DIGITAL SYSTEM DESIGN APPLICATIONS

(CRN: 11275)

THE REPORT OF PROJECT – 2

Faculty of Electrical and Electronics Engineering

Electronics and Communication Engineering

Yusuf Tekin – 040200043

Yusuf Tekin
040200043

1. Baud Rate Gen

```
`timescale 1ns / 1ps

module baud_gen #(parameter Rx_resolution = 16)(
    input clk,
    input rst,
    input BR_mode, // 0 -> 9600 and 1 -> 115200
    output reg clk_Tx,
    output reg clk_Rx
    );

    parameter clk_freq = 100000000; //100MHz
    localparam DIV_9600 = (clk_freq / 9600) -1;
    localparam DIV_9600_Rx = (clk_freq / (9600 * Rx_resolution)) -1;
    localparam DIV_115200 = (clk_freq / 115200) -1;
    localparam DIV_115200_Rx = (clk_freq / (115200 * Rx_resolution)) -1;

    reg [15:0] baud_count = 0;
    reg [15:0] baud_count_Rx = 0;
    reg [15:0] baud_limit;
    reg [15:0] baud_limit_Rx;

    always @(posedge clk) begin
        if(BR_mode) begin
            baud_limit <= DIV_115200;
            baud_limit_Rx <= DIV_115200_Rx;
        end else begin
            baud_limit <= DIV_9600;
            baud_limit_Rx <= DIV_9600_Rx;
        end
    end
    always @(posedge clk or posedge rst) begin
        if(rst) begin
            baud_count <= 0;
            clk_Tx <= 1'b0;
        end else begin
            if(baud_count < baud_limit) begin
                baud_count <= baud_count +1;
                clk_Tx <= 1'b0;
            end else begin
                baud_count <= 0;
                clk_Tx <= 1'b1;
            end
        end
    end

    always @(posedge clk or posedge rst) begin
        if(rst) begin
            baud_count_Rx <= 0;
            clk_Rx <= 1'b0;
        end else begin
            if(baud_count_Rx < baud_limit_Rx) begin
                baud_count_Rx <= baud_count_Rx +1;
                clk_Rx <= 1'b0;
            end else begin
                baud_count_Rx <= 0;
                clk_Rx <= 1'b1;
            end
        end
    end
endmodule
```

Figure 1 - Baud Rate Gen Verilog Code

Yusuf Tekin
040200043

```
`timescale 1ns / 1ps

module baud_gen_tb();

    wire clk_Rx, clk_Tx;
    reg rst, clk, BR_mode;

    baud_gen #(.Rx_resolution(16)) uut(
        .clk(clk),
        .rst(rst),
        .BR_mode(BR_mode),
        .clk_Tx(clk_Tx),
        .clk_Rx(clk_Rx)
        );

    always #5 clk = ~clk;

    initial begin
        clk = 1'b0; rst = 1'b1; BR_mode = 1'b0;
        #10; rst = 1'b0;
        #1000000;
        rst = 1'b1; BR_mode = 1'b1;
        #10; rst = 1'b0;
        #1000000;
        $finish();
    end
endmodule
```

Figure 2 - Baud Rate Gen Testbench Code

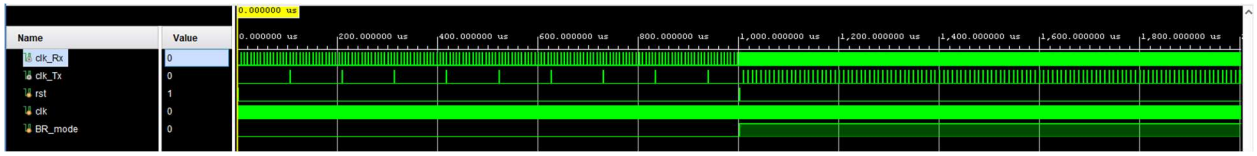

Figure 3 - Baud Rate Gen Behavioral Simulation

In this baud rate generator design the resolution of the receiver of the UART is parametric and chosen to be 16 for the continuation of the project. To be able to switch between the 9600 Hz and 115200 Hz baud rate clocks, there is a BR_mode input implemented in the design. For this 1-bit input “0” chooses 9600 Hz and 1 chooses 115200 Hz. In the behavioral simulation it can be seen that the clocks are working as intended with the 1/16 ratio.

