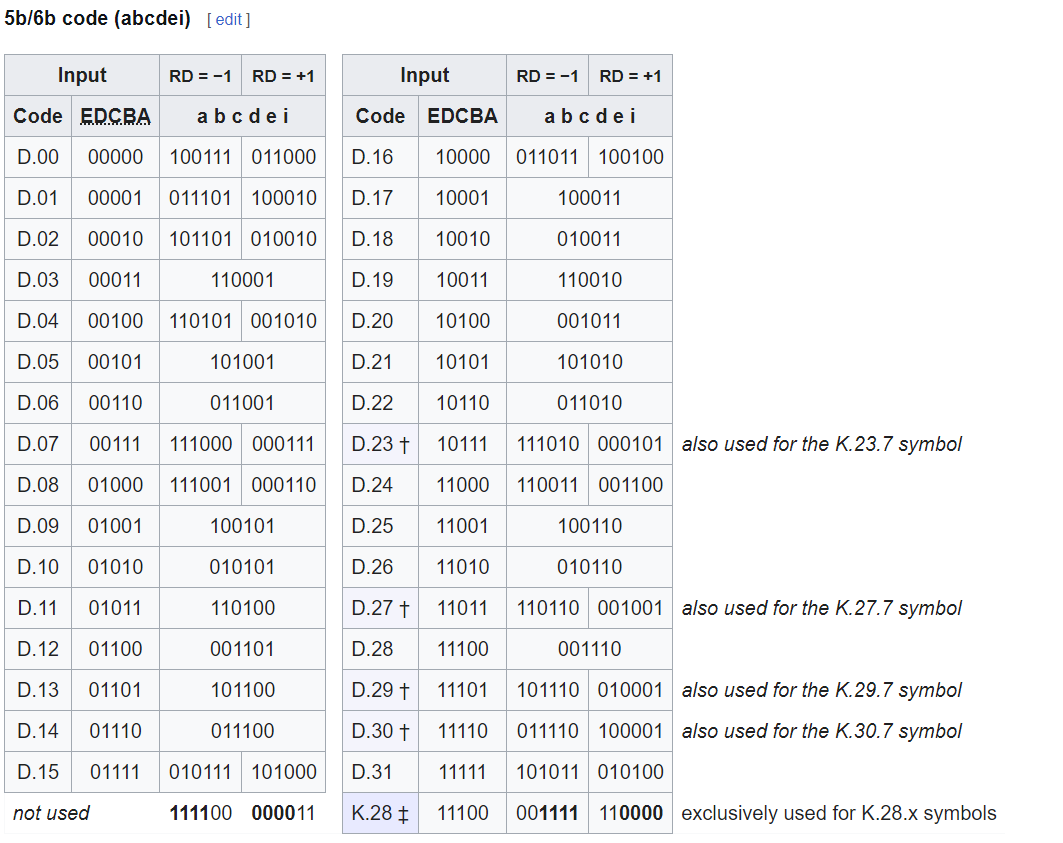
Description automatically generated**

**At many parameters 8b/10b encoding is more suitable for line coding**

1. **DC Balance:**
   * Ensures an equal number of 1s and 0s over time, eliminating the DC component of the signal.
   * This is crucial for AC-coupled channels or those with transformers, as DC components can cause distortion or saturation.
2. **Bounded Disparity:**
   * Limits the maximum number of consecutive 1s or 0s.
   * This prevents long periods of constant voltage, which can lead to signal degradation and difficulties in clock recovery.
3. **Clock Recovery:**
   * Provides frequent transitions in the signal, aiding the receiver in accurately recovering the clock signal.
   * This is essential for maintaining synchronization between the transmitter and receiver.
4. **Error Detection:**
   * Some 10-bit codewords are reserved for special purposes (e.g., control signals, error detection).
   * This allows the receiver to detect certain types of errors during transmission.

**Comparison with Other Line Codes:**

* **Unipolar NRZ:** Lacks DC balance and can have poor clock recovery.
* **Polar NRZ:** May have a DC component and limited clocking information.
* **Manchester Coding:** Provides good clocking but has a higher bandwidth requirement due to the transitions in the middle of each bit.



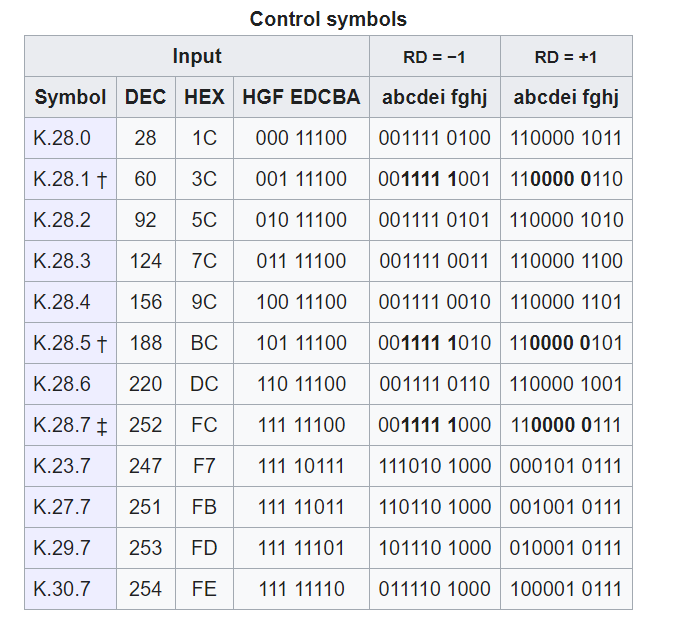
A screenshot of a computer code

Description automatically generated

**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.

Resulting from the 5b/6b and 3b/4b tables the following 12 control symbols are allowed to be sent:



* By simply encoding and converting the data to serial bit string we can transmit the data over a metal wire line

