EN – 2111 Electronic Circuit Design

UART Implementation in FPGA

Group Members

Navindu De Silva – 210097M
Kumuthu De Zoysa – 210108C
Dilshan N.L. – 210129P

Introduction

The report will discuss the implementation of a receiver and transmitter for transmitting one byte using UART between two FPGAs. It will also address how to synchronize the baud rates of the transmitter (Tx) and receiver (Rx). Additionally, an example testbench will be included to demonstrate the practical application of these concepts.

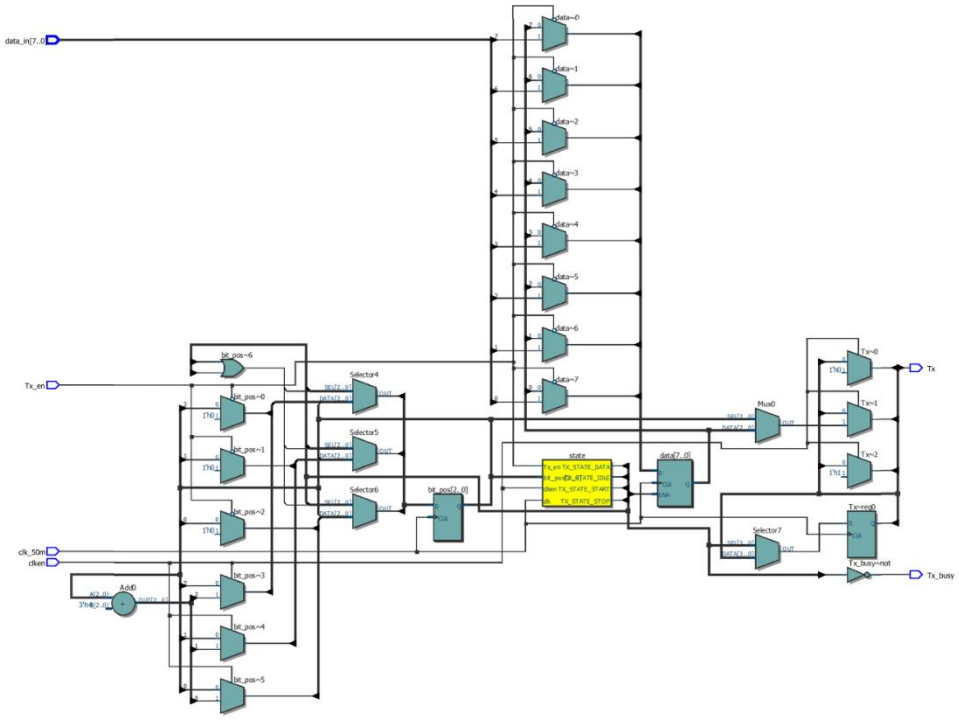
Transmitter Implementation

Verilog code:

```
// Module: transmitter
// Description: This module simulates the behavior of a UART transmitter.
// It takes an 8-bit data input and transmits it serially based on clock and enable signals.
// The transmission process involves four states: IDLE, START, DATA, and STOP.

module transmitter(
    input wire [7:0] data_in,  // 8-bit input data to be transmitted
    input wire Tx_en,          // Transmit enable signal
    input wire clk_50m,        // 50 MHz clock signal
    input wire clken,          // Clock enable for controlling transmission timing
    output reg Tx,             // Serial output transmitting data bit-by-bit
    output wire Tx_busy        // Signal indicating the transmitter is busy
);

    // Initial condition: Set the transmission line to high (idle state)
    initial begin
        Tx = 1'b1;
    end

    // State definitions using 2-bit encoding for the finite state machine (FSM)
    parameter TX_STATE_IDLE  = 2'b00;  // IDLE state: waiting for enable signal
    parameter TX_STATE_START = 2'b01;  // START state: start bit transmission
    parameter TX_STATE_DATA  = 2'b10;  // DATA state: data bits transmission
    parameter TX_STATE_STOP  = 2'b11;  // STOP state: stop bit transmission to complete the
cycle

    // Internal registers
    reg [7:0] data = 8'h00;          // Buffer to hold data being transmitted
    reg [2:0] bit_pos = 3'h0;        // Bit position counter for data transmission
    reg [1:0] state = TX_STATE_IDLE; // Current state of the FSM

    // FSM for controlling transmission based on state and clock signals
    always @(posedge clk_50m) begin
        case (state)
            TX_STATE_IDLE: begin  // Wait for enable signal to start transmission
                if (~Tx_en) begin
                    state <= TX_STATE_START;
                    data <= data_in;   // Load data from input
                    bit_pos <= 3'h0;   // Reset bit position
                end
            end
            TX_STATE_START: begin  // Transmit start bit (logic low)
                if (clken) begin
                    Tx <= 1'b0;
                    state <= TX_STATE_DATA;
                end
            end
            TX_STATE_DATA: begin  // Transmit data bits
                if (clken) begin
                    Tx <= data[bit_pos];
                    bit_pos <= bit_pos + 1;
                    if (bit_pos == 3'h7)  // Check if all bits are transmitted
                        state <= TX_STATE_STOP;
                end
            end
            TX_STATE_STOP: begin  // Transmit stop bit (logic high) and return to idle
                if (clken) begin
                    Tx <= 1'b1;
                    state <= TX_STATE_IDLE;
                end
            end
            default: begin  // Default case to handle unexpected states
```

```
62                  Tx <= 1'b1;  // Ensure line is idle
63                  state <= TX_STATE_IDLE;
64              end
65          endcase
66      end
67
68      // Output busy signal when the transmitter is not in IDLE state
69      assign Tx_busy = (state != TX_STATE_IDLE);
70
71  endmodule
72
```