Yusuf Tekin
040200043

2. Transmitter Unit

```
`timescale 1ns / 1ps
module uart_tx(
    input clk_Tx,
    input clk,
    input rst,
    input Tx_enable,
    input [7:0] Tx_data_in,
    output reg Tx_data_out,
    output reg start_flag,
    output reg busy,
    output reg done
    );

    reg [3:0] bit_index = 4'b0;
    reg [7:0] data_buffer = 8'b0;
    reg [1:0] state = 2'b00;
    parameter IDLE  = 2'b00, START = 2'b01,
              DATA  = 2'b10, STOP  = 2'b11;

    always @(posedge clk or posedge rst) begin
        if(rst) begin
            state <= IDLE;
            Tx_data_out <= 1'b1;
            start_flag <= 1'b0;
            busy <= 1'b0;
            done <= 1'b0;
            bit_index <= 4'b0;
        end else if(clk_Tx) begin
            case(state)
                IDLE: begin
                    Tx_data_out <= 1'b1;
                    busy <= 1'b0;
                    done <= 1'b0;
                    bit_index <= 4'b0;
                    if(Tx_enable) begin
                        state <= START;
                        data_buffer <= Tx_data_in;
                        Tx_data_out <= 1'b0;
                        start_flag <= 1'b1;
                    end else begin
                        state <= IDLE;
                    end
                end
                START: begin
                    Tx_data_out <= data_buffer[bit_index];
                    bit_index <= bit_index + 1;
                    state <= DATA;
                    start_flag <= 1'b0;
                    busy <= 1'b1;
                end
                DATA: begin
                    start_flag <= 1'b0;
                    if(bit_index < 8) begin
                        Tx_data_out <= data_buffer[bit_index];
                        bit_index <= bit_index + 1;
                        state <= DATA;
                    end else begin
                        bit_index <= 0;
                        busy <= 1'b0;
                        done <= 1'b1;
                        state <= STOP;
                    end
                end
                STOP: begin
                    Tx_data_out <= 1'b1;
                    data_buffer <= 8'b0;
                    state <= IDLE;
                end
                default: begin
                    state <= IDLE;
                end
            endcase
        end
    end
endmodule
```

Figure 4 - Transmitter Unit Verilog Code

```
`timescale 1ns / 1ps
module uart_tx_tb();

    wire done, busy, start_flag, Tx_data_out;
    reg clk, clk_Tx, rst, enable;
    reg [7:0] Tx_data_in;

    reg [15:0] baud_count = 0;
    reg [15:0] baud_limit;

    localparam DIV_9600 = (100000000 / 9600) -1;

    uart_tx uut(
        .clk_Tx(clk_Tx),
        .clk(clk),
        .rst(rst),
        .Tx_enable(enable),
        .Tx_data_in(Tx_data_in),
        .Tx_data_out(Tx_data_out),
        .start_flag(start_flag),
        .busy(busy),
        .done(done)
        );

    always #5 clk = ~clk;

    always @(posedge clk or posedge rst) begin
        if(rst) begin
            baud_count <= 0;
            clk_Tx <= 1'b0;
            baud_limit <= DIV_9600;
        end else begin
            if(baud_count < baud_limit) begin
                baud_count <= baud_count +1;
                clk_Tx <= 1'b0;
            end else begin
                baud_count <= 0;
                clk_Tx <= 1'b1;
            end
        end
    end

    initial begin
        clk = 1'b0; rst = 1'b1; #10;
        rst = 1'b0; enable = 1'b1; #1000;
        Tx_data_in = 8'b1100_0011;
        #1000000;
        Tx_data_in = 8'b1111_1111;
        #1000000;
        Tx_data_in = 8'b0000_0000;
        #1000000;
         Tx_data_in = 8'b1010_1010;
        #1000000;
        enable = 1'b0;
        $finish();
        end
endmodule
```

Figure 5 - Transmitter Unit Testbench Code

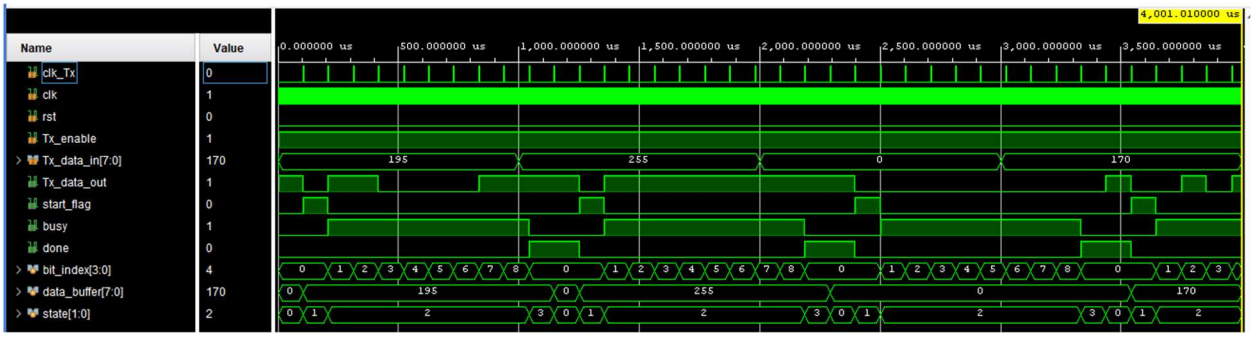

Figure 6 - Transmitter Behavioral Simulation

The transmitter unit uses an input which is determined in the baud rate generator as a clock for the state transitions. At standby, the transmitter stays at the IDLE state until both Tx_enable goes to logic-1 and the start bit goes to logic-0 while giving logic-1s to the output. When these two conditions are met at the next baud rated clock cycle the state machine operates in the START state which makes the start flag go high for a baud rated time. After the next baud rated clock hits high the 8-bit data already saved in the buffer starts to be given as the output in order (from least to most significant bit) and the busy signal goes high in the DATA state. At last, only the done signal goes high, and the output starts to give 1s until the state goes to IDLE and waits for the input again.

