2025-2026 FALL TERM

Digital System Design Applications Project 2

AHMET UTKU ALKAN
040220116

1- EXPLANATION PART

UART Protocol and Asynchronous Communication Logic

UART is one of the most fundamental serial communication protocols that enables data transfer between computer systems and microcontrollers. It is called "asynchronous" because there is no common clock line between the transmitter and receiver that determines when the data will be read. Instead, both sides agree in advance on the Baud Rate (bits per second) which determines the communication speed. Data transmission occurs by sequentially stringing bits over a single line (TX). When the line is idle (IDLE), the signal level is continuously maintained at logic-1 (High). This indicates that the line is not broken and a new data packet is ready to arrive. Communication begins when the transmitter pulls the line to logic-0 (Low), i.e., sends a Start Bit, allowing the receiver to wake up and begin timing the incoming data.

Transmitter (TX): Serializing Data

The main task of the Transmitter module is to send 8-bit parallel data (bytes) from the processor or system sequentially over a single line. This process is managed by a Finite State Machine (FSM). The system is initially in the IDLE state, and the line is at level '1'. When the send command (tx_en) arrives, the FSM switches to the START state and pulls the line to '0' for one bit; this is a "get ready, data is coming" alert to the receiver. Then it switches to the DATA state. Here, the available 8 bits of data are usually sent to the line sequentially from LSB (Least Significant Bit) to MSB (Most Significant Bit). The ticks from the Baud Generator determine exactly how long each bit will remain on the line. After all bits have been sent, it switches to the STOP state, and the line is pulled back to level '1', terminating the packet. The transmitter produces a busy signal until this process is complete.

Receiver (RX): Sampling and Capture

The receiver module is the most complex part of the UART because it doesn't know when data will arrive and needs to be constantly on alert. While the module is in the IDLE state, it constantly listens to the line. The system switches to START mode as soon as it detects the line dropping from '1' to '0' (Start Bit), and starts its internal counter. This is where oversampling comes into play. For example, if the system is operating in 8x mode, the receiver waits for the 4th tick to find the middle of each bit. This ensures that the data is read from the cleanest midpoint, avoiding unstable regions at the edges of the signal. After the start bit is verified, the system switches to DATA mode, and every 8 ticks (when the bit expires) the value on the line is read and stored in a shift register. When all 8 bits are complete, the system switches to STOP mode, the stop bit is verified as '1', and the serially received data is converted to parallel and presented to the system (rx_done).