**Verification testing of encoder:**

**A screenshot of a computer

Description automatically generated**

**By this we can design we can design the transmitter circuit**

**Creation of transmission block**

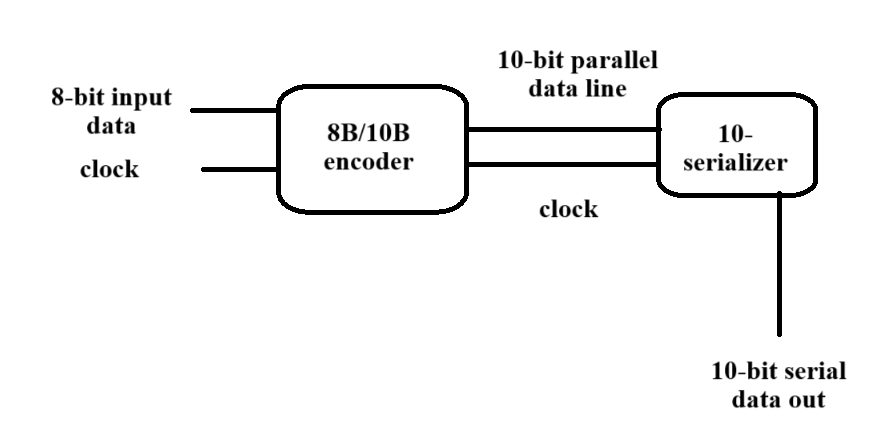
**Inputs:**

* **clk**: Clock signal
* **din[7:0]**: 8-bit parallel data input
* **en**: Enable signal
* **kin**: Control signal
* **rst**: Reset signal

**Blocks:**

1. **8b10b Encoder (encoder\_8b10)**:
   * Inputs: clk, din[7:0], en, kin, rst
   * Output: dout[9:0] (10-bit encoded data)
2. **Serializer**:
   * Inputs: clk, dout[9:0] (from encoder), rst
   * Control Input: load
   * Output: serial\_out

**Output:** **serial\_out**: Serialized data output



Block diagram for transmission block

**Implementation of this block and its RTL design:A diagram of a computer network

Description automatically generated**

**Testbench verification of transmission block:**

**A screenshot of a computer

Description automatically generatedWith this we can conclude the design and verification of digital transmission block**

**Design and implementation of receiver block**

At the time of receiving the data we need to recover the clock from the received data because when we have serialized the data the bit rate is also incised by the number of bits, so we need to retrieve the clock and, we need to maintain the data frame same by not losing the data

For this purpose, we need to design a circuit which will recover bits and if any bit has lost its pulse peak it needs to amplify and retrieve the data and clock frequency from received data we can use this clock data recovery module

**Theory Summary for the Clock and Data Recovery (CDR) Circuit:**

The **Clock and Data Recovery (CDR)** circuit is designed to recover the clock signal and correctly sample incoming serial data in high-speed communication systems. It ensures the synchronization of the data stream with the recovered clock, making the system capable of extracting valid data from the serial signal. The CDR process is essential in systems like serial communication, where the transmitted clock is often not available for recovery at the receiver.

**Key Functions of the CDR Circuit:**

1. **Upsampling the Serial Data:**
   * The CDR circuit uses multiple phases of a clock signal (typically 4-phase in this design) to unsampled the incoming serial data.
   * Each clock phase samples the data at different times, creating a set of 10-bit wide unsampled signals from a single-bit serial input (a\_rx).
   * This up sampling x`ensures that data transitions can be detected more accurately since the data is being sampled at multiple clock edges.
2. **Phase Selection:**
   * The CDR circuit needs to determine the **best phase** of the clock at which to recover the data. The best phase corresponds to the clock phase where the signal transitions occur with the most reliable timing (i.e., where the "eye" of the signal is widest).
   * In this simplified implementation, the phase selection logic is fixed to 2'b01, corresponding to the second phase. However, in a more sophisticated design, this logic would evaluate the transition points in the upsampled data and dynamically select the phase that aligns best with the incoming data.
3. **Recovered Clock Output:**
   * Once the best clock phase is selected, the recovered clock (bit\_clock) is output, aligned with the chosen clock phase.
   * The recovered clock can then be used for proper data synchronization in subsequent parts of the system.
4. **Data Sampling:**
   * After selecting the best phase, the CDR circuit samples the upsampled data at the chosen clock phase.
   * The sampled data (samp\_test) represents the 10 bits that are synchronized with the recovered clock, making it valid for further processing.

**Mathematical and Logical Flow:**

* **Upsampling Logic:**

c\_rx\_upsampled[i]=shift(a\_rx,9)c\\_rx\\_upsampled[i] = \text{shift}(a\\_rx, 9)c\_rx\_upsampled[i]=shift(a\_rx,9)

This takes the serial input data and shifts it into a 10-bit wide signal at each clock phase.

* **Phase Selection:** The best phase is determined by detecting transitions in the upsampled data, typically done by measuring signal quality or using a transition detection algorithm. The result is stored in best\_samp.
* **Recovered Clock:**

bit\_clock=clk[best\_samp]\text{bit\\_clock} = \text{clk}[best\\_samp]bit\_clock=clk[best\_samp]

This output clock is the phase that aligns best with the data transitions.

* **Data Sampling:**

samp\_test=c\_rx\_upsampled[best\_samp]\text{samp\\_test} = c\\_rx\\_upsampled[best\\_samp]samp\_test=c\_rx\_upsampled[best\_samp]

The data is sampled at the selected phase and output as samp\_test.

**Applications of the CDR Circuit:**

* **Serial Communication Systems:** The CDR module is widely used in high-speed serial communication systems like Ethernet, PCIe, or USB, where data is transmitted serially, and the clock is embedded or needs to be recovered at the receiver.
* **High-Speed Data Transmission:** This circuit is vital for ensuring the data integrity and timing recovery in systems that operate at high speeds, where clock and data are transmitted together but need to be recovered separately at the receiver.

**Challenges in CDR Design:**

1. **Phase Detection:** In real-world applications, determining the best phase dynamically is complex and requires algorithms for transition detection and phase error estimation.
2. **Clock Skew:** Variations in the timing between the transmitter and receiver clocks can lead to errors in phase alignment, which the CDR must correct.
3. **Signal Integrity:** High-frequency signals are prone to noise and jitter, making the accurate recovery of the clock and data more challenging

**Basic information about circuit:**

In serial communication systems, data and clock signals are often transmitted together, but the receiver does not have a separate clock. The CDR module extracts the clock information from the data stream and aligns the sampling to ensure the correct interpretation of transmitted bits. Without CDR, bit errors due to misaligned sampling would increase, reducing the system's reliability.