Yusuf Tekin
040200043

3. Receiver Unit

```
`timescale 1ns / 1ps
module uart_rx (
    input clk,
    input rst,
    input clk_Rx,
    input Rx_data_in,
    input Rx_enable,
    output reg [7:0] Rx_data_out,
    output reg Rx_start,
    output reg Rx_busy,
    output reg Rx_done
    );

    parameter IDLE  = 2'b00, START = 2'b01,
              DATA  = 2'b10, STOP  = 2'b11;

    reg [1:0] state = 2'b00;
    reg [3:0] sample_count;
    reg [3:0] bit_index;
    reg [7:0] data_buffer;
    reg [3:0] bit_weight = 0;
    reg error = 0;

     always @(posedge clk or posedge rst) begin
        if(rst) begin
            state <= IDLE;
            Rx_start <= 1'b0;
            Rx_busy <= 1'b0;
            Rx_done <= 1'b0;
            Rx_data_out <= 8'b0;
            sample_count <= 4'b0;
            bit_index <= 4'b0;
            data_buffer <= 8'b0;
            bit_weight <= 4'b0;
            error <= 1'b0;
        end else if(clk_Rx) begin
            /*if(error) begin
                state <= IDLE;
            end*/
            case(state)
                IDLE: begin
                    Rx_busy <= 1'b0;
                    sample_count <= 4'b0;
                    bit_index <= 4'b0;
                    error <= 1'b0;
                    bit_weight <= 4'b0;
                    if(Rx_enable && ~Rx_data_in) begin
                        state <= START;
                        Rx_start <= 1'b1;
                    end else begin
                        state <= IDLE;
                    end
                end
                START: begin
                    if(sample_count < 7) begin
                        sample_count <= sample_count + 1;
                        bit_weight <= bit_weight + Rx_data_in;
                    end else begin
                        sample_count <= 0;
                        if(bit_weight < 4) begin  //If mostly zeros
                            sample_count <= 0;
                            state <= DATA;
                            Rx_start <= 1'b0;
                            Rx_busy <= 1'b1;
                            bit_weight <= 4'b0;
                        end else begin            // If not mostly zeros
                            state <= IDLE;
                            error <= 1'b1;
                            Rx_start <= 1'b0;
                        end
                    end
                end
                DATA: begin
                    if(sample_count < 7) begin
                        sample_count <= sample_count + 1;
                        bit_weight <= bit_weight + Rx_data_in;
                    end else begin
                        sample_count <= 0;
                        bit_weight <= 4'b0;
                        if(bit_index < 8) begin
                            bit_index <= bit_index + 1;
                            if(bit_weight < 4) begin            //Majority Low: bit = 0
                                data_buffer[bit_index] <= 1'b0;
                            end else begin                      //Majority High: bit = 1
                                data_buffer[bit_index] <= 1'b1;
                            end
                        end else begin
                            bit_index <= 4'b0;
                            state <= STOP;
                            Rx_busy <= 1'b0;
                        end
                    end

                end
                STOP: begin
                    if (sample_count < 7) begin
                        sample_count <= sample_count + 1;
                        bit_weight <= bit_weight + Rx_data_in;
                    end else begin
                        sample_count <= 0;
                        bit_weight <= 4'b0;
                        if (bit_weight > 3) begin // Valid stop bit (majority high)
                            Rx_data_out <= data_buffer; // Output received data
                            Rx_done <= 1'b1;
                            state <= IDLE;
                        end else begin
                            error <= 1'b1; // Framing error
                            state <= IDLE;
                        end
                    end
                end
            endcase
        end
     end
endmodule
```