Synthesized circuit:

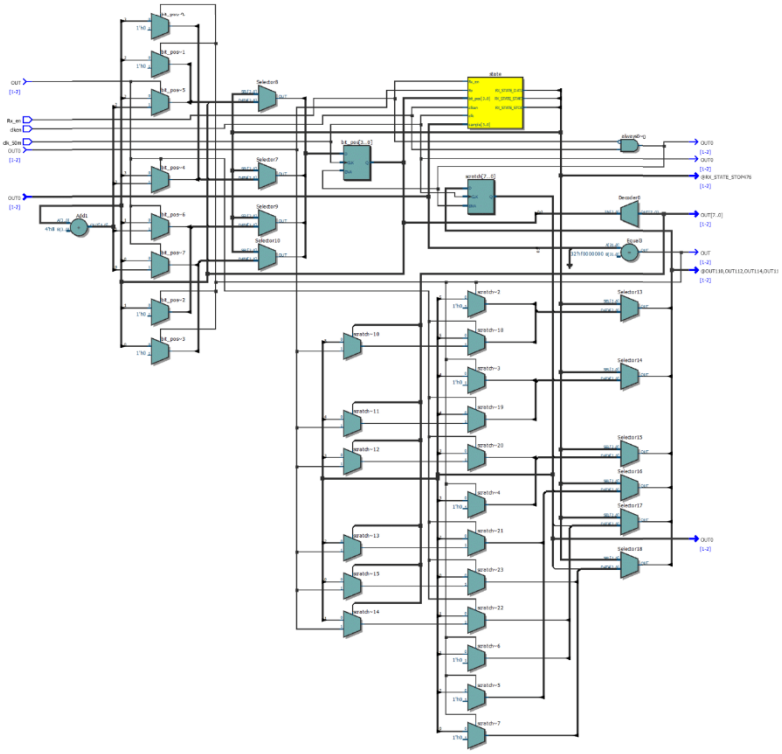


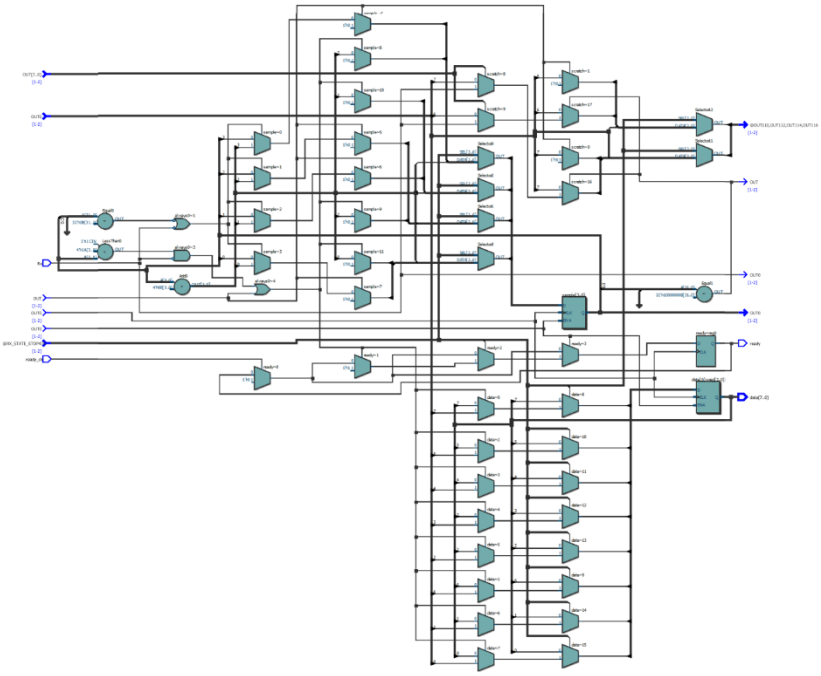
Receiver Implementation

Verilog code:

```
// Module: receiver
// Description: Implements a UART receiver that uses an FSM to handle data reception.
// It supports oversampling and handles synchronization issues by checking the received data
// across multiple clock cycles. The receiver processes data only when enabled by Rx_en.

module receiver (
    input wire Rx,             // Serial input receiving data
    input wire Rx_en,          // Receiver enable signal
    output reg ready,          // Signal to indicate data is ready to be read
    input wire ready_clr,      // Signal to clear the ready state
    input wire clk_50m,        // System clock
    input wire clken,          // Clock enable for controlling reception timing
    output reg [7:0] data      // Output data register
);

    // Initialize ready and data signals
    initial begin
        ready = 1'b0;          // Set ready to 0 initially
        data = 8'b0;           // Clear data initially
    end

    // Define states for the reception process
    parameter RX_STATE_START = 2'b00;  // Waiting for start bit
    parameter RX_STATE_DATA  = 2'b01;  // Receiving data bits
    parameter RX_STATE_STOP  = 2'b10;  // Checking stop bit

    // Internal state registers
    reg [1:0] state = RX_STATE_START;  // Initial state
    reg [3:0] sample = 0;              // Sample counter
    reg [3:0] bit_pos = 0;             // Position in the data byte being received
    reg [7:0] scratch = 8'b0;          // Temporary storage for the incoming data

    // Process incoming data on the positive edge of the system clock
    always @(posedge clk_50m) begin
        if (ready_clr)
            ready <= 1'b0;  // Reset ready signal when cleared

        if (clken && ~Rx_en) begin  // Only process data if clock is enabled and Rx is not enabled
            case (state)