**1. Clock and Data Recovery (CDR) Module**

**Purpose**:  
The CDR module is designed to recover timing and data from a serialized data stream by sampling it at the optimal phase of the clock. It ensures the correct reconstruction of the transmitted data.

**Key Functions**:

1. **Bitwise Operations**:
   * Performs XOR (^), AND (&), and OR (|) operations on the input bits to detect patterns and relationships.
2. **Result Computation**:
   * res\_A[0]: Represents the parity (XOR of all bits) of the input. Indicates if the number of 1s is odd or even.
   * res\_A[1]: Combines several terms derived from specific combinations of the input bits. Detects more complex patterns in the input.
   * res\_A[2]: Computes the logical AND of all input bits. It’s 1 only if all bits are 1.

**Applications**:

* Bitwise operations are useful in detecting signal patterns, error correction, or counting the number of 1s in a signal.
* Helps the CDR module or other components in making decisions based on signal characteristics.

**Basic Theoretical Concepts Involved**

1. **Phase Sampling**:  
   The input signal is sampled at multiple clock phases to find the best point for accurate data recovery.
2. **Transition Detection**:  
   Identifying edges (0 to 1 or 1 to 0 transitions) helps align the sampling clock to the centre of data bits, ensuring reliable data recovery.
3. **Bitwise Operations**:  
   XOR, AND, and OR operations are fundamental in digital circuits to detect patterns, compute parity, and validate signal integrity.
4. **Sequential Logic**:  
   The use of registers (\_reg signals) ensures synchronization and retains past states for comparison and decision-making.

**Implementation of CDR**

**1. Inputs and Outputs:**

**Inputs:**

* rst (Reset): Resets the circuit to an initial state.
* clks\_in [3:0] (multi-phase clock input): A 4-phase clock used for oversampling the input serial data.
* a\_rx (Serial data input): The incoming serial data stream.

**Outputs:**

* bit\_clock: The recovered clock signal, aligned to the best phase of the input clock.
* samp\_test [9:0]: The sampled data output (10 bits), based on the selected best clock phase.

**2. Functional Blocks:**

**(A) Oversampling the Input Data:**

* The circuit uses **four different phases of a clock** (clks\_in[3:0]) to sample the incoming serial data (a\_rx).
* Each phase stores **10 bits** of upsampled data (c\_rx\_upsampled[3:0]).
* The circuit ensures that the input data is sampled at different clock edges to increase accuracy.

**(B) Phase Selection Logic:**

* The circuit determines the best sampling phase (best\_samp), which helps in correctly extracting the data.
* Currently, the logic **always selects 2'b01 as the best phase** (a placeholder logic).
* In a real implementation, this block would analyze transitions in the sampled data and select the phase with the best eye opening.

**(C) Recovered Clock and Data Extraction:**

* The **bit clock (bit\_clock)** is set based on the selected clock phase.
* The **sampled data (samp\_test)** is assigned from the upsampled data corresponding to the selected clock.

**3. Issues and Improvements:**

**Issues in the Current Design:**

1. **Incorrect Best Phase Selection:**
   * The best phase (best\_samp) is hardcoded (2'b01), which is not dynamic.
   * It should analyze transitions in the upsampled data to select the optimal phase dynamically.
2. **Incorrect Use of clks\_in[best\_samp] in Always Block:**
   * The syntax always @(posedge clks\_in[best\_samp] ...) is **not synthesizable** because best\_samp is a variable.
   * Instead, a **multiplexer-based clock selection** should be used.
3. **Incomplete Upsampling Logic:**
   * The upsampling logic only assigns {9'b0, a\_rx} to c\_rx\_upsampled[x] (i.e., only last bit is used).
   * Ideally, **a shift register should be used** to store past values for full 10-bit capture.

**Mathematical expression for this circuit**

**1. Inputs and Outputs:**

**Inputs:**

* rst (Reset): Resets the circuit to an initial state.
* clks\_in [3:0] (Multi-phase clock input): A 4-phase clock used for oversampling the input serial data.
* a\_rx (Serial data input): The incoming serial data stream.

**Outputs:**

* bit\_clock: The recovered clock signal, aligned to the best phase of the input clock.
* samp\_test [9:0]: The sampled data output (10 bits), based on the selected best clock phase.

**2. Functional Blocks:**

**(A) Oversampling the Input Data:**

* The circuit uses **four different phases of a clock** (clks\_in[3:0]) to sample the incoming serial data (a\_rx).
* Each phase stores **10 bits** of upsampled data (c\_rx\_upsampled[3:0]).
* The circuit ensures that the input data is sampled at different clock edges to increase accuracy.

**(B) Phase Selection Logic:**

* The circuit determines the best sampling phase (best\_samp), which helps in correctly extracting the data.
* Currently, the logic **always selects 2'b01 as the best phase** (a placeholder logic).
* In a real implementation, this block would analyze transitions in the sampled data and select the phase with the best eye opening.