Figure 7 - Receiver Unit Verilog Code

Yusuf Tekin
040200043

```
`timescale 1ns / 1ns
module uart_rx_tb();

    // Parameters for 9600 baud, 8x oversampling
    localparam DIV_9600_Rx = (100000000 / (9600 * 8)) - 1;

    reg [15:0] baud_count_Rx = 0;
    reg [15:0] baud_limit_Rx;

    reg clk, rst, clk_Rx, Rx_data_in, Rx_enable;

    wire [7:0] Rx_data_out;
    wire Rx_start, Rx_busy, Rx_done;

    uart_rx uut (
        .clk(clk),
        .rst(rst),
        .clk_Rx(clk_Rx),
        .Rx_data_in(Rx_data_in),
        .Rx_enable(Rx_enable),
        .Rx_data_out(Rx_data_out),
        .Rx_start(Rx_start),
        .Rx_busy(Rx_busy),
        .Rx_done(Rx_done)
    );


    always #5 clk = ~clk;

    // Baud clock generation: 8x baud clock for 9600 baud rate
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            baud_count_Rx <= 0;
            clk_Rx <= 1'b0;
            baud_limit_Rx <= DIV_9600_Rx;
        end else begin
            if (baud_count_Rx < baud_limit_Rx) begin
                baud_count_Rx <= baud_count_Rx + 1;
                clk_Rx <= 1'b0;
            end else begin
                baud_count_Rx <= 0;
                clk_Rx <= 1'b1;
            end
        end
    end

    always @(*) begin
        if(Rx_done == 1'b1) begin
            if(Rx_data_out == 8'b1110_0001) begin
                $display("Frame 1: Received data is correct = %b", Rx_data_out);
            end else if(Rx_data_out == 8'b1010_1010) begin
                $display("Frame 2: Received data is correct = %b", Rx_data_out);
            end else if(Rx_data_out == 8'b0000_1111) begin
                $display("Frame 3: Received data is correct = %b", Rx_data_out);
            end else if(Rx_data_out == 8'b1111_1111) begin
                $display("Frame 4: Received data is correct = %b", Rx_data_out);
            end else begin
                $display("Received data is not correct = %b", Rx_data_out);
            end
        end
    end


    initial begin

        clk = 1'b0;
        rst = 1'b1;
        Rx_data_in = 1'b1;
        #37;
        rst = 1'b0;
        Rx_enable = 1'b1;
        #3500;

        // Frame 1: 8'b11100001 (Start bit = 0, Data = 11100001, Stop bit = 1)
        Rx_data_in = 1'b0; #104167; // Start bit
        Rx_data_in = 1'b1; #104167; // Bit 0
        Rx_data_in = 1'b0; #104167; // Bit 1
        Rx_data_in = 1'b0; #104167; // Bit 2
        Rx_data_in = 1'b0; #104167; // Bit 3
        Rx_data_in = 1'b0; #104167; // Bit 4
        Rx_data_in = 1'b1; #104167; // Bit 5
        Rx_data_in = 1'b1; #104167; // Bit 6
        Rx_data_in = 1'b1; #104167; // Bit 7
        Rx_data_in = 1'b1; #104167; // Stop bit

        Rx_data_in = 1'b1; #104167;
        Rx_data_in = 1'b1; #104167;
        Rx_data_in = 1'b1; #104167;
        Rx_data_in = 1'b1; #104167;

        // Frame 2: 8'b10101010 (Start bit = 0, Data = 10101010, Stop bit = 1)
        Rx_data_in = 1'b0; #104167; // Start bit
        Rx_data_in = 1'b0; #104167; // Bit 0
        Rx_data_in = 1'b1; #104167; // Bit 1
        Rx_data_in = 1'b0; #104167; // Bit 2
        Rx_data_in = 1'b1; #104167; // Bit 3
        Rx_data_in = 1'b0; #104167; // Bit 4
        Rx_data_in = 1'b1; #104167; // Bit 5
        Rx_data_in = 1'b0; #104167; // Bit 6
        Rx_data_in = 1'b1; #104167; // Bit 7
        Rx_data_in = 1'b1; #104167; // Stop bit

        Rx_data_in = 1'b1; #104167;
        Rx_data_in = 1'b1; #104167;
        Rx_data_in = 1'b1; #104167;
        Rx_data_in = 1'b1; #104167;

        // Frame 3: 8'b00001111 (Start bit = 0, Data = 00001111, Stop bit = 1)
        Rx_data_in = 1'b0; #104167; // Start bit
        Rx_data_in = 1'b1; #104167; // Bit 0
        Rx_data_in = 1'b1; #104167; // Bit 1
        Rx_data_in = 1'b1; #104167; // Bit 2
        Rx_data_in = 1'b1; #104167; // Bit 3
        Rx_data_in = 1'b0; #104167; // Bit 4
        Rx_data_in = 1'b0; #104167; // Bit 5
        Rx_data_in = 1'b0; #104167; // Bit 6
        Rx_data_in = 1'b0; #104167; // Bit 7
        Rx_data_in = 1'b1; #104167; // Stop bit

        Rx_data_in = 1'b1; #104167;
        Rx_data_in = 1'b1; #104167;
        Rx_data_in = 1'b1; #104167;
        Rx_data_in = 1'b1; #104167;

        // Frame 4: 8'b11111111 (Start bit = 0, Data = 00001111, Stop bit = 1)
        Rx_data_in = 1'b0; #104167; // Start bit
        Rx_data_in = 1'b1; #104167; // Bit 0
        Rx_data_in = 1'b1; #104167; // Bit 1
        Rx_data_in = 1'b1; #104167; // Bit 2
        Rx_data_in = 1'b1; #104167; // Bit 3
        Rx_data_in = 1'b1; #104167; // Bit 4
        Rx_data_in = 1'b1; #104167; // Bit 5
        Rx_data_in = 1'b1; #104167; // Bit 6
        Rx_data_in = 1'b1; #104167; // Bit 7
        Rx_data_in = 1'b1; #104167; // Stop bit

        Rx_data_in = 1'b1; #104167;
        Rx_data_in = 1'b1; #104167;
        Rx_data_in = 1'b1; #104167;
        Rx_data_in = 1'b1; #104167;
        $finish();
    end
endmodule
```

