Department of Electronic and Telecommunication

University of Moratuwa

Group 07

- **200123H – DHANOMIKA A.P.N.**
- **200118X – DESHAPRIYA G.I.**
- **200117T – DESHAN K.A.G.D.**

S4-EN2111 - Electronic Circuit Design

UART Transceiver Implementation in FPGA

Transmitter

```
module transmitter #(
    parameter CLOCKS_PER_PULSE = 16
)
(
    input logic [7:0] data_in,
    input logic data_en,
    input logic clk,
    input logic rstn,
    output logic tx,
    output logic tx_busy
);
    enum {TX_IDLE, TX_START, TX_DATA, TX_END} state;

    logic[7:0] data = 8'b0;
    logic[2:0] c_bits = 3'b0;
    logic[$clog2(CLOCKS_PER_PULSE)-1:0] c_clocks = 0;

    always_ff @(posedge clk or negedge rstn) begin
       if (!rstn) begin
           c_clocks <= 0;
           c_bits <= 0;
           data <= 0;
              tx <= 1'b1;
           state <= TX_IDLE;
       end else begin
           case (state)
           TX_IDLE: begin
              if (~data_en) begin
                 state <= TX_START;
                 data <= data_in;
                 c_bits <= 3'b0;
                 c_clocks <= 0;
              end else tx <= 1'b1;
           end
           TX_START: begin
              if (c_clocks == CLOCKS_PER_PULSE-1) begin
                 state <= TX_DATA;
                 c_clocks <= 0;
              end else begin
                 tx <= 1'b0;
                 c_clocks <= c_clocks + 1;
              end
           end
           TX_DATA: begin
              if (c_clocks == CLOCKS_PER_PULSE-1) begin
                 c_clocks <= 0;
                 if (c_bits == 3'd7) begin
                    state <= TX_END;
                 end else begin
                    c_bits <= c_bits + 1;
                    tx <= data[c_bits];
                 end
              end else begin
                 tx <= data[c_bits];
                 c_clocks <= c_clocks + 1;
              end
           end
           TX_END: begin
              if (c_clocks == CLOCKS_PER_PULSE-1) begin
                 state <= TX_IDLE;
                 c_clocks <= 0;
              end else begin
                 tx <= 1'b1;
                 c_clocks <= c_clocks + 1;
              end
           end
              default: state <= TX_IDLE;
           endcase
       end
    end
    assign tx_busy = (state != TX_IDLE);

endmodule
```

The provided Verilog code implements a compact serial transmitter module. It utilizes an internal state machine to transmit data in a synchronized manner. The module supports an 8-bit data input, data enable signal, clock signal, and asynchronous reset. It operates in four states: IDLE, START, DATA, and END. Upon data transmission initiation, the module progresses through these states, transmitting one bit at a time synchronized with the clock signal. The transmitter's busy/idle status is indicated through an output signal. Overall, this code represents a concise implementation of a serial transmitter with controlled and synchronized data transmission.

Receiver

```
module receiver #(
    parameter CLOCKS_PER_PULSE = 16
)
(
    input logic clk,
    input logic rstn,
    input logic ready_clr,
    input logic rx,
    output logic ready,
    output logic [7:0] data_out
);

    enum {RX_IDLE, RX_START, RX_DATA, RX_END} state;

    logic[2:0] c_bits;
    logic[$clog2(CLOCKS_PER_PULSE)-1:0] c_clocks;

    logic[7:0] temp_data;
    logic rx_sync;

    always_ff @(posedge clk or negedge rstn) begin

       if (!rstn) begin
          c_clocks <= 0;
          c_bits <= 0;
          temp_data <= 8'b0;
          //data_out <= 8'b0;
          ready <= 0;
          state <= RX_IDLE;

       end else begin
          rx_sync <= rx;  // Synchronize the input sig

          case (state)
          RX_IDLE : begin
             if (rx_sync == 0) begin
                state <= RX_START;
                c_clocks <= 0;
             end
          end
          RX_START: begin
             if (c_clocks == CLOCKS_PER_PULSE/2-1) begin
                state <= RX_DATA;
                c_clocks <= 0;
             end else
                c_clocks <= c_clocks + 1;
          end
          RX_DATA : begin
             if (c_clocks == CLOCKS_PER_PULSE-1) begin
                c_clocks <= 0;
                temp_data[c_bits] <= rx_sync;
                if (c_bits == 3'd7) begin
                   state <= RX_END;
                   c_bits <= 0;
                end else c_bits <= c_bits + 1;
             end else c_clocks <= c_clocks + 1;
          end
          RX_END : begin
             if (c_clocks == CLOCKS_PER_PULSE-1) begin
                //data_out <= temp_data;
                ready <= 1'b1;
                state <= RX_IDLE;
                c_clocks <= 0;
             end else c_clocks <= c_clocks + 1;
          end
          default: state <= RX_IDLE;
          endcase
       end
    end
    assign data_out = temp_data;
endmodule
```

