Department of Electronic & Telecommunication Engineering
University of Moratuwa, Sri Lanka

UART Implementation in FPGA Report

Submitted by:
Jayakody J.A.K. - 220247J & Jayarathne H.A.C.N. - 220252U
Team 12

Submitted in Partial Fulfillment of the Requirements for the Module
EN 2111 – Electronic Circuit Design

09/05/2025

Contents

1 Introduction

This report describes the design and functionality of a top-level system composed of a UART transceiver and a 3-digit multiplexed display. The transceiver integrates UART (Universal Asynchronous Receiver/Transmitter) communication with a 7-segment display that shows data received over UART. A 4-bit switch input is used to send data, and the transmitted data is displayed on a 3-digit multiplexed 7-segment display.

The system is built using System Verilog and consists of several modules, including:

1. **transceiver**: Handles data transmission via UART and manages the 7-segment display.
2. **display_3digit_multiplexed**: Displays a number on a 3-digit 7-segment display.
3. **uart**: The base UART module used for data transmission and reception.
4. **seven_segment_decoder**: A module for decoding 4-bit values to 7-segment display values.

UART (Universal Asynchronous Receiver/Transmitter) is a serial communication protocol used for transmitting and receiving data between devices without a shared clock. The **8-bit UART protocol** specifically refers to a data frame that carries **8 bits (1 byte)** of actual data.

1.1 Structure of an 8-bit UART Frame

An 8-bit UART data frame typically includes:

- In IDLE state the line is held HIGH

1. **Start Bit (1 bit)**
   - Logic level **0** (LOW)
   - Indicates the beginning of a data frame.
2. **Data Bits (8 bits)**
   - Actual data payload (can range from 0 to 255).
   - Sent LSB (Least Significant Bit) first by default.
3. **Optional Parity Bit (0 or 1 bit) Not used in this implementation**
   - Error-checking bit (not used in your current design).
4. **Stop Bit (5 bits were used as guard bits)**
   - Logic level **1** (HIGH)
   - Marks the end of the data frame.

1.2 Timing and Synchronization

- **Asynchronous Communication**: No clock is shared; both sender and receiver must agree on a common **baud rate** (We used 9600 as the baud rate).
- Each bit is held for a fixed duration (determined by the baud rate 5208 cycles at 50MHz in our case), and the receiver samples bits at the center of each bit period.

2 System Overview

2.1 Top-Level System (transceiver)

The top-level module integrates UART transmission and reception along with driving a 3-digit multiplexed 7-segment display. The components of this module include:

- **4-bit Switch (sw)**: Provides input for transmission.
- **Button (btn)**: Triggers data transmission when pressed.
- **3-Digit 7-Segment Display (seg and an)**: Displays the data received via UART.
- **UART Interface**: Handles serial data transmission and reception between the system and external devices.

The Baud rate of the system is configured to be 9600 and the word width is set to be 8 bits.

2.2 3-Digit Multiplexed Display (display_3digit_multiplexed)

The display module controls the 3-digit multiplexed 7-segment display and converts the 8-bit data received via UART into a 3-digit decimal number. The module handles the following:

- **Input**: 8-bit data (num) representing the value to be displayed.
- **Output**: 7-segment segment control (seg) and active-low digit enable (an).

The module splits the 8-bit value into hundreds, tens, and units, displaying each value on a corresponding digit. It refreshes at a 1 kHz rate using the 50 MHz clock.

2.3 UART Communication

The UART module implements serial data transmission and reception:

- **Transmit**: The data is sent out using the tx line when the btn (send trigger) is pressed, and the sw input provides the 4-bit data padded with leading zeros to be transmitted.
- **Receive**: The data is received from an external source through the rx line and is stored in m_data when the receiver indicates that valid data is available (m_valid).

3 Detailed Design

3.1 Top-Level System Design – `transceiver`

The `transceiver` module integrates UART communication with a 3-digit 7-segment display to form a simple serial data transmission and display system.

Main Features

- **4-bit Switch (`sw`):** User input data ranging from 0 to 15.
- **Push Button (`btn`):** Triggers UART data transmission.
- **UART Interface:** Sends zero-extended 8-bit data (`0000xxxx`) via `tx` and receives 8-bit data via `rx`.
- **3-digit 7-Segment Display (`seg`, `an`):** Displays the received 8-bit value (0–255).