Figure 8 - Receiver Unit Testbench Code

```
run all
Frame 1: Received data is correct = 11100001
Frame 2: Received data is correct = 10101010
Frame 3: Received data is correct = 00001111
Frame 4: Received data is correct = 11111111
$finish called at time : 5834362 ns : File "C:/Users/jsphtkn/Vivado Projects/sstu_project_2_UART/sstu_project_2_UART.srcs/sim_1/new/uart_rx_tb.v" Line 151
```

Figure 9 - Receiver Unit TCL Console Output

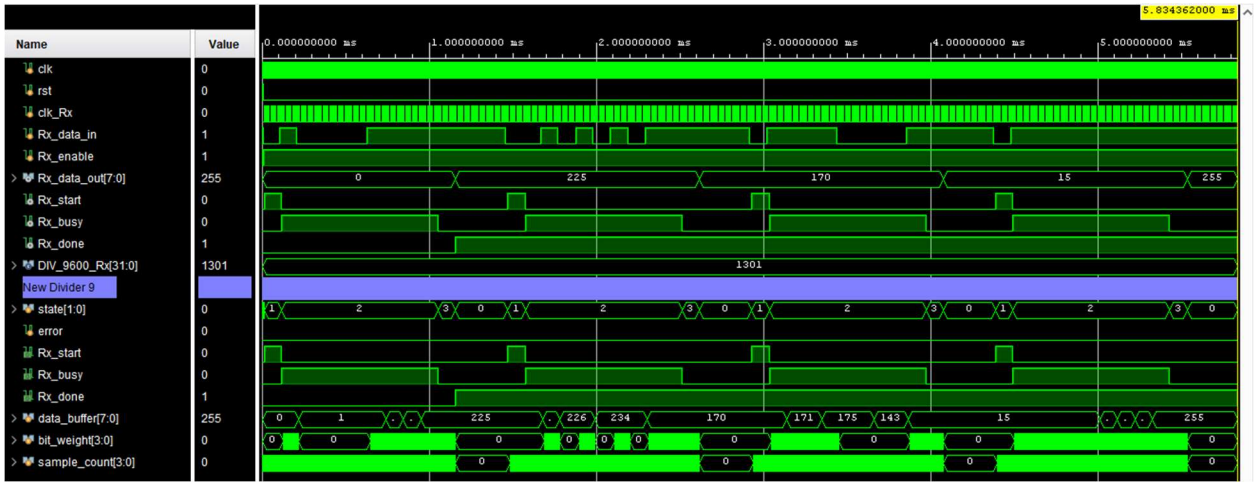

Figure 10 - Receiver Unit Behavioral Simulation

The receiver unit has a more complicated structure than the transmitter. A buffer is defined likewise however, the purpose of the buffer is to hold the 8-bit inputs one by one while shifting until the done signal goes high. To calculate the resolution of the input signals to determine whether the start or stop bits are correct, bit weight calculations are done. With the help of the error register, accuracy of the output signals can be calculated. In the design, the x8 baud calculations are done by using a sampling counter and the bit weight calculations. Since the resolution for each bit is 8, whether the bit weight is higher or lower than 4 is what determines the detected logic. The output and done signals are chosen to be held until the next start bit to make the observations easier. Furthermore, the testbench is designed to display TCL Console feedback for whether the output values are matching with the given inputs (Figure 9).