**(C) Recovered Clock and Data Extraction:**

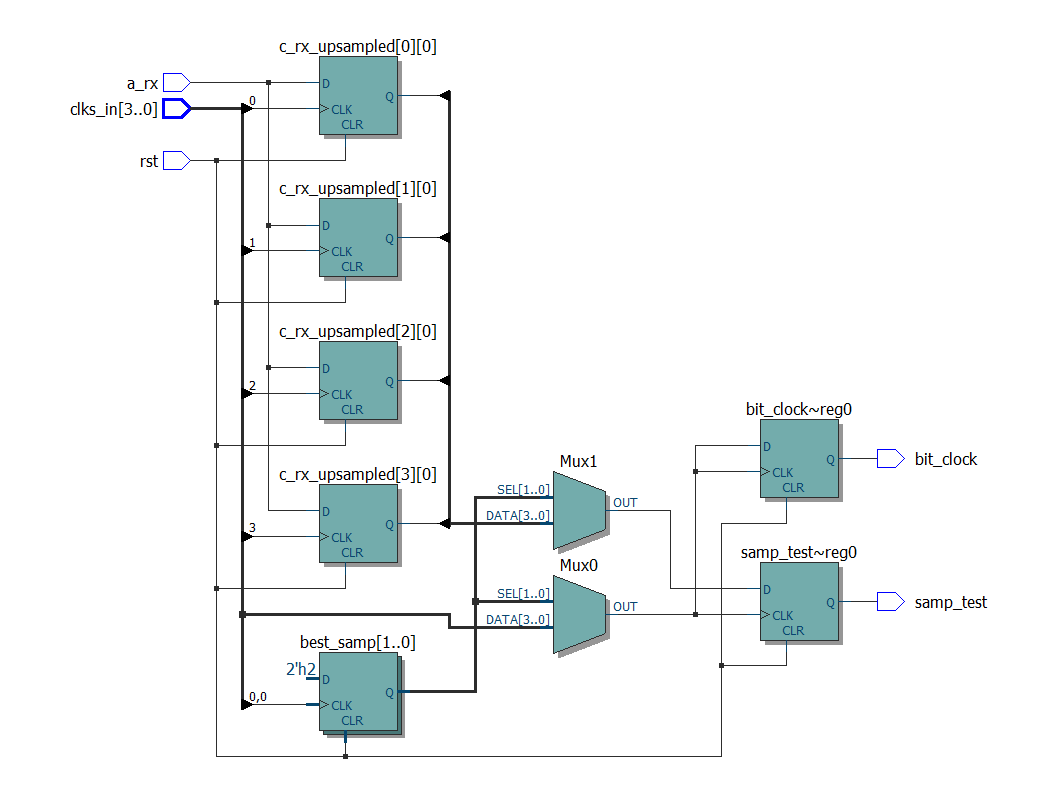
* The **bit clock (bit\_clock)** is set based on the selected clock phase.
* The **sampled data (samp\_test)** is assigned from the upsampled data corresponding to the selected clock.

**3. Issues and Improvements:**

**Issues in the Current Design:**

1. **Incorrect Best Phase Selection:**
   * The best phase (best\_samp) is hardcoded (2'b01), which is not dynamic.
   * It should analyze transitions in the upsampled data to select the optimal phase dynamically.
2. **Incorrect Use of clks\_in[best\_samp] in Always Block:**
   * The syntax always @(posedge clks\_in[best\_samp] ...) is **not synthesizable** because best\_samp is a variable.
   * Instead, a **multiplexer-based clock selection** should be used.
3. **Incomplete Upsampling Logic:**
   * The upsampling logic only assigns {9'b0, a\_rx} to c\_rx\_upsampled[x] (i.e., only last bit is used).
   * Ideally, **a shift register should be used** to store past values for full 10-bit capture.

RTL implementation



**After this recovery we must deserialize that 10-bit data**

**De-serializer**

**Introduction to de-serializer:**

A de-serializer is a hardware or software component used to convert serialized data (data transmitted in a single stream, one bit at a time) back into its parallel form (multiple bits transmitted simultaneously). It is commonly used in communication systems and digital electronics to receive and process data streams.

**Key Concepts:**

1. **Serialization**: In many communication systems, data is serialized to reduce the number of wires or channels needed for transmission, improving efficiency.
2. **De-serialization**: At the receiving end, the serialized data must be converted back to its original parallel format for further processing.

**Applications:**

* **High-Speed Data Communication**: Used in protocols like PCIe, USB, Ethernet, and HDMI, where data is serialized for transmission.
* **Clock Recovery**: De-serializers often integrate mechanisms to recover the clock signal from the data stream (e.g., using a Clock Data Recovery (CDR) circuit).
* **Interfacing with Memory and Processors**: De-serializers are used in systems interfacing with parallel memory or processors after receiving serialized data.

**Example in Hardware:**

* **SPI (Serial Peripheral Interface)**: In SPI, the master transmits serialized data to the slave. The slave uses a de-serializer to convert the received serial stream back into parallel data for processing.

**Block Diagram:**

For a **N-bit de-serializer**, the process involves:

1. **Input Shift Register**: Bits are shifted in serially on each clock pulse.
2. **Parallel Output Register**: After N bits are received, the parallel register outputs the complete data word.

**Importance of de-serializer in digital communication:**

The de-serializer plays a crucial role in digital communication systems by enabling the efficient transmission and reception of high-speed data. Its importance stems from the following key factors

**1. Efficient Data Transmission and Reception**

* **Serial Transmission:** High-speed serial links are used to transmit data over long distances or high-speed channels using fewer wires or traces.
* **De-serialization:** Converts the received serial data back to parallel form for processing, enabling efficient handling by digital systems like microprocessors, FPGAs, or ASICs.

**2. Bandwidth Utilization**

* **High-Speed Links:** Serial data transmission maximizes bandwidth over a single channel or medium.
* **Parallel Data Recovery:** De-serialization ensures that the system can process data efficiently by spreading the received serial stream across multiple parallel lines.

**3. Clock and Data Synchronization**

* De-serializers often include mechanisms for clock data recovery (CDR) to align the data bits with the system clock, ensuring accurate reconstruction of parallel data.

**4. Reduced Hardware Complexity in Transmission**

* By allowing serial transmission, the serializer and de-serializer (SerDes) pair reduces the number of physical interconnects required between components.
* At the receiver, the de-serializer expands this efficient single-channel transmission back into parallel data without needing complex receiver designs.

**5. Support for High-Speed Protocols**

* Essential in implementing digital communication protocols like PCIe, Ethernet, SATA, USB, and more, which use serial transmission for high-speed data transfer.
* The de-serializer ensures compatibility and proper data recovery in these systems.

**6. Error Detection and Correction**

* Many de-serializers integrate features like data alignment, frame detection, and error checking (e.g., CRC) to ensure robust communication in noisy environments.

