Department of Electronics and Telecommunication Engineering
University of Moratuwa

EN2111 – Electronic Circuit Design

Implementation of UART protocol on a FPGA

Group No: 15

Dodangoda D.K.S.J	210150V
Dulnath W.H.R	210152E
EDIRISINGHE E.A.D.D.D.	210153H

UART Protocol

UART (Universal Asynchronous Receiver-Transmitter) is a serial communication protocol commonly used in embedded devices.

It operates asynchronously, meaning that both the transmitter and receiver must agree on parameters such as the Baud rate, number of data bits, parity bits, stop bits, and flow control.

In UART communication, the transmission line remains in a high state when communication is not active. When data transmission begins, the line voltage changes from high to low, signaling the start of communication. At this point, the receiver starts sampling the incoming data.

TX

a) 1 START bit
b) 5, 6, 7, or 8 data bits
c) 1 PARITY bit (optional)
d) 1, 1.5, or 2 STOP bits

RX

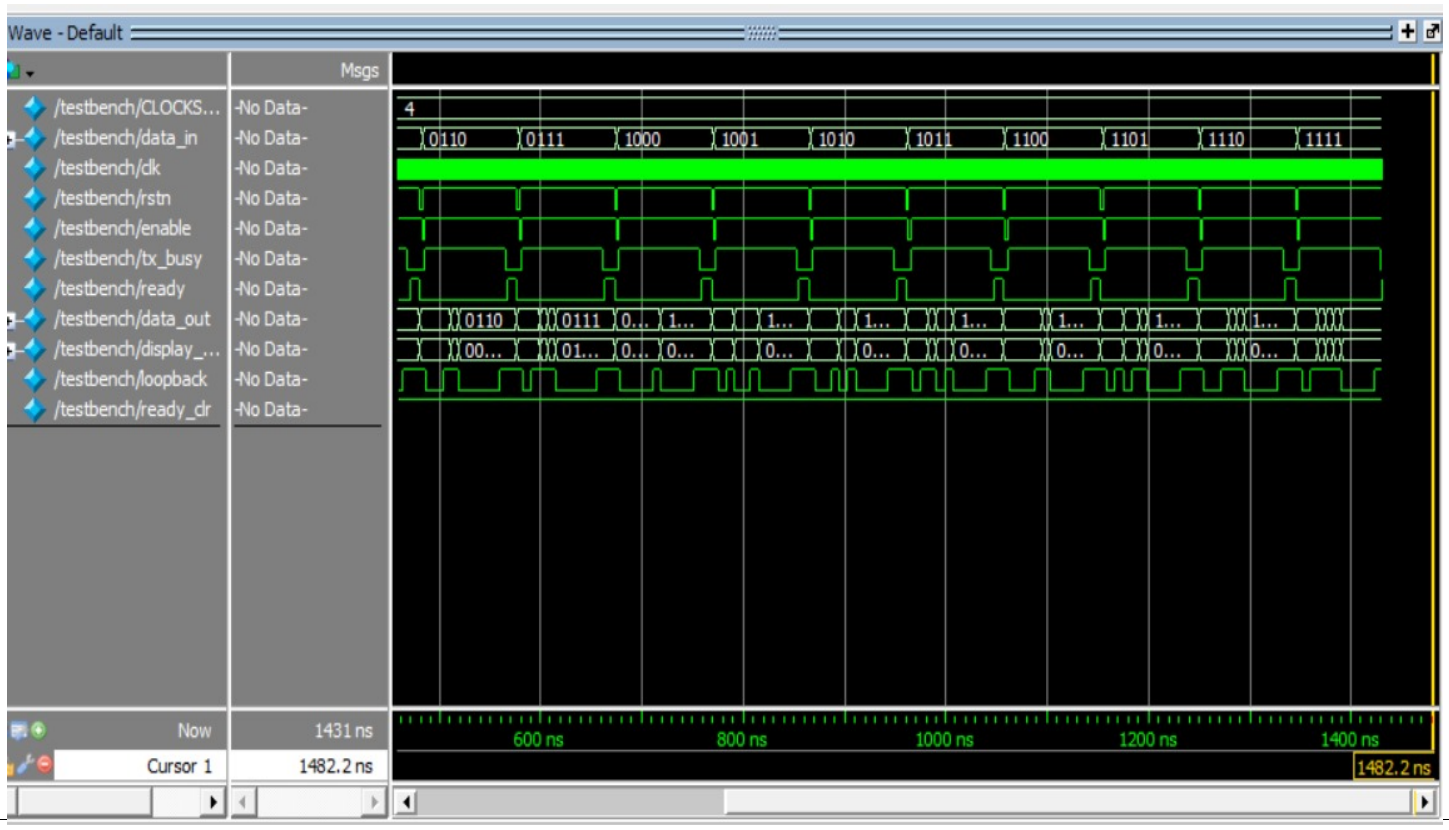
a) 1 START bit
b) 5, 6, 7, or 8 data bits
c) 1 PARITY bit (optional)
d) 1 STOP bit (any other STOP bits transferred with the above data are not detected)

Implementation on the FPGA

In the UART implementation, there are four main states: Idle, Start, Data, and Stop. When transmission begins, the receiver samples the data at a rate typically higher than the baud rate.

We started by implementing a test bench where a known bit stream was sent, and we checked whether it was received properly. Afterward, we implemented both the transmitter and receiver on the FPGA and verified that the transmission was functioning correctly in a self-loop configuration. Finally, we tested the functionality by connecting two separate FPGA boards, one set as a transmitter and the other as a receiver, and verified the transmission between them.

Results



Codes

Transmitter - *transmitter.sv*

```
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
```

Reciever - *receiver.sv*

```
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 signal using a ff
      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
```

Binary to 7-seg conversion - *binary_to_7seg.sv*

```
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
```

UART module – *uart.sv*

```
module uart #(
  parameter CLOCKS_PER_PULSE = 520)
(
  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
```

Test bench – *testbench.sv*

```
`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", data_out, data_in);
     $finish();
    end
else begin
     if (data_out == 4'b1111) begin
          //Check if received data is 11111111
        $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
```