4. Top Module

```
`timescale 1ns / 1ps

module uart_top(
    input clk,
    input rst,
    input [7:0] data_in,
    input Tx_en,
    input Rx_en,
    input BR_mode,
    output [7:0] data_out
    );

    wire signal, clk_Rx, clk_Tx;

    uart_rx receiver_1(
        .clk(clk),
        .rst(rst),
        .clk_Rx(clk_Rx),
        .Rx_data_in(signal),
        .Rx_enable(Rx_en),
        .Rx_data_out(data_out),
        .Rx_start(),
        .Rx_busy(),
        .Rx_done());

    uart_tx transmitter_1(
        .clk_Tx(clk_Tx),
        .clk(clk),
        .rst(rst),
        .Tx_enable(Tx_en),
        .Tx_data_in(data_in),
        .Tx_data_out(signal),
        .start_flag(),
        .busy(),
        .done()
        );

    baud_gen BR_generator(
        .clk(clk),
        .rst(rst),
        .BR_mode(BR_mode),
        .clk_Tx(clk_Tx),
        .clk_Rx(clk_Rx)
        );

endmodule
```

Figure 11 - Top Module Verilog Code

```
`timescale 1ns / 1ns

module uart_top_tb();

wire [7:0] data_out;
reg rst, clk, Tx_en, Rx_en;
reg [7:0] data_in;
reg [1:0] BR_mode; // 0 -> 9600 and 1 -> 115200

uart_top uut(
    .clk(clk),
    .rst(rst),
    .data_in(data_in),
    .Tx_en(Tx_en),
    .Rx_en(Rx_en),
    .BR_mode(BR_mode),
    .data_out(data_out)
    );

    always #5 clk = ~clk;

    initial begin
        clk = 1'b0; BR_mode = 1'b0;  //9600
        Tx_en = 1'b0; Rx_en = 1'b0;
        data_in = 8'b0000_0000;
        rst = 1'b1; #10; rst = 1'b0;
        Tx_en = 1'b1; Rx_en = 1'b1;

        data_in = 8'b1100_0011; Tx_en = 1'b1;
        #1000000;
        data_in = 8'b1111_1111; Tx_en = 1'b1;
        #1100000;
        data_in = 8'b0000_0000; Tx_en = 1'b1;
        #1200000;
        data_in = 8'b1010_1010; Tx_en = 1'b1;
        #750000; Tx_en = 1'b0;
        #1500000;

        BR_mode = 1'b1;  //115200
        Tx_en = 1'b0; Rx_en = 1'b0;
        data_in = 8'b0000_0000;
        rst = 1'b1; #10; rst = 1'b0;
        Tx_en = 1'b1; Rx_en = 1'b1;

        data_in = 8'b1100_0011;
        #100000;
        data_in = 8'b1111_1111;
        #150000;
        data_in = 8'b0000_0000;
        #160000;
        data_in = 8'b1010_1010;
        #200000;
        Tx_en = 1'b0; Rx_en = 1'b0;
        #10000;
        $finish();
    end
endmodule
```

Figure 12 - Top Module Texstbench Code

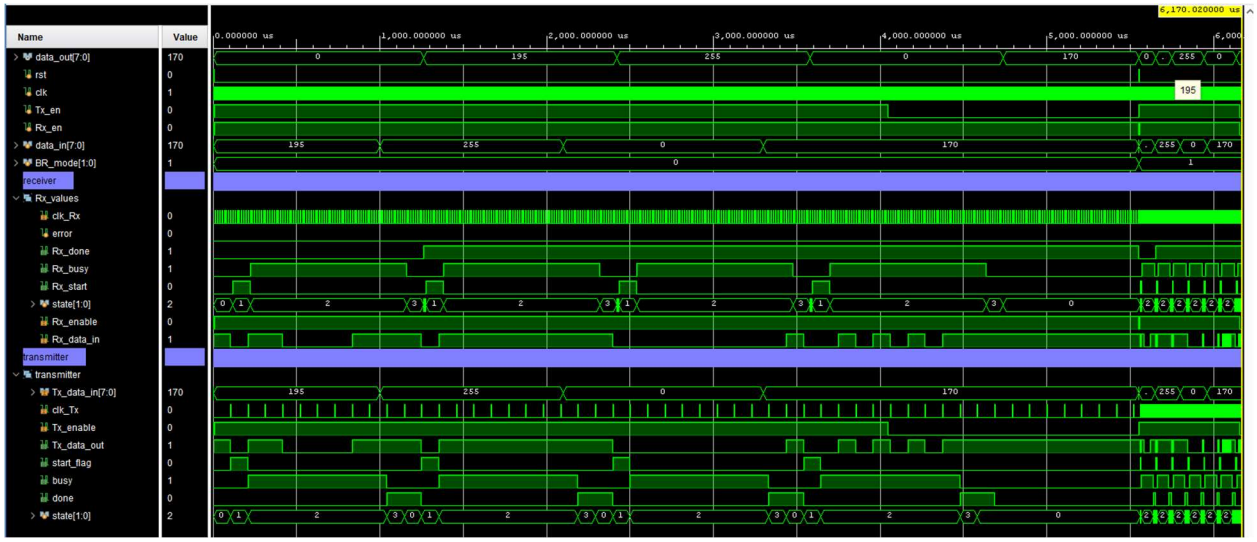

Figure 13 - Top Module Behavioral Simulation

The top module combines both transmitter and receiver with the addition of the baud generator. In the simulation (Figure 13) the inputs (195, 225, 0, 170) are given to the transmitter and expected to be obtained from receiver. The simulation switches the baud rates at the proper moment to test it out. In this testbench the output results are correct with both baud rates. However, if the delays between the inputs are given shorter, the receiver's error register goes high, and the results are corrupted. Since the delay values are chosen long enough, the different phases chosen for the input delays do not affect the system. Which is what a asynchronous system should operate.