**7. Applications Across Diverse Systems**

* **Networking:** Used in Ethernet and optical communication systems.
* **Storage:** Converts serialized data from SATA or NVMe interfaces back to parallel for storage controllers.
* **Video Systems:** Handles high-speed video transmission, such as HDMI or DisplayPort, by de-serializing serialized streams.
* **Telecommunication:** Used in baseband units, receivers, and transceivers for reconstructing data.

**8. Power and Area Efficiency**

* A de-serializer enables compact, high-speed serial communication systems, reducing the need for bulky parallel buses, thus saving power and space.

**9. Scalability for Higher Data Rates**

* As communication systems scale to higher data rates, de-serializers support increasing bit rates by leveraging advanced techniques like multi-lane parallelization.

**what happens with the bit rate od data when we de-serializer it after receiving?**

When data is de-serialized after reception, the **bit rate remains unchanged**, but the **data rate per channel decreases**, as the data is spread across multiple parallel lines. Here's a detailed explanation:

**Bit Rate**

* **Definition**: The number of bits transmitted per second (bps) over the serial link.
* **Impact**: The bit rate of the serial data is preserved because the de-serializer does not alter the total number of bits transmitted per second.

**Data Rate per Parallel Line**

* **Definition**: The rate at which data is available on each parallel output line.
* **Impact**: De-serialization divides the incoming serial bit stream across multiple parallel data lines. As a result:
  + If the serial data stream operates at RbR\_bRb​ bits per second (bps) and the de-serializer outputs NNN parallel lines, the data rate on each line is reduced to Rp=RbNR\_p = \frac{R\_b}{N}Rp​=NRb​​.
  + This spreading reduces the frequency requirements on the receiving end for handling each line.

**Example:**

* **Input**: A serial link operates at 10 Gbps.
* **Output**: After de-serialization into 10 parallel lines:
  + Bit rate remains 10 Gbps10 \, \text{Gbps}10Gbps for the total system.
  + Data rate per parallel line becomes 10 Gbps10=1 Gbps per line\frac {10 \, \text{Gbps}}{10} = 1 \, \text{Gbps per line}1010Gbps​=1Gbps per line.

**Main Components and Their Functionality**

**1. Deserializer Module**

The **Deserializer** converts incoming serial data into parallel data using a 10-bit shift register, manages clock synchronization, and detects valid data frames.

* **Inputs**:
  + rst: Resets the system.
  + clks\_in: A 4-phase clock input for clock and data recovery.
  + a\_rx: The serial data input.
  + disparity\_d: Input signal for controlling running disparity.
* **Outputs**:
  + c\_parallel\_out: 10-bit parallel data output.
  + clk\_out: Recovered clock signal for parallel data.
  + disparity\_q: Running disparity indicator.
  + c\_data\_valid: Indicates when valid parallel data is available.

**Key Functionalities:**

1. **Shift Register**:
   * Serial data from sampled\_data is shifted into a 10-bit register (shift\_reg).
   * Previous state of the shift register is stored in shift\_reg\_prev.
2. **Comma Detection**:
   * Detects special patterns (commas) like 10'b0011111000 or 10'b1100000111 in the shift register to identify frame boundaries.
3. **Data Valid and Clock Generation**:
   * A counter (cycle\_count) determines when valid parallel data is available and generates clk\_out.
   * c\_data\_valid is asserted when valid parallel data is ready to be output.
4. **Parallel Data Output**:
   * When c\_data\_valid is high, the deserialized 10-bit data is output via c\_parallel\_out.
5. **Running Disparity Management**:
   * Tracks the disparity signal (disparity\_q) to ensure DC balance in the transmission.

**2. Clock and Data Recovery (CDR) Module**

The CDR module recovers the clock and samples the serial input data (a\_rx) to ensure proper timing.

* **Inputs**:
  + rst: Resets the CDR module.
  + clks\_in: A 4-phase clock input for sampling.
  + a\_rx: Serial data input.
* **Output**:
  + sampled\_data: Data sampled at the best clock phase.

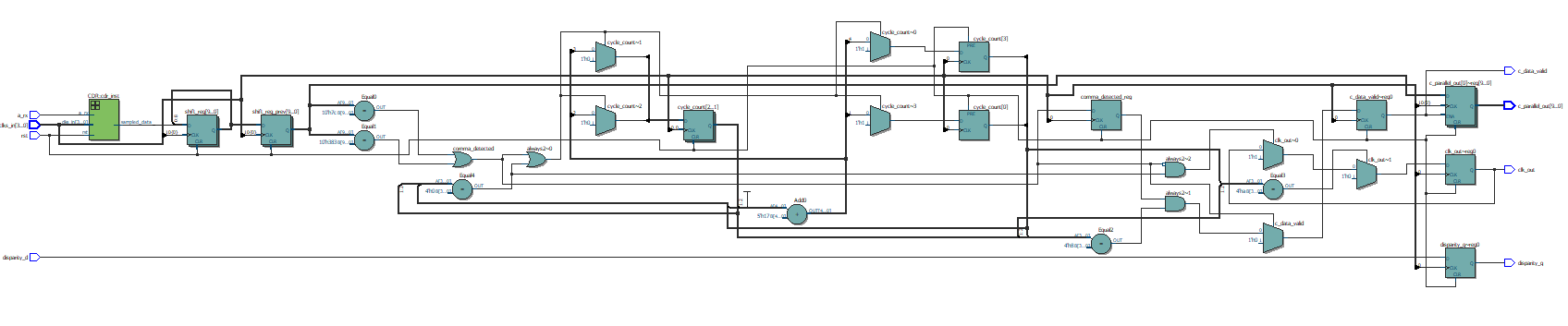
**Key Functionalities:**

1. **Upsampling**:
   * Uses 4-phase clocks (clks\_in) to sample the input data at different phases, capturing transitional points.
2. **Best Sampling Point Selection**:
   * Compares sampled signals to identify the optimal phase for sampling the data.
   * This ensures data is sampled at the most stable point in its transition.
3. **Sampled Data Output**:
   * Outputs the sampled data (sampled\_data) based on the selected best phase.