                RX_STATE_START: begin  // Handle the start bit
                    if (!Rx || sample != 0)  // Check for the start condition
                        sample <= sample + 1;  // Increment sample counter
                    if (sample == 15) begin  // If a full bit has been sampled
                        state <= RX_STATE_DATA;  // Move to data receiving state
                        bit_pos <= 0;  // Reset bit position
                        sample <= 0;   // Reset sample counter
                        scratch <= 0;  // Clear scratch register
                    end
                end
                RX_STATE_DATA: begin  // Handle data reception
                    sample <= sample + 1;  // Increment sample counter
                    if (sample == 8) begin  // Midpoint of data bit sampling
                        scratch[bit_pos[2:0]] <= Rx;  // Store bit in scratch register
                        bit_pos <= bit_pos + 1;  // Move to the next bit
                    end
                    if (bit_pos == 8 && sample == 15)  // If last bit sampled
                        state <= RX_STATE_STOP;  // Move to stop bit verification
                end
                RX_STATE_STOP: begin  // Handle stop bit
                    if (sample == 15 || (sample >= 8 && !Rx)) begin  // Verify stop bit condition

                        state <= RX_STATE_START;  // Reset to start for new transmission
                        data <= scratch;  // Transfer received data
                        ready <= 1'b1;  // Indicate that data is ready
                        sample <= 0;  // Reset sample counter
                    end else
                        sample <= sample + 1;  // Continue sampling stop bit
                end
                default: begin  // Default case to handle unexpected states
                    state <= RX_STATE_START;  // Reset to initial state
                end
            endcase
        end
    end

endmodule

```

Synthesized circuit:



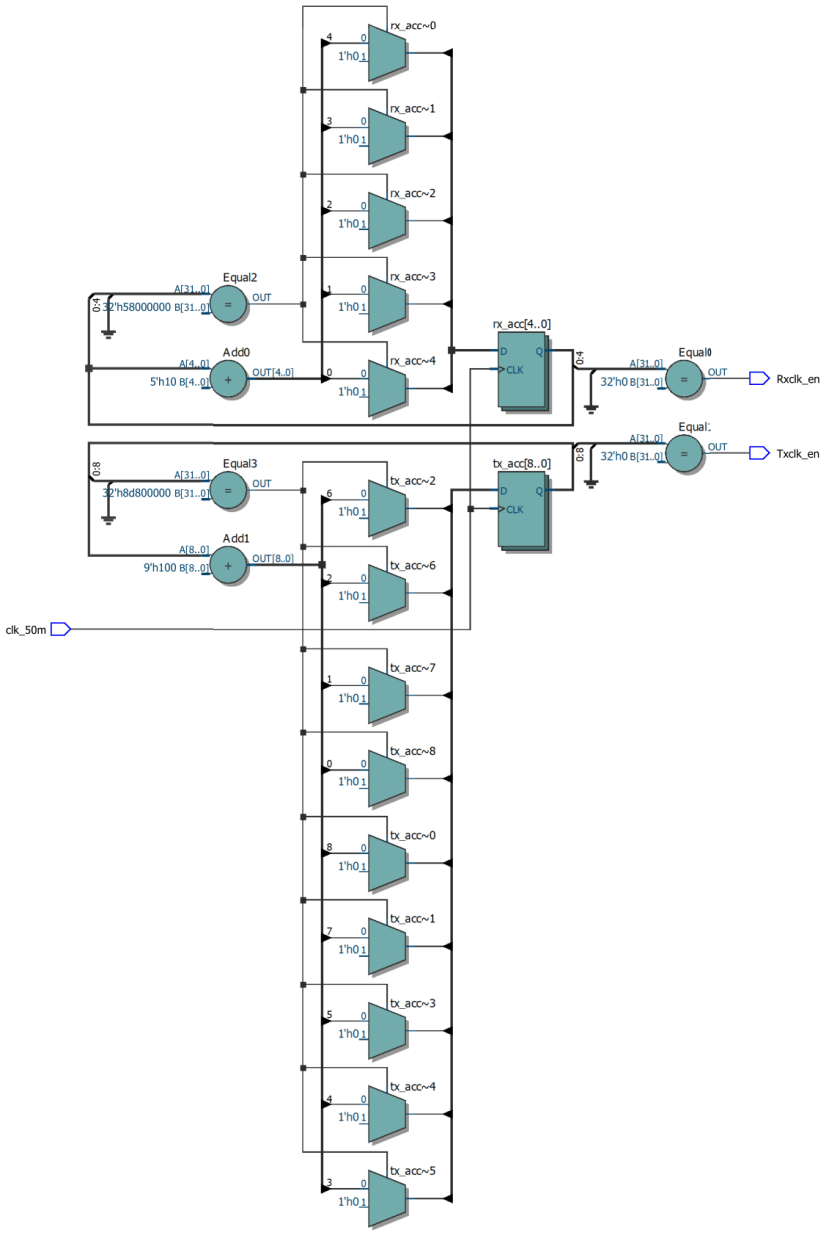


Baud rate Matching

Verilog code:

```
1   // Module: baudrate
2   // Description: This module divides a 50 MHz input clock to generate enable signals
3   // for both transmitting (Tx) and receiving (Rx) that operate at a baud rate of 115200.
4   // The Rx clock is additionally oversampled by 16x to increase the reliability of data
    reception.
5
6   module baudrate(
7       input wire clk_50m,       // Input clock at 50 MHz
8       output wire Rxclk_en,     // Enable signal for the Rx clock, oversampled
9       output wire Txclk_en      // Enable signal for the Tx clock
10  );
11
12      // Parameters for calculating the number of clock cycles per baud bit
13      parameter RX_ACC_MAX = 50000000 / (115200 * 16); // Calculate max count for Rx
    oversampled by 16x
14      parameter TX_ACC_MAX = 50000000 / 115200;        // Calculate max count for Tx
15      parameter RX_ACC_WIDTH = $clog2(RX_ACC_MAX);     // Determine bit width needed for Rx
    accumulator
16      parameter TX_ACC_WIDTH = $clog2(TX_ACC_MAX);     // Determine bit width needed for Tx
    accumulator
17
18      // Accumulators for baud rate generation
19      reg [RX_ACC_WIDTH - 1:0] rx_acc = 0;  // Rx accumulator, initialized to 0
20      reg [TX_ACC_WIDTH - 1:0] tx_acc = 0;  // Tx accumulator, initialized to 0
21
22      // Generate Rx and Tx clock enable signals
23      // Enable signals are active when accumulators reset
24      assign Rxclk_en = (rx_acc == 0);
25      assign Txclk_en = (tx_acc == 0);
26
27      // Baud rate clock generation for Rx
28      // This clock is oversampled by 16x for better sampling of incoming data
29      always @(posedge clk_50m) begin
30          if (rx_acc == RX_ACC_MAX - 1)  // Check if the accumulator has reached its max value
31              rx_acc <= 0;               // Reset the accumulator
32          else
33              rx_acc <= rx_acc + 1;      // Increment the accumulator
34      end
35
36      // Baud rate clock generation for Tx
37      // This clock matches the baud rate of 115200 for data transmission
38      always @(posedge clk_50m) begin
39          if (tx_acc == TX_ACC_MAX - 1)  // Check if the accumulator has reached its max value
40              tx_acc <= 0;               // Reset the accumulator
41          else
42              tx_acc <= tx_acc + 1;      // Increment the accumulator
43      end
44
45  endmodule
46
```

Synthesized circuit:

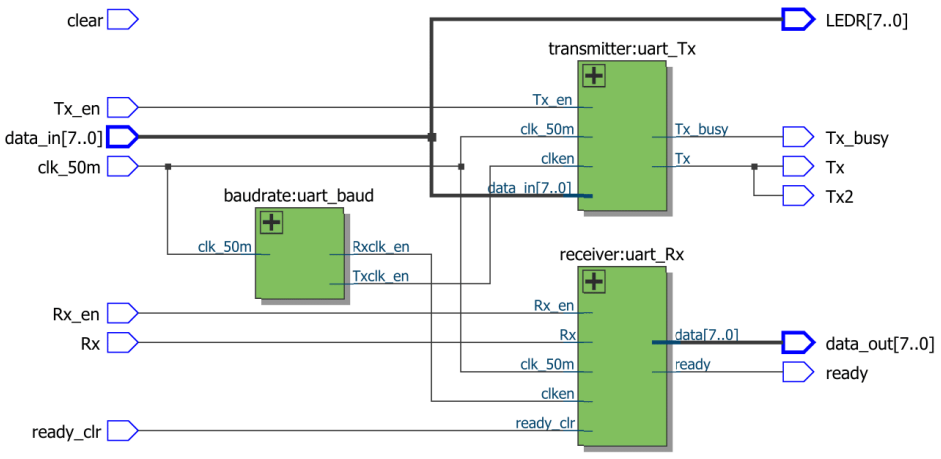


Final UART Tx-Rx Circuit

Verilog code:

```
// Module: uart
// Description: This module integrates the components of a UART interface including
// a baud rate generator, a transmitter, and a receiver. It handles both transmitting
// and receiving data at a specified baud rate, controlled by enabling signals.

module uart(
    input wire [7:0] data_in,   // 8-bit input data to be transmitted
    input wire Tx_en,           // Enable signal for transmitter
    input wire clear,           // Not used in this instantiation (consider removal if not
required)
    input wire clk_50m,         // System clock at 50 MHz
    output wire Tx,             // Transmitted serial data output
    output wire Tx_busy,        // Signal indicating transmitter is busy
    input wire Rx,              // Received serial data input
    input wire Rx_en,           // Enable signal for receiver
    output wire ready,          // Signal to indicate data is ready to be read
    input wire ready_clr,       // Signal to clear the ready state
    output wire [7:0] data_out, // 8-bit output data received
    output [7:0] LEDR,          // LED output directly reflecting input data (for debugging
or status)
    output wire Tx2             // Duplicate of Tx for additional interfacing
);

    // Assign LEDs to mirror input data for visual debugging or demonstration
    assign LEDR = data_in;

    // Duplicate the Tx signal to an additional output pin for further use
    assign Tx2 = Tx;

    // Internal connections for baud rate enable signals
    wire Txclk_en, Rxclk_en;

    // Instantiate the baud rate generator
    baudrate uart_baud(
        .clk_50m(clk_50m),
        .Rxclk_en(Rxclk_en),    // Enable signal for the receiver clock
        .Txclk_en(Txclk_en)     // Enable signal for the transmitter clock
    );

    // Instantiate the transmitter module
    transmitter uart_Tx(
        .data_in(data_in),
        .Tx_en(Tx_en),
        .clk_50m(clk_50m),
        .clken(Txclk_en),       // Use Tx clock enable for transmitter operation
        .Tx(Tx),
        .Tx_busy(Tx_busy)
    );

    // Instantiate the receiver module
    receiver uart_Rx(
        .Rx(Rx),
        .Rx_en(Rx_en),
        .ready(ready),
        .ready_clr(ready_clr),
        .clk_50m(clk_50m),
        .clken(Rxclk_en),       // Use Rx clock enable for receiver operation
        .data(data_out)
    );

endmodule