Resource	Utilization	Available	Utilization %
LUT	85	63400	0.13
FF	80	126800	0.06
IO	21	210	10.00

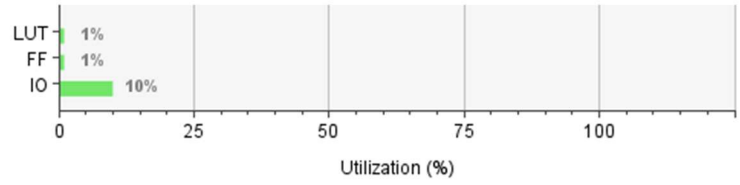

Figure 14 - Utilization Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.584 ns	Worst Hold Slack (WHS):	0.187 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	106	Total Number of Endpoints:	106	Total Number of Endpoints:	81

All user specified timing constraints are met.

Figure 15 - Timing Summary

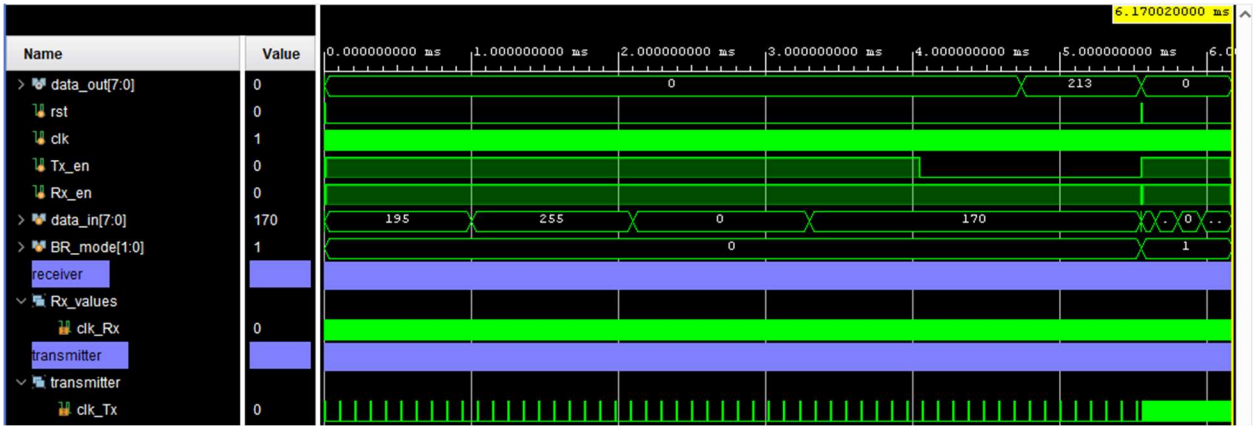

Figure 16 - Post Implementation Timing Simulation

5. Explaining UART

UART (Universal Asynchronous Receiver-Transmitter) is a asynchronous serial communication device which the data format and transmission speeds are configurable. It has two main parts which are transmitter and receiver. The transmitter has multiple bits of input and a 1-bit output. It sends data bits one by one from the transmitter and receives one by one from the receiver. After receiving the data, the receiver stores that data in a buffer until the stop bit arrives. Just after receiving the stop bit, the output of the receiver reflects the buffer values. In this design 1-bit start bit, 8-bit data bits and 1-bit stop bit are used.

The transmission starts operating when both the enable is high, and the start bit comes. After sending 8 bits of data, it stops after the stop bit. The same system goes with the receiver, it starts with the start bit and stores data with 8 data bits until the stop bit occurs. This system doesn't require a common clock to operate, however, both the transmitter and the receiver must operate at the same baud rate. When the receiver stores the data, the receiver samples the incoming signal at 8x the baud rate during each bit period. It determines the value of each bit by taking multiple samples and choosing the majority logic level.