The given Verilog code represents a receiver module designed to receive data serially using a clock signal. It utilizes an internal state machine to control the reception process. The module synchronizes the incoming data signal, captures the data bit by bit, and stores it in a temporary register. Once all the bits are received, the module sets the ready signal and outputs the received data. The receiver transitions between states based on the clock cycles and resets to the idle state after data reception is complete. Overall, this code implements a basic serial receiver module that efficiently captures and outputs received data.

Testbench

```
`timescale 1ns/1ps

module testbench();

   localparam CLOCKS_PER_PULSE = 4;
  logic [3:0] data_in = 4'b0001;
   logic clk = 0;
   logic rstn = 0;
   logic enable = 1;

   logic tx_busy;
   logic ready;
   logic [3:0] data_out;
   logic [7:0] display_out;

   logic loopback;
   logic ready_clr = 1;

   uart #(.CLOCKS_PER_PULSE(CLOCKS_PER_PULSE))
         test_uart(.data_in(data_in),
                   .data_en(enable),
                   .clk(clk),
                   .tx(loopback),
                   .tx_busy(tx_busy),
                   .rx(loopback),
                   .ready(ready),
                   .ready_clr(ready_clr),
                   .led_out(data_out),
                   .display_out(display_out),
                   .rstn(rstn)
                   );


   always begin
      #1 clk = ~clk;
   end

   initial begin
      $dumpfile("testbench.vcd");
      $dumpvars(0, testbench);
```

```
      rstn <= 1;
    enable <= 1'b0;
      #2 rstn <= 0;
    #2 rstn <= 1;
    #5 enable <= 1'b1;
  end

  always @(posedge ready) begin

    if (data_out != data_in) begin
      $display("FAIL: rx data %x does not match tx %x"
      $finish();
    end else begin
       if (data_out == 4'b1111) begin //Check if recei
         $display("SUCCESS: all bytes verified");
         $finish();
       end

      #10 rstn <= 0;
      #2 rstn <= 1;
      data_in <= data_in + 1'b1;
      enable <= 1'b0;
      #2 enable <= 1'b1;
    end
  end
endmodule
```

The provided Verilog code represents a compact testbench for the UART module. It initializes the necessary signals and parameters for testing the UART functionality. The testbench includes a clock signal, asynchronous reset signal, data input signal, enable signal, and various output signals. It verifies the correctness of data transmission and reception by comparing the received data with the transmitted data. If any discrepancies are found, an error message is displayed, and the simulation is terminated. Upon successful verification of all data bytes, a success message is displayed. This testbench provides an efficient and thorough validation of the UART module's functionality.

Top Level Module

```
module uart #(
    parameter CLOCKS_PER_PULSE = 5208
)(
    input logic [3:0] data_in,
    input logic data_en,
    input logic clk,
    input logic rstn,
    output logic tx,
    output logic tx_busy,
    input logic ready_clr,
    input logic rx,
    output logic ready,
    output logic [3:0] led_out,
    output logic [6:0] display_out
);
    logic [7:0] data_input;
    logic [7:0] data_output;
    transmitter #(.CLOCKS_PER_PULSE(CLOCKS_PER_PULSE)) uart_tx (
       .data_in(data_input),
       .data_en(data_en),
       .clk(clk),
       .rstn(rstn),
       .tx(tx),
       .tx_busy(tx_busy)
    );
    receiver #(.CLOCKS_PER_PULSE(CLOCKS_PER_PULSE)) uart_rx (
       .clk(clk),
       .rstn(rstn),
       .ready_clr(ready_clr),
       .rx(rx),
       .ready(ready),
       .data_out(data_output)
    );
    binary_to_7seg converter (
       .data_in(data_output[3:0]),
       .data_out(display_out)
    );
    assign data_input = {4'b0, data_in};
    assign led_out = data_output[3:0];

endmodule
```

The provided Verilog code implements a concise UART module that facilitates bidirectional data communication. It includes a transmitter submodule, a receiver submodule, and a binary-to-7-segment converter. The module supports data transmission and reception using clock and reset signals, along with control and data inputs. It efficiently handles data transmission and reception while providing a display output for the received data. Overall, this compact UART module enables reliable and synchronized data communication in a minimal implementation.