```

Synthesized circuit:



This implementation accepts an 8-bit input and transmits it from the transmitter. While transmitting, it is crucial to monitor the state of the Tx_en port. Subsequently, the receiver accepts this input and displays it on the LEDs. Similarly, in the receiver, it is essential to consider the state of the Rx_en port to begin receiving.

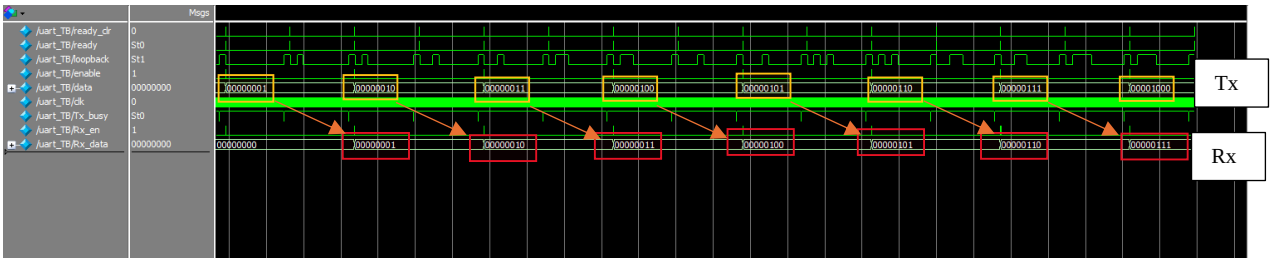
A small testbench is given below:

Test-bench.

Verilog code:

```verilog
// Testbench Module: uart_TB
// Description: This testbench is designed to verify the functionality of a UART module
// by implementing a serial loopback. It transmits data bytes and checks if the received
// data matches the transmitted data. This ensures both the transmitter and receiver
// components of the UART are functioning correctly.

//`include "uart.v" // Include the UART module definition

module uart_TB();

    // Testbench control signals
    reg [7:0] data = 0;      // Data to be transmitted
    reg clk = 0;             // Test clock signal
    reg enable = 0;          // Enable signal for transmitter
    reg Rx_en = 0;           // Enable signal for receiver

    // UART module outputs to be monitored
    wire Tx_busy;            // Indicates if the transmitter is busy
    wire ready;              // Indicates if the receiver has data ready
    wire [7:0] Rx_data;      // Holds the data received from the UART

    // Loopback wire for connecting Tx and Rx internally
    wire loopback;
    reg ready_clr = 0;       // Signal to clear the 'ready' flag in the receiver

    // Instantiation of the UART module
    uart test_uart(
        .data_in(data),
        .Tx_en(enable),
        .clk_50m(clk),
        .Tx(loopback),
        .Tx_busy(Tx_busy),
        .Rx(loopback),
        .ready(ready),
        .ready_clr(ready_clr),
        .Rx_en(Rx_en),       // Connect the Rx_en signal
        .data_out(Rx_data)
    );

    // Initial block to setup and start the test
    initial begin
        $dumpfile("uart.vcd");  // Set up the VCD file for waveform analysis
        $dumpvars(0, uart_TB);  // Record simulation data for all variables in the testbench

        // Initial state of control signals
        enable <= 1'b1;  // Initially enable the transmitter
        Rx_en <= 1'b1;   // Initially enable the receiver
        #2 enable <= 1'b0;  // Disable after a short delay to simulate a transmission trigger
        #2 Rx_en <= 1'b0;   // Disable after a short delay
    end

    // Clock generation
    always begin
        #1 clk = ~clk;  // Toggle the clock every time unit to simulate a 50MHz clock
    end

    // Check the received data when it is ready
    always @(posedge ready) begin
        #2 ready_clr <= 1;  // Clear the ready signal after a delay to process the received
data
        #2 ready_clr <= 0;  // Reset the ready clear signal
        if (Rx_data != data) begin

            // If the received data does not match the sent data, print an error message
            $display("FAIL: rx data %x does not match tx %x", Rx_data, data);
            $finish;  // End the simulation
        end else begin
            // Check for specific data value to determine end of the test
            if (Rx_data == 8'h2) begin  // Arbitrary end condition based on expected test
data sequence
                $display("SUCCESS: all bytes verified");
                $finish;  // End the simulation on success
            end
            // Prepare for the next test iteration
            data <= data + 1'b1;  // Increment the data to send
            enable <= 1'b1;       // Re-enable the transmitter
            Rx_en <= 1'b1;        // Re-enable the receiver
            #2 enable <= 1'b0;    // Toggle enable signals to mimic behavior
            #2 Rx_en <= 1'b0;
        end
    end
endmodule