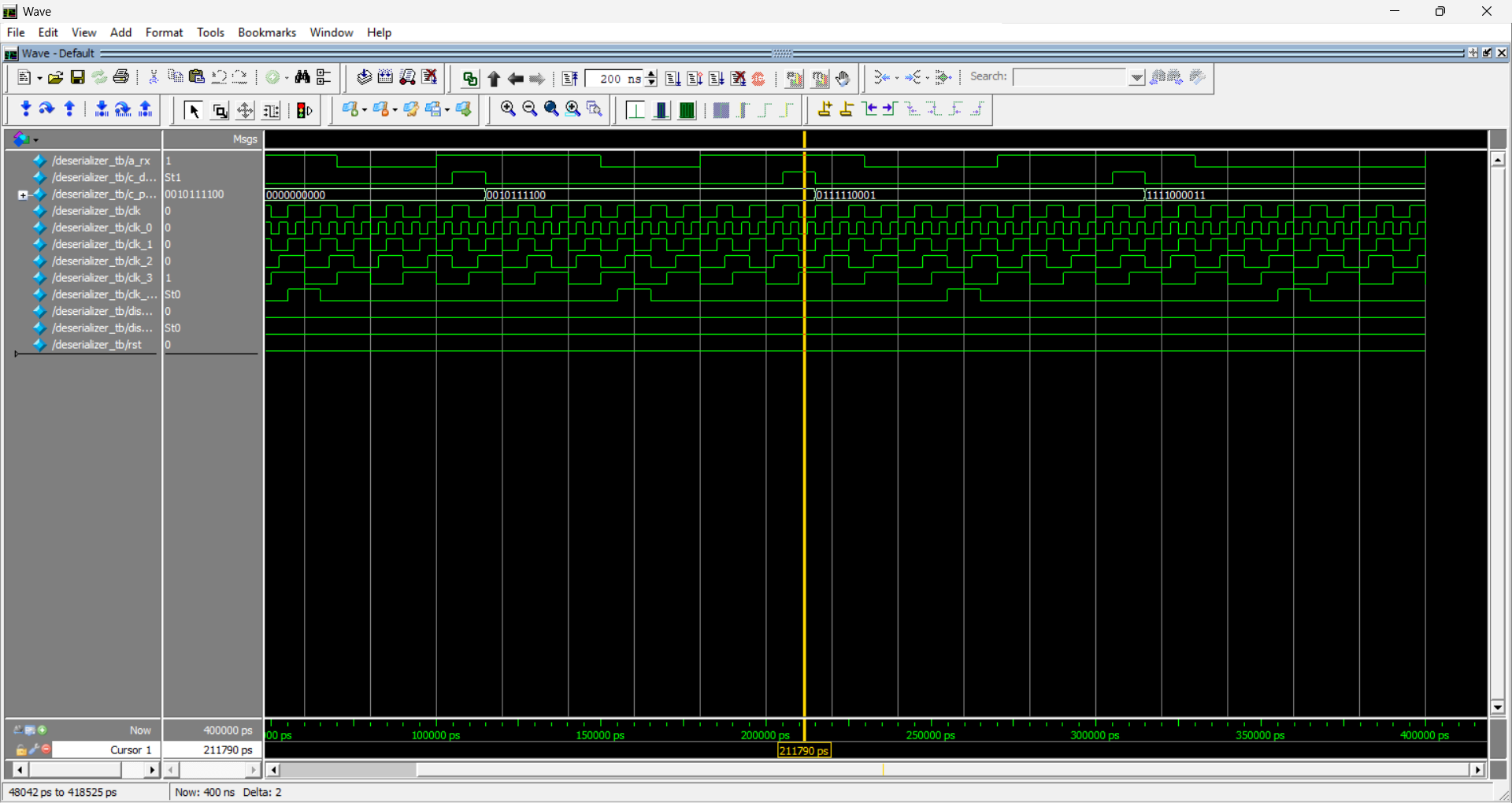
**How It Works Together**

1. The serial data is input into the CDR module, which aligns it with the appropriate clock phase and outputs sampled data.
2. The **Deserializer** shifts the sampled data into a 10-bit shift register, detecting frame boundaries using comma detection.
3. Once valid data is detected, it is output as parallel data (c\_parallel\_out), with a valid signal (c\_data\_valid) and a clock signal (clk\_out).
4. Running disparity is managed to ensure signal integrity in high-speed communication.

RTL-implementation:

****

Testbench implimantation:

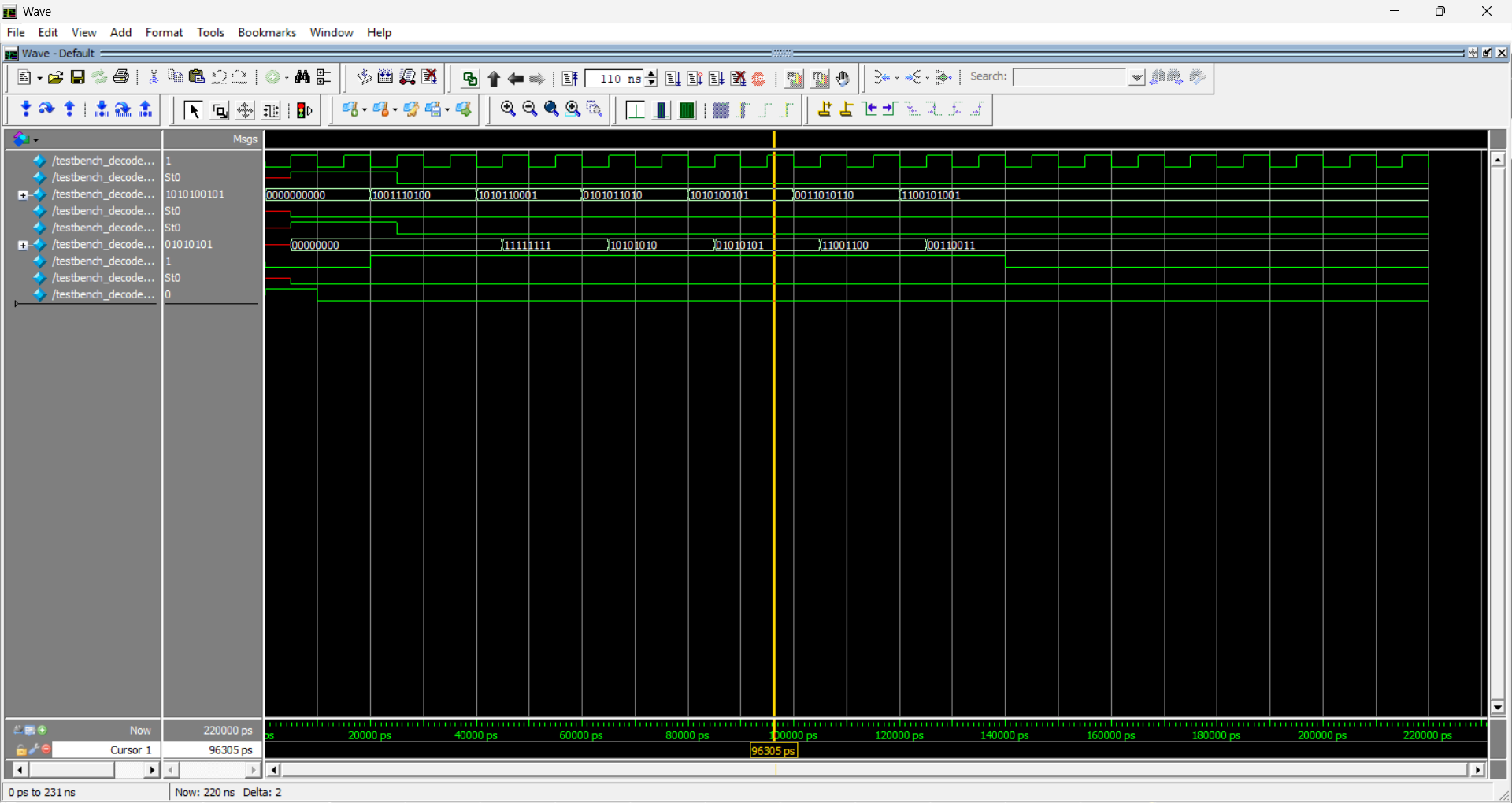


This architecture is common in protocols like PCIe, Ethernet, or USB, where data is transmitted serially and must be deserialized at the receiver end.

**After deserialization we must decode the data by 8B/10B decoder**

* Advantage of using 8B/10B decoding is that we don’t have to separately design the logic we can do revers coding of the 8B/10B encoding and we can regenerate the original signals that we have encoded
* We can also add one mor feature is that in 8B/10B encoding we can add disparity and error detection in this 8b/10B scrambler coding