System Configuration

- **Baud Rate:** 9600
- **Clock Frequency:** 50 MHz
- **Word Length:** 8 bits
- **UART Format:** 1 start bit, 8 data bits, 1 stop bit

Operation

1. When the `btn` is pressed, the 4-bit `sw` value is sent over UART.
2. The received UART data is latched and stored internally.
3. The display module continuously multiplexes the stored data across 3 digits to show the full 8-bit number.

```
module transceiver #(
  parameter CLOCKS_PER_PULSE = 5208,
            BITS_PER_WORD    = 8
)(
  input  logic clk,
  input  logic rstn,
  output logic clk_test,

  input  logic [3:0] sw,          // 4-bit switch input
  input  logic btn,               // Send trigger button
  output logic [6:0] seg,         // 7-segment display output
  output logic [2:0] an,

  output logic tx,                // UART transmit
  input  logic rx                 // UART receive
);

  // UART TX signals
  logic s_valid, s_ready;
  logic [7:0] s_data;

  // UART RX signals
  logic m_valid;
  logic [7:0] m_data;

  // Display latch
  logic [7:0] display_data;

  //Clock output for measurements
  assign clk_test = clk;

  // Instantiate UART
  uart #(
    .CLOCKS_PER_PULSE(CLOCKS_PER_PULSE),
    .BITS_PER_WORD(BITS_PER_WORD)
  ) uart_inst (
    .clk(clk),
    .rstn(rstn),
    .s_valid(s_valid),
    .s_data(s_data),
    .s_ready(s_ready),
    .tx(tx),
    .rx(rx),
    .m_valid(m_valid),
    .m_data(m_data)
  );
```

Figure 1: Tranceiver Top Level module

```
// Button edge detection & transmit control
  logic btn_prev;
  always_ff @(posedge clk or negedge rstn) begin
    if (!rstn) begin
      btn_prev <= 0;
      s_valid <= 0;
      s_data <= 0;
    end else begin
      btn_prev <= btn;
      if (btn && !btn_prev && s_ready) begin
        s_data <= {4'b0000, sw};
        s_valid <= 1;
      end else begin
        s_valid <= 0;
      end
    end
  end

  // Latch every valid received value
  always_ff @(posedge clk or negedge rstn) begin
   //display_data <= m_data;
    if (!rstn)
      display_data <= 8'd0;
      else if (m_valid)
      display_data <= m_data;
  end

  // Drive 3-digit display
  display_3digit_multiplexed display_inst (
    .clk(clk),
    .num(m_data),
    .an(an),
    .seg(seg)
  );

endmodule
```

Figure 2: Tranceiver Top Level modulen

3.2 Data Transmission

When the button (btn) is pressed, the 4-bit switch input (sw) is latched, and a UART transmission is triggered. The btn acts as a trigger for data transmission, and s_valid is asserted when the data is valid and ready to be sent. The transmitted data is packed as an 8-bit value ({4'b0000, sw}) and transmitted through the tx line. The system waits for the s_ready signal, which indicates that the UART transmitter is ready to send new data. We can use this to feed parallel data to the transmitter module. The master can be signaled that the transmitter is ready with it.

3.2.1 UART Transmitter (uart_tx)

The UART transmitter module implements a finite state machine (FSM) to transmit data over a serial connection. The system works as follows:

- **IDLE state**: The system waits for s_valid to indicate that valid data is available for transmission.
- **SEND state**: Once valid data is ready, the system starts sending the start bit (0), followed by the 8 data bits, and then the stop bit (1).

The transmitter is controlled by s_valid (data validity) and outputs tx (the serial data line) and s_ready (indicating when the transmitter is ready for new data).

```
module uart_tx #(
  parameter CLOCKS_PER_PULSE = 5208,                // 50 MHz / 9600
      baud
            BITS_PER_WORD    = 8,
            PACKET_SIZE      = BITS_PER_WORD + 5     // 1 start + 8
               data + 2 stop + (optional parity)
)(
  input  logic clk,
  input  logic rstn,
  input  logic s_valid,
  input  logic [BITS_PER_WORD-1:0] s_data,
  output logic tx,
  output logic s_ready
);
  // Constants
  localparam END_BITS = PACKET_SIZE - BITS_PER_WORD - 1; // Number of
      stop bits (assuming 2)
  // FSM States
  typedef enum logic {IDLE, SEND} state_t;
  state_t state;
  // Internal registers
  logic [PACKET_SIZE-1:0] s_packet;
  logic [PACKET_SIZE-1:0] s_packet_reg;
  logic [$clog2(PACKET_SIZE)-1:0] c_bits;
  logic [$clog2(CLOCKS_PER_PULSE)-1:0] c_clocks;
  // Format packet: stop bits (1's), data, start bit (0)
  always_comb begin
    s_packet = { {END_BITS{1'b1}}, s_data, 1'b0 }; // LSB first
  end
  // Output is the LSB of the shift register
  assign tx = s_packet_reg[0];
```