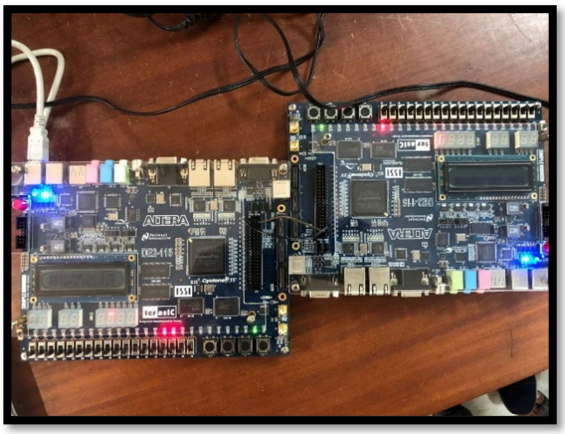
Binary to 7-segment converter

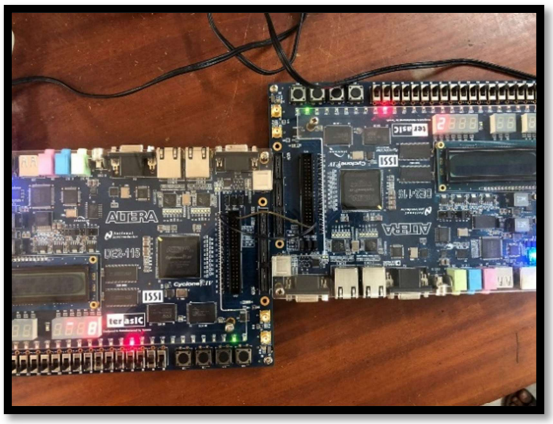
```

module binary_to_7seg (
    input logic [3:0] data_in,
    output logic [6:0] data_out
);
    // Make a LUT to convert digits to 7 segment output
    // Input - 4 bits, output - 7 bits
    logic [15:0][6:0] lut_7seg;

    // Output is gfedcba
    assign lut_7seg[0] = 7'b0111111;
    assign lut_7seg[1] = 7'b0000110;
    assign lut_7seg[2] = 7'b1011011;
    assign lut_7seg[3] = 7'b1001111;
    assign lut_7seg[4] = 7'b1100110;
    assign lut_7seg[5] = 7'b1101101;
    assign lut_7seg[6] = 7'b1111101;
    assign lut_7seg[7] = 7'b0000111;
    assign lut_7seg[8] = 7'b1111111;
    assign lut_7seg[9] = 7'b1101111;
    assign lut_7seg[15:10] = 7'b0;    // unused

    assign data_out = ~lut_7seg[data_in];

endmodule
```

The provided Verilog code showcases a compact binary-to-7-segment converter module. The module takes a 4-bit binary input and produces a 7-bit output representing the segments of a 7-segment display. It utilizes a look-up table (LUT) to map each possible input value to the corresponding 7-bit segment pattern. The LUT is initialized with assignments for values from 0 to 9, while unused values are set to zeros. By applying the complement operation to the LUT output, the module generates the appropriate 7-segment display pattern for the given binary input. This simple and efficient module enables the conversion of binary values to their visual representation on a 7-segment display.

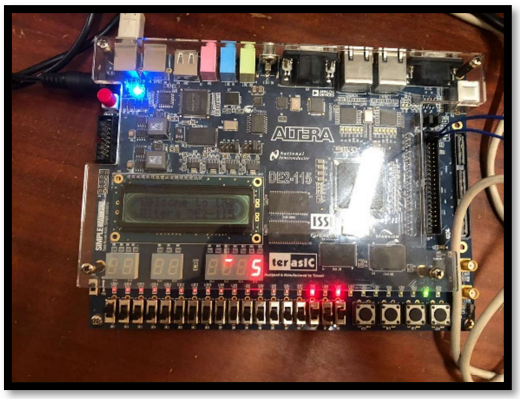
Results

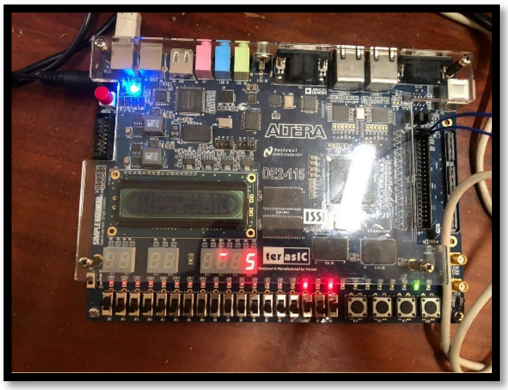
Test with another group





Loopback Test





Simulation result/ Timing Diagrams