**Verification testing of decoder:**

****

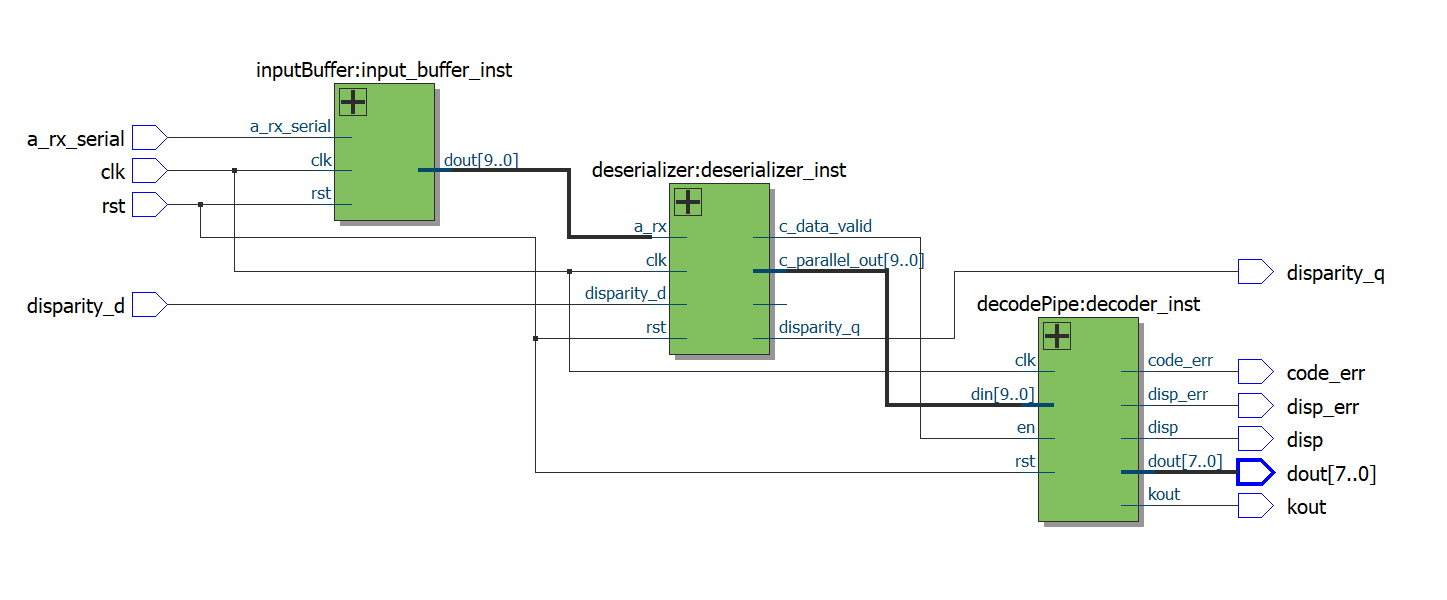
**Creation of receiver circuit:**

**Block Diagram Description:**

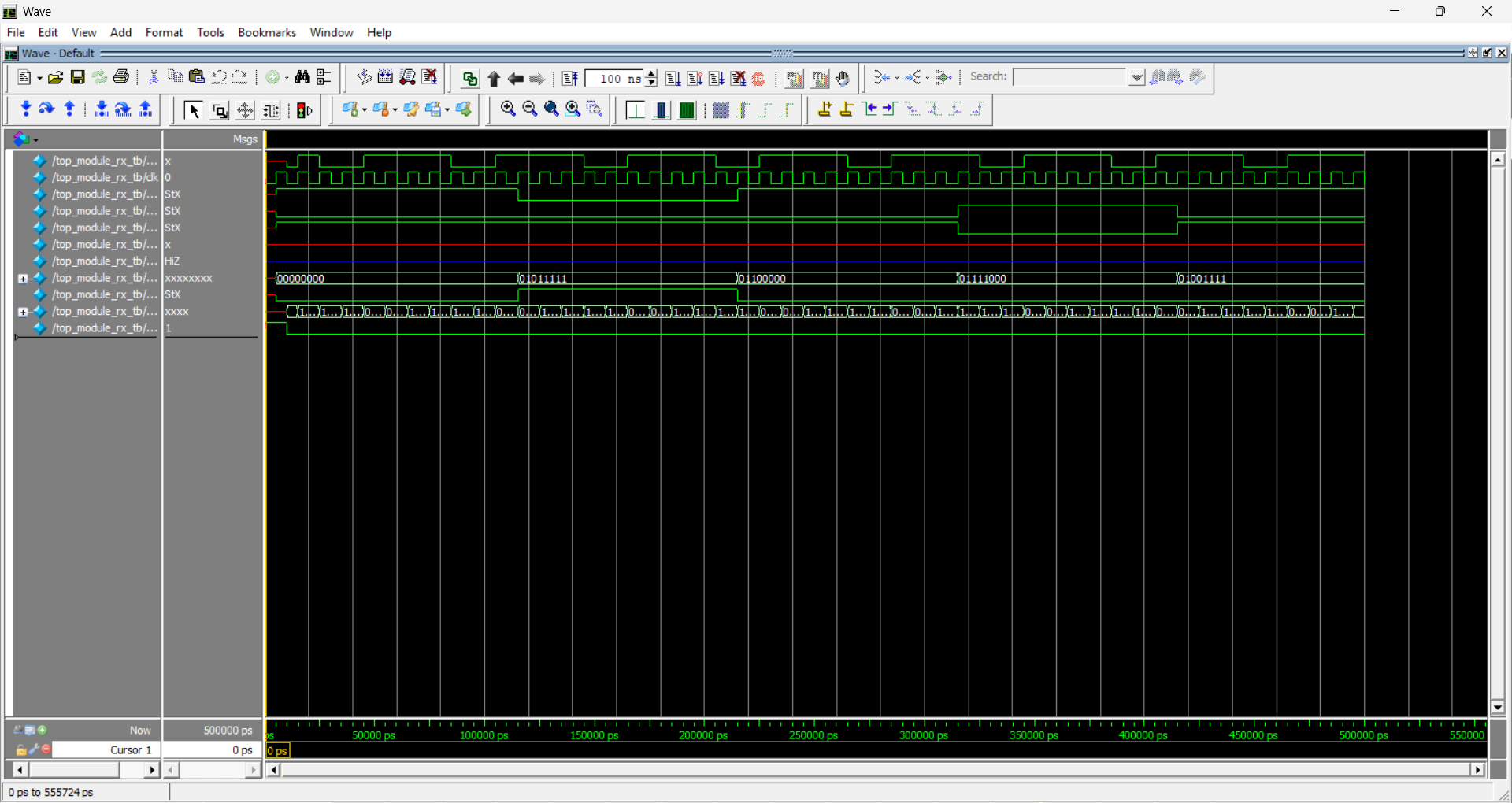
The diagram illustrates the process of receiving, deserializing, and decoding serial data. Let's break down each block and its function:

1. **Input Buffer:**
   * **Function:** This block acts as the initial storage for the incoming serial data stream. It temporarily holds the data bits as they arrive.
   * **Inputs:**
     + **Received Serial Data:** The stream of data bits coming in sequentially.
     + **Clock:** A timing signal that synchronizes the input buffer with the incoming data rate, ensuring proper sampling and storage of the serial data.
   * **Outputs:**
     + **Data:** The buffered serial data, now ready to be processed further.
2. **CDR (Clock Data Recovery):**
   * **Function:** The CDR block is crucial for extracting the clock signal from the incoming serial data. This is essential because the clock might not be explicitly transmitted along with the data.
   * **Inputs:**
     + **Data:** The buffered serial data from the input buffer.
   * **Outputs:**
     + **Data:** The serial data, potentiallyRetimed or cleaned up.
     + **Clock:** The recovered clock signal, now synchronized with the data stream.
3. **Deserializer:**
   * **Function:** This is the core component that converts the serial data stream into a parallel data format. It takes the single stream of bits and arranges them into groups (bytes or words) for simultaneous processing.
   * **Inputs:**
     + **Data:** The serial data from the CDR block.
     + **Clock:** The recovered clock signal from the CDR block, used to time the deserialization process.
   * **Outputs:**
     + **Deserialized Data:** The parallel data output, where each line represents a bit of the byte or word.
4. **Decoder:**
   * **Function:** The decoder block interprets the deserialized data based on the encoding scheme used. It translates the raw data into a meaningful format (e.g., ASCII characters, instructions, etc.).
   * **Inputs:**
     + **Deserialized Data:** The parallel data from the deserializer.
   * **Outputs:**
     + **Output Data:** The final decoded data, ready for use by the receiving system.

**RTL design of receiver circuit:**

****

**Verification and testing of receiver circuit:**

****

**Observation of varies parameters of netlist file and understanding the different usage of library files**

* In VLSI/ASIC design after finishing the design prosses we can decide which technologies do we have to use
* for manufacturing of our designed circuit, we must have to get the PDK (physical design kit) file from varies foundry like intel corporation, TSMC, Samsung electronics
* The PDK have the technology to fabricate the transistor devices MOSFET or FinFET according to PDK file the device is selected
* After the design and verification of our designed circuit we must need to perform the synthesis of circuit with reference to the PDK file
* While synthesising our circuit we must specify the clock port and clock frequency
* During the synthesis we can observe the area , speed and power which is used by our circuit

During the synthesis we need to define our library