Figure 3: UART Transmitter Module Implementation

```
  // FSM and logic
  always_ff @(posedge clk or negedge rstn) begin
    if (!rstn) begin
      state          <= IDLE;
      s_packet_reg   <= '1; // Idle line is high
      c_bits         <= 0;
      c_clocks       <= 0;
    end else begin
      case (state)
        IDLE: begin
          if (s_valid) begin
            s_packet_reg <= s_packet;
            c_bits       <= 0;
            c_clocks     <= 0;
            state        <= SEND;
          end
        end
        SEND: begin
          if (c_clocks == CLOCKS_PER_PULSE - 1) begin
            c_clocks <= 0;
            if (c_bits == PACKET_SIZE - 1) begin
              state        <= IDLE;
              s_packet_reg <= '1; // Reset line to idle (high)
            end else begin
              c_bits       <= c_bits + 1;
              s_packet_reg <= s_packet_reg >> 1; // Shift next bit
                  out
            end
          end else begin
            c_clocks <= c_clocks + 1;
          end
        end
        default: state <= IDLE;
      endcase
    end
  end
  // s_ready indicates the module is ready for new data
  assign s_ready = (state == IDLE);
endmodule
```

Figure 4: UART Transmitter Module Implementation

3.3 Data Reception

When the data is received by the UART receiver, the m_valid signal is asserted, and the received 8-bit data (m_data) is latched into display_data. This data is then passed to the 3-digit multiplexed display for visualization.

3.3.1 UART Receiver (uart_rx)

The UART receiver operates similarly but in reverse:

- **IDLE state**: Waits for the falling edge of the start bit (rx == 0).
- **START state**: The receiver waits for the middle of the start bit to stabilize and then begins sampling the data bits.

- **DATA state**: Samples 8 bits of data.
- **STOP state**: Validates the stop bit (rx == 1) and outputs the received data.

```
module uart_rx #(
  parameter CLOCKS_PER_PULSE = 5208, // For 9600 baud at 200 MHz
            BITS_PER_WORD    = 8
)(
  input  logic clk,
  input  logic rstn,
  input  logic rx,
  output logic m_valid,
  output logic [BITS_PER_WORD-1:0] m_data
);

  localparam PACKET_SIZE = BITS_PER_WORD + 2;  // 1 start + 8 data +
     1 stop
  typedef enum logic [1:0] {IDLE, START, DATA, STOP} state_t;

  state_t state;

  // Internal registers
  logic [$clog2(CLOCKS_PER_PULSE)-1:0] c_clocks;
  logic [$clog2(BITS_PER_WORD)-1:0]    c_bits;
  logic [BITS_PER_WORD-1:0]            shift_reg;
```

Figure 5: UART Rceiver Module Implementation

```
  always_ff @(posedge clk or negedge rstn) begin
    if (!rstn) begin
      state      <= IDLE;
      c_clocks   <= 0;
      c_bits     <= 0;
      shift_reg  <= 0;
      m_data     <= 0;
      m_valid    <= 0;
    end else begin
      m_valid <= 0;  // default

      case (state)
        // Wait for falling edge of start bit
        IDLE: if (rx == 0) begin
          state    <= START;
          c_clocks <= 0;
        end

        // Wait until middle of start bit
        START: if (c_clocks == CLOCKS_PER_PULSE / 2 - 1) begin
          c_clocks <= 0;
          c_bits   <= 0;
          state    <= DATA;
        end else begin
          c_clocks <= c_clocks + 1;
        end

        // Sample 8 data bits
        DATA: if (c_clocks == CLOCKS_PER_PULSE - 1) begin
          c_clocks <= 0;
          shift_reg <= {rx, shift_reg[BITS_PER_WORD-1:1]};  // LSB
              first

          if (c_bits == BITS_PER_WORD - 1) begin
            state <= STOP;
          end else begin
            c_bits <= c_bits + 1;
          end
        end else begin
          c_clocks <= c_clocks + 1;
        end

        // Wait one stop bit
        STOP: if (c_clocks == CLOCKS_PER_PULSE - 1) begin
          c_clocks <= 0;
          m_data   <= shift_reg;
          m_valid  <= 1;
          state    <= IDLE;
        end else begin
          c_clocks <= c_clocks + 1;
        end
      endcase
    end
  end

endmodule
```

Figure 6: UART Transmitter Module Implementation

When the receiver successfully decodes a byte, it sets m_valid high, indicating the data is available in m_data.

```
module uart #(
  parameter CLOCKS_PER_PULSE = 5208,  // For 9600 baud at 50 MHz
            BITS_PER_WORD    = 8
)(
  input  logic clk,
  input  logic rstn,

  // TX interface
  input  logic s_valid,                  //from data source  , input control to the transmitter
  input  logic [BITS_PER_WORD-1:0] s_data,  //from data source  , input data to the transmitter
  output logic s_ready,                //to the data source
  output logic tx,

  // RX interface
  input  logic rx,
  output logic m_valid,                //data output validity
  output logic [BITS_PER_WORD-1:0] m_data   //output data
);

  // Instantiate TX module
  uart_tx #(
    .CLOCKS_PER_PULSE(CLOCKS_PER_PULSE),
    .BITS_PER_WORD(BITS_PER_WORD)
  ) tx_inst (
    .clk(clk),
    .rstn(rstn),
    .s_valid(s_valid),
    .s_data(s_data),
    .tx(tx),
    .s_ready(s_ready)
  );

  // Instantiate RX module
  uart_rx #(
    .CLOCKS_PER_PULSE(CLOCKS_PER_PULSE),
    .BITS_PER_WORD(BITS_PER_WORD)
  ) rx_inst (
    .clk(clk),
    .rstn(rstn),
    .rx(rx),
    .m_valid(m_valid),
    .m_data(m_data)
  );

endmodule
```

Figure 7: Full UART Module Implementation

3.4 3-Digit Multiplexed Display

The display_3digit_multiplexed module works by converting the 8-bit received data into hundreds, tens, and units, and displaying each on the corresponding digit of the 7-segment display. The display is multiplexed to ensure that only one digit is enabled at a time, switching between digits at a 1 kHz refresh rate.

- **num Input**: The received 8-bit data is split into three digits (hundreds, tens, units).
- **an Output**: A 3-bit active-low control signal that enables one digit at a time.
- **seg Output**: A 7-bit signal that controls the 7 segments of the display (each segment is active LOW in this case).

```
module display_3digit_multiplexed (
  input  logic        clk,          // System clock
  input  logic [7:0]  num,          // 8-bit number (0  255  )
  output logic [6:0]  seg,          // Segment outputs (gfedcba)
  output logic [2:0]  an            // Digit enable (active LOW)
);

  logic [3:0] digit_val;
  logic [1:0] digit_idx;
  logic [15:0] refresh_counter;
  logic [6:0] seg_raw;

  // Convert number to BCD digits
  logic [3:0] hundreds, tens, units;

  always_comb begin
    hundreds = num / 100;
    tens     = (num % 100) / 10;
    units    = num % 10;
  end

  // 1 kHz refresh rate from 50 MHz clock
  always_ff @(posedge clk) begin
    refresh_counter <= refresh_counter + 1;
    if (refresh_counter == 49999) begin
      refresh_counter <= 0;
      digit_idx <= digit_idx + 1;
    end
  end
```

Figure 8: 3 digit 7 segment display implementation

```
  // Select current digit value based on digit_idx
  always_comb begin
    case (digit_idx)
      2'd0: begin
        digit_val = units;
        an = 3'b001; // Enable rightmost digit (active-high)
      end
      2'd1: begin
        digit_val = tens;
        an = 3'b010; // Middle digit
      end
      2'd2: begin
        digit_val = hundreds;
        an = 3'b100; // Leftmost digit
      end
      default: begin
        digit_val = 4'd0;
        an = 3'b000; // All off
      end
    endcase
  end

  // Decode digit to segments
  seven_segment_decoder decoder (
    .num(digit_val),
    .seg(seg_raw)
  );


  assign seg = seg_raw;

endmodule
```

Figure 9: 3 digit 7 segment display implementation

```
module seven_segment_decoder (
  input logic [3:0] num,     // 4-bit input (0-9)
  output logic [6:0]seg    // 7-bit output (gfedcba)
);

  always_comb begin
    case (num)
      4'd0     : seg = ~7'b0111111; //0
      4'd1     : seg = ~7'b0000110; //1
      4'd2     : seg = ~7'b1011011; //2
      4'd3     : seg = ~7'b1001111; //3
      4'd4     : seg = ~7'b1100110; //4
      4'd5     : seg = ~7'b1101101; //5
      4'd6     : seg = ~7'b1111101; //6
      4'd7     : seg = ~7'b0000111; //7
      4'd8     : seg = ~7'b1111111; //8
      4'd9     : seg = ~7'b1101111; //9
      default : seg = ~7'b1000000; //-(Invalid Input)
    endcase
  end

  endmodule
```

Figure 10: digit 7 segment display implementation

3.5 Clocking and Control

The system is clocked by the internal 50 MHz clock, and the multiplexing refresh counter generates a 1 kHz refresh rate for the display. The btn press triggers the sending of the data, while the multiplexing logic ensures smooth and continuous display updates.

4 RTL Design Verification and Testbenches

4.1 Testbench Description

The testbenches for this design includes simulations of button presses, switch inputs, and UART communication. It ensures the correct data is transmitted and displayed on the 7-segment display. The transmission is not done in real clock rates considering the time and clock pulses needed for full scale simulations

- **Button Press Simulation**: The testbench simulates button presses that trigger the data transmission.
- **Switch Input Simulation**: The testbench feeds different 4-bit switch values (sw) and verifies that the corresponding data is transmitted and displayed.
- **UART Transmission and Reception**: The test bench verifies that the transmitted data is correctly received and displayed. Loop back was used to verify the behavior of the transmitter and the receiver.

4.1.1 UART TX test bench

```
1  `timescale 1ns/1ps
2
3  module uart_tx_tb;
4
5    // Parameters
6    localparam CLOCKS_PER_PULSE = 10;  // Use small value for faster
         simulation
7    localparam BITS_PER_WORD    = 8;
8    localparam PACKET_SIZE      = BITS_PER_WORD + 5;
9
10   // DUT Signals
11   logic clk;
12   logic rstn;
13   logic s_valid;
14   logic [BITS_PER_WORD-1:0] s_data;
15   logic tx;
16   logic s_ready;
17
18   // Instantiate the UART transmitter
19   uart_tx #(
20     .CLOCKS_PER_PULSE(CLOCKS_PER_PULSE),
21     .BITS_PER_WORD(BITS_PER_WORD),
22     .PACKET_SIZE(PACKET_SIZE)
23   ) dut (
24     .clk(clk),
25     .rstn(rstn),
26     .s_valid(s_valid),
27     .s_data(s_data),
28     .tx(tx),
29     .s_ready(s_ready)
30   );
31
32   // Clock generation: 100MHz
33   always #5 clk = ~clk; // 10ns period
```

Figure 11: UART TX test bench

```
  // Test sequence
   initial begin
     // Initialize
     clk = 0;
     rstn = 0;
     s_valid = 0;
     s_data = 8'h00;

     // Reset pulse
     #20;
     rstn = 1;

     // Wait for DUT to become ready
     @(posedge clk);
     wait (s_ready);

     // Send byte 0xA5 (10100101)
     s_data = 8'hA5;
     s_valid = 1;
     @(posedge clk);
     s_valid = 0;

     // Wait for transmission to complete
     wait (s_ready);

     // Another byte
     @(posedge clk);
     s_data = 8'h3C;
     s_valid = 1;
     @(posedge clk);
     s_valid = 0;

     // Wait and finish
     wait (s_ready);
     #100;
     $finish;
   end

   // Monitor TX line
   initial begin
     $dumpfile("uart_tx_tb.vcd");
     $dumpvars(0, uart_tx_tb);
     $display("Time\tTX");
     $monitor("%0t\t%b", $time, tx);
   end

endmodule
```