```

Simulated Waveform:



Codes given is an example of transmitter to transmit pre-defined byte and increment it 1 by 1:

```
module transmitter(
    input wire wr_en,      // Enable wire to start
    input wire clk_50m,    // Clock signal for the transmitter
    input wire clken,      // Clock enable for the transmitter
    output reg Tx,         // A single 1-bit register variable to hold transmitting bit
    output wire Tx_busy    // Transmitter is busy signal
);

// Initialization
initial begin
    Tx = 1'b1; // Initialize Tx to 1 to begin the transmission
end

// Define the 4 states using parameters
parameter TX_STATE_IDLE  = 2'b00;
parameter TX_STATE_START = 2'b01;
parameter TX_STATE_DATA  = 2'b10;
parameter TX_STATE_STOP  = 2'b11;

// Data handling
reg [7:0] data_in = 8'b00001011;
reg [7:0] data = 8'h00;
reg [2:0] bit_pos = 3'h0;
reg [1:0] state = TX_STATE_IDLE;

// Timer counter to increment data_in every 2 seconds
reg [26:0] counter = 0; // 27-bit counter, enough to count up to 100 million

always @(posedge clk_50m) begin
    // Increment the counter
    counter <= counter + 1;
    if (counter >= 100000000) begin
        counter <= 0;          // Reset the counter every 2 seconds
        data_in <= data_in + 1; // Increment data_in
    end
end

always @(posedge clk_50m) begin
    case (state) // Consider the 4 states of the transmitter
    TX_STATE_IDLE: begin
        if (~wr_en) begin
            state <= TX_STATE_START; // Assign the start signal to state
            data <= data_in;         // Assign input data vector to the current data
            bit_pos <= 3'h0;         // Assign the bit position to zero
        end
    end
    TX_STATE_START: begin
        if (clken) begin
            Tx <= 1'b0; // Set Tx = 0 indicating transmission has started
            state <= TX_STATE_DATA;
        end
    end
    TX_STATE_DATA: begin
        if (clken) begin
            if (bit_pos == 3'h7) // Transmit all bits from 0 to 7
                state <= TX_STATE_STOP; // When bit position has finally reached 7, assign
state to stop transmission
            else
                bit_pos <= bit_pos + 3'h1; // Increment the bit position
            Tx <= data[bit_pos]; // Set Tx to the data value of the bit position
        end
    end

    TX_STATE_STOP: begin
        if (clken) begin
            Tx <= 1'b1; // Set Tx = 1 after transmission has ended
            state <= TX_STATE_IDLE; // Move to IDLE state once a transmission has been
completed
        end
    end
    default: begin
        Tx <= 1'b1; // Always begin with Tx = 1 and state assigned to IDLE
        state <= TX_STATE_IDLE;
    end
    endcase
end

assign Tx_busy = (state != TX_STATE_IDLE); // Assign the BUSY signal when the transmitter
is not idle

endmodule
```