STATE DIAGRAMS

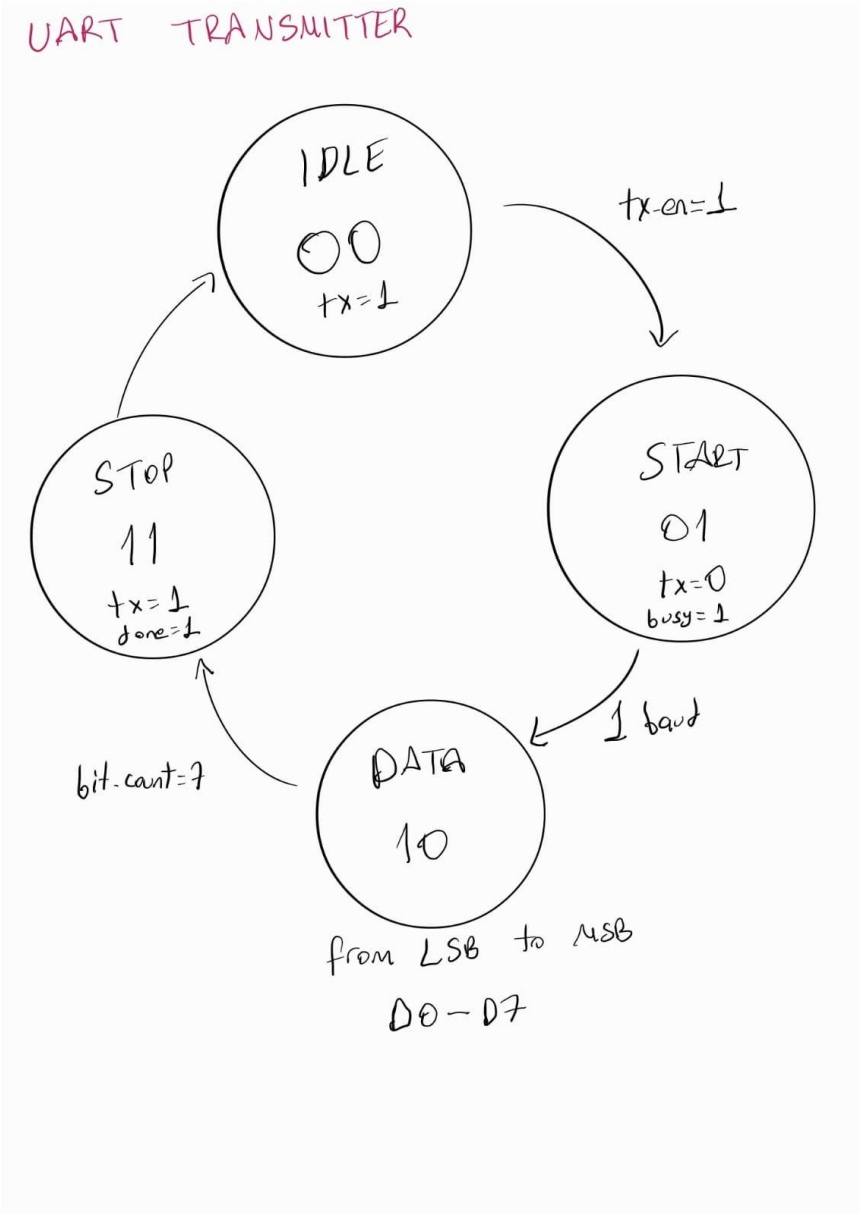

Figure 1.1: State Diagram of UART Transmitter

Examining the status diagram showing the control logic of the UART transmitter module, it can be seen that the system operates within four basic state cycles: IDLE (00), START (01), DATA (10), and STOP (11). Initially, the line is idle (tx=1), but upon receiving the transmission command (tx_en=1), the system switches to the START state, pulling the line to logic-0 and activating the busy signal. After one baud period, it switches to the DATA state, and the 8-bit data packet is transmitted sequentially from the LSB to the MSB (D0-D7). When the transmission of the data bits is complete (bit_cnt=7), the system switches to the STOP state; at this stage, the line is pulled back to logic-1, the packet is terminated, a done signal is generated, and the system returns to the IDLE state for a new transmission.

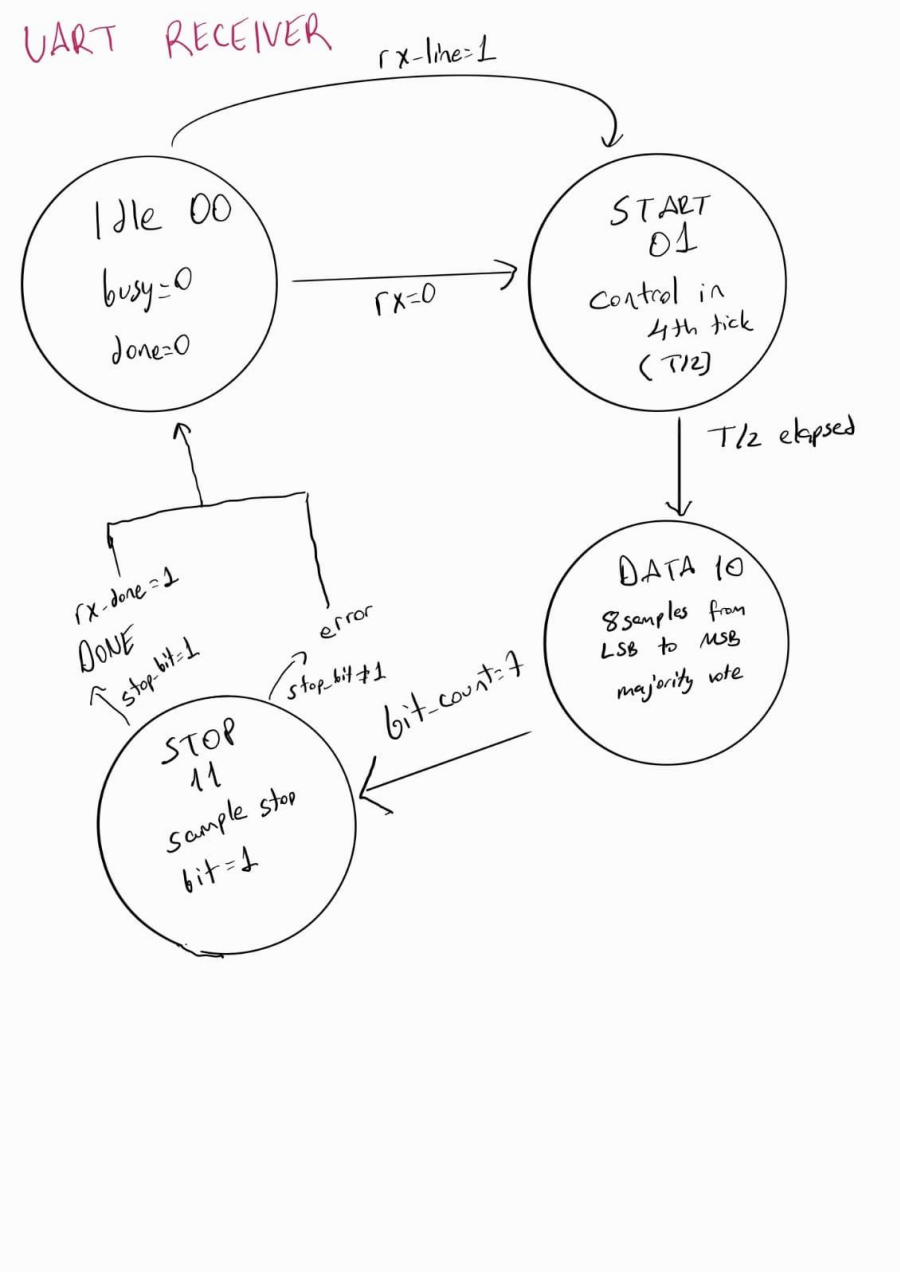

Figure 1.2: State Diagram of UART Receiver

The status diagram showing the control structure of the UART receiver module consists of four basic phases: IDLE (00), START (01), DATA (10), and STOP (11). Initially, in the IDLE state, the system continuously listens to the rx line. When the line drops to logic-0 (Start Bit), it switches to the START state. As indicated in the diagram, a check is performed in the middle of the bit duration (at the 4th tick or T/2) to prevent erroneous readings. Then, it switches to the DATA state, and the 8-bit data packet is sampled sequentially from the LSB to the MSB. When data reception is complete (bit_cnt=7), it switches to the STOP state, and it is checked whether the line is at logic-1 (Stop Bit) level. If the stop bit is valid, the rx_done signal is generated and the data is transferred to the system; otherwise, an error is accepted, and the system returns to its initial state.

TOP MODULE BLOCK DIAGRAM