Figure 12: UART TX test bench

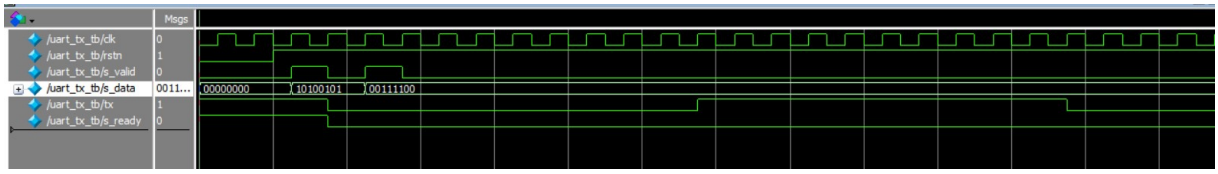

Figure 13: UART transmitter testing wave form

4.1.2 UART RX testbench

```
`timescale 1ns / 1ps

module uart_tx_rx_tb;

  // Parameters
  parameter CLOCKS_PER_PULSE = 10;  // Keep small for simulation
  parameter BITS_PER_WORD    = 8;

  // Signals
  logic clk = 0, rstn = 0;
  logic s_valid, s_ready;
  logic [BITS_PER_WORD-1:0] s_data;
  logic tx, rx;
  logic m_valid;
  logic [BITS_PER_WORD-1:0] m_data;

  // Clock generation
  always #5 clk = ~clk;  // 100 MHz

  // Instantiate TX
  uart_tx #(
    .CLOCKS_PER_PULSE(CLOCKS_PER_PULSE),
    .BITS_PER_WORD(BITS_PER_WORD)
  ) tx_inst (
    .clk(clk),
    .rstn(rstn),
    .s_valid(s_valid),
    .s_data(s_data),
    .tx(tx),
    .s_ready(s_ready)
  );

  // Connect TX to RX
  assign rx = tx;

  // Instantiate RX
  uart_rx #(
    .CLOCKS_PER_PULSE(CLOCKS_PER_PULSE),
    .BITS_PER_WORD(BITS_PER_WORD)
  ) rx_inst (
    .clk(clk),
    .rstn(rstn),
    .rx(rx),
    .m_valid(m_valid),
    .m_data(m_data)
  );
```

Figure 14: UART RX testbench

```
// Test vector
 logic [BITS_PER_WORD-1:0] test_data [0:2];
 int i, received;

 initial begin
   test_data[0] = 8'hA5;
   test_data[1] = 8'h3C;
   test_data[2] = 8'hF0;

   $display("Starting simulation...");
   rstn = 0;
   s_valid = 0;
   #50;
   rstn = 1;

   for (i = 0; i < 3; i++) begin
     // Wait until transmitter is ready
     @(posedge clk);
     wait (s_ready);
     s_data = test_data[i];
     s_valid = 1;
     @(posedge clk);
     s_valid = 0;

     // Wait for valid data from receiver
     wait (m_valid);
     $display("TX: %02X -> RX: %02X %s", test_data[i], m_data,
              (m_data == test_data[i]) ? "PASS" : "FAIL");

     // Give a few idle cycles between transmissions
     repeat (5) @(posedge clk);
   end

   $display("Simulation complete.");
   #50;
   $finish;
 end

endmodule
```

Figure 15: UART RX testbench

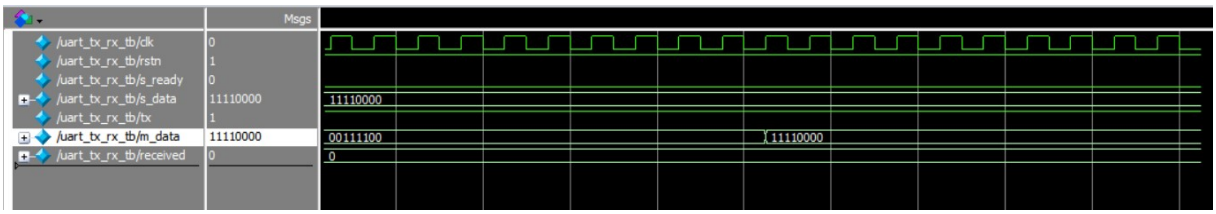

Figure 16: UART receiver wave form

4.1.3 Top Module Test bench

```
`timescale 1ns / 1ps

module transceiver_tb;

  parameter CLOCKS_PER_PULSE = 5208;  // Smaller for faster
      simulation
  parameter BITS_PER_WORD = 8;

  logic clk = 0;
  logic rstn;
  logic [3:0] sw;
  logic btn;
  logic tx, rx;
  logic [6:0] seg;

  // Clock generation: 100 MHz
  always #5 clk = ~clk;

  // DUT
  transceiver #(
    .CLOCKS_PER_PULSE(CLOCKS_PER_PULSE),
    .BITS_PER_WORD(BITS_PER_WORD)
  ) dut (
    .clk(clk),
    .rstn(rstn),
    .sw(sw),
    .btn(btn),
    .seg(seg),
    .tx(tx),
    .rx(rx)
  );

  // Connect tx back to rx for loopback
  assign rx = tx;
```

Figure 17: Top Module Test bench

```
// Test sequence
 initial begin
   $display("Starting transceiver testbench...");

   // Reset
   rstn = 0;
   sw = 4'd0;
   btn = 0;
   #50;
   rstn = 1;

   // Send known values instead of random ones
   for (int i = 0; i < 10; i++) begin
     @(posedge clk);
     sw = i % 10;  // Cycle through 0 to 9
     @(posedge clk);
     btn = 1;  // Simulate button press
     @(posedge clk);
     btn = 0;

     // Wait for UART transmission + reception
     #(CLOCKS_PER_PULSE * (BITS_PER_WORD + 2) * 10);  // Conservative wait

     $display("Sent: %0d, Segment: %b", sw, seg);
   end

   $finish;
 end

endmodule
```

Figure 18: Top Module Test bench

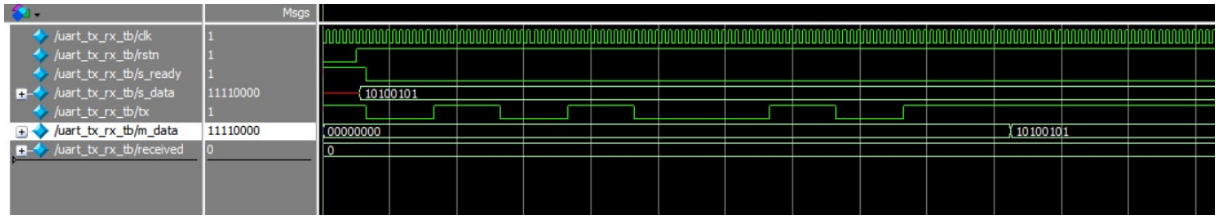

Figure 19: UART testing wave form

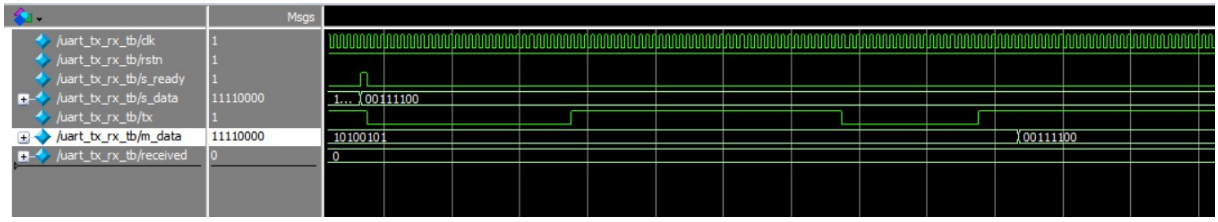

Figure 20: UART testing wave form

5 FPGA Implementation

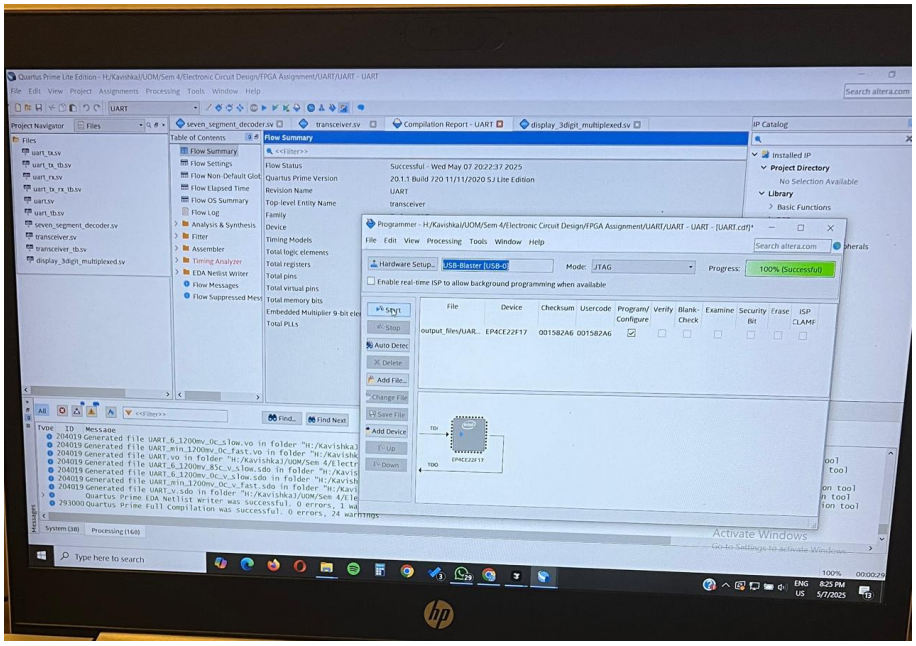

Figure 21: Code Uploaded Successfully. The completion message confirms that the UART implementation with 7-segment display driver was properly synthesized, placed, routed, and uploaded to the target FPGA device.

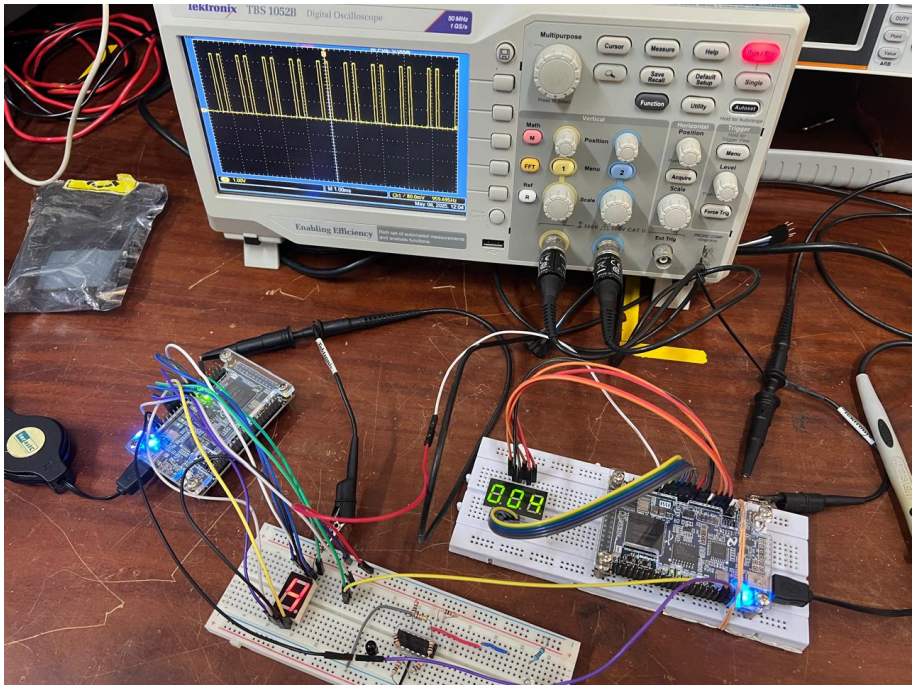

Figure 22: Physical lab setup showing the UART implementation in action. Two FPGA boards are interconnected via jumper wires to establish serial communication. The right(your) board is configured with our implementation, displaying received data on its 7-segment display, while the left board belongs to -the peer team. The setup demonstrates successful bidirectional UART communication between independent FPGA implementations.

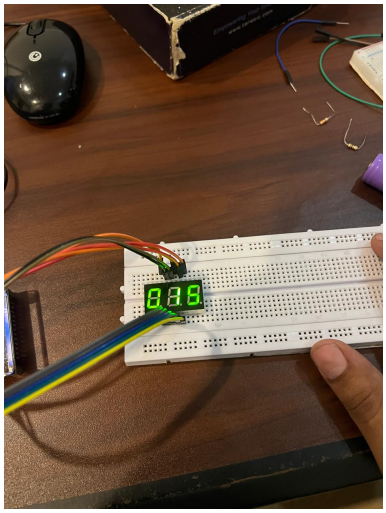

Figure 23: 3 Digit Seven Segment Dispaly Testing -Close-up view of the 3-digit 7-segment display showing the value received via UART communication. The multiplexed display correctly renders the decimal representation of the received 8-bit data. In this loopback test configuration, the FPGA board's tx pin is directly connected to its rx pin, demonstrating that data transmitted is correctly received and displayed.

6 Conclusion

The transceiver and 3-digit multiplexed display system have been successfully implemented and simulated. The system demonstrates the ability to send and receive data via UART and display the received data on a 3-digit 7-segment display. The design meets the required functionality and is ready for further refinement or integration into a larger system.

7 Future Work

Future improvements may include:

- **Extension**: Expand the system to transmit multiple words in a single transmission session.
- **Error Checking**: Implementing parity checking in UART transmission for error detection.
- **Display Expansion**: Expanding the display to support larger numbers or hexadecimal values.
- **Optimization**: Reducing the number of logic gates used in the multiplexing logic for improved resource utilization.