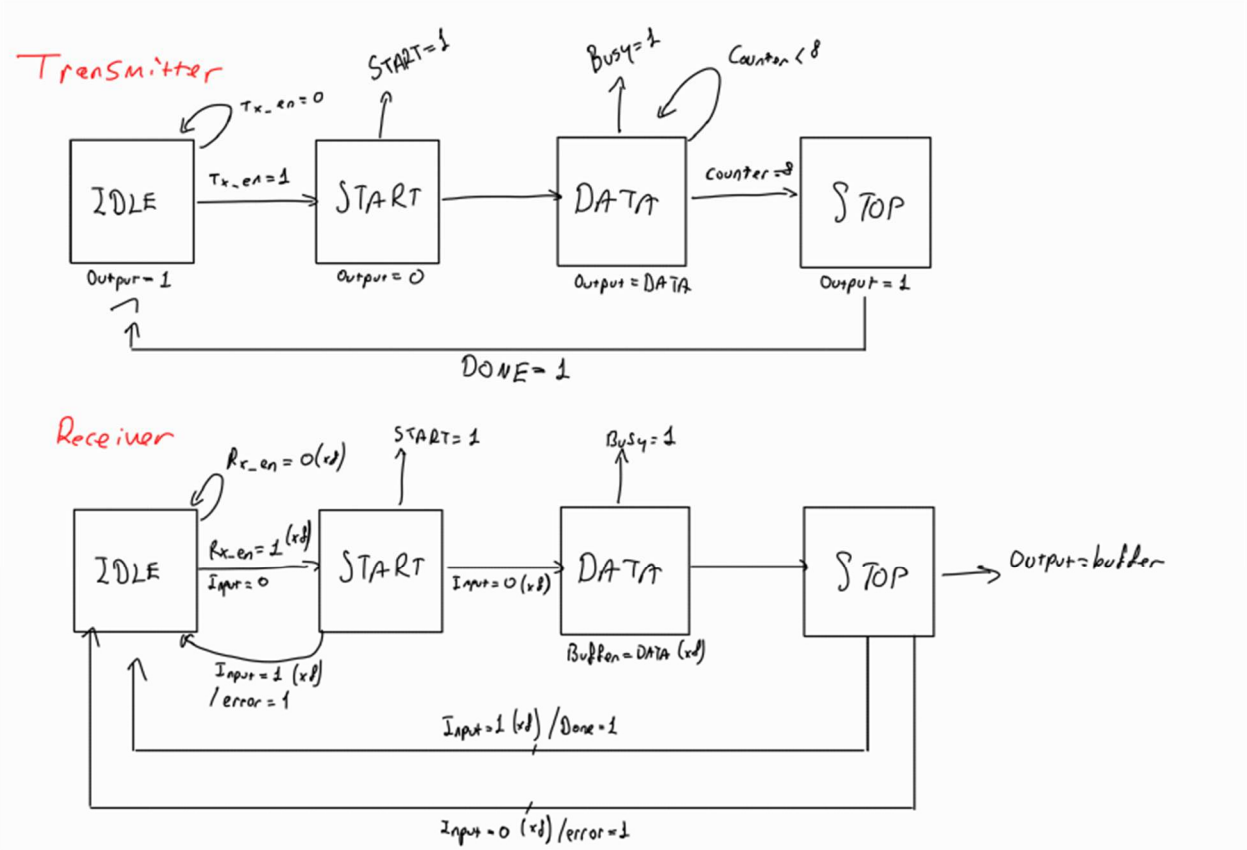

Figure 17 - Transmitter and Receiver State Diagram

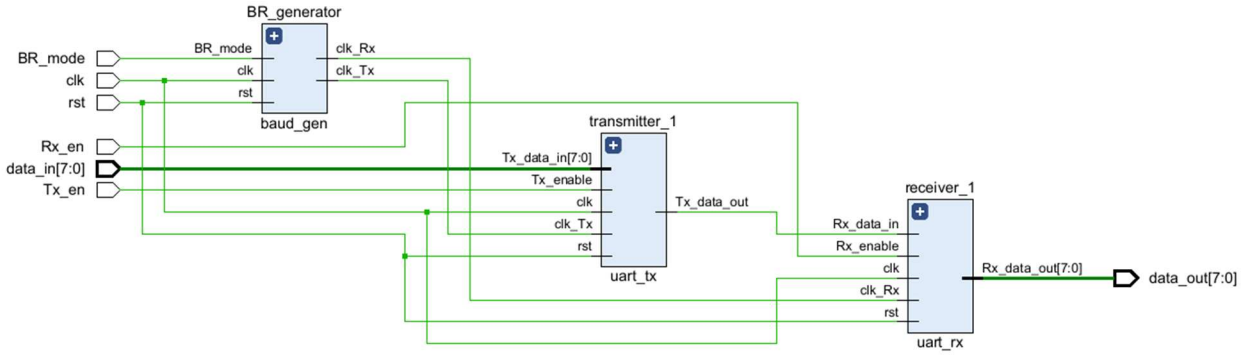

Figure 18 - Block Diagram of the Top Design

References

- “KeyStone Architecture Universal Asynchronous Receiver/Transmitter (UART) User Guide.” Available: https://www.ti.com/lit/ug/sprugp1/sprugp1.pdf
- “Universal asynchronous receiver-transmitter,” *Wikipedia*. https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter
- E. Pena and M. G. Legaspi, “UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter | Analog Devices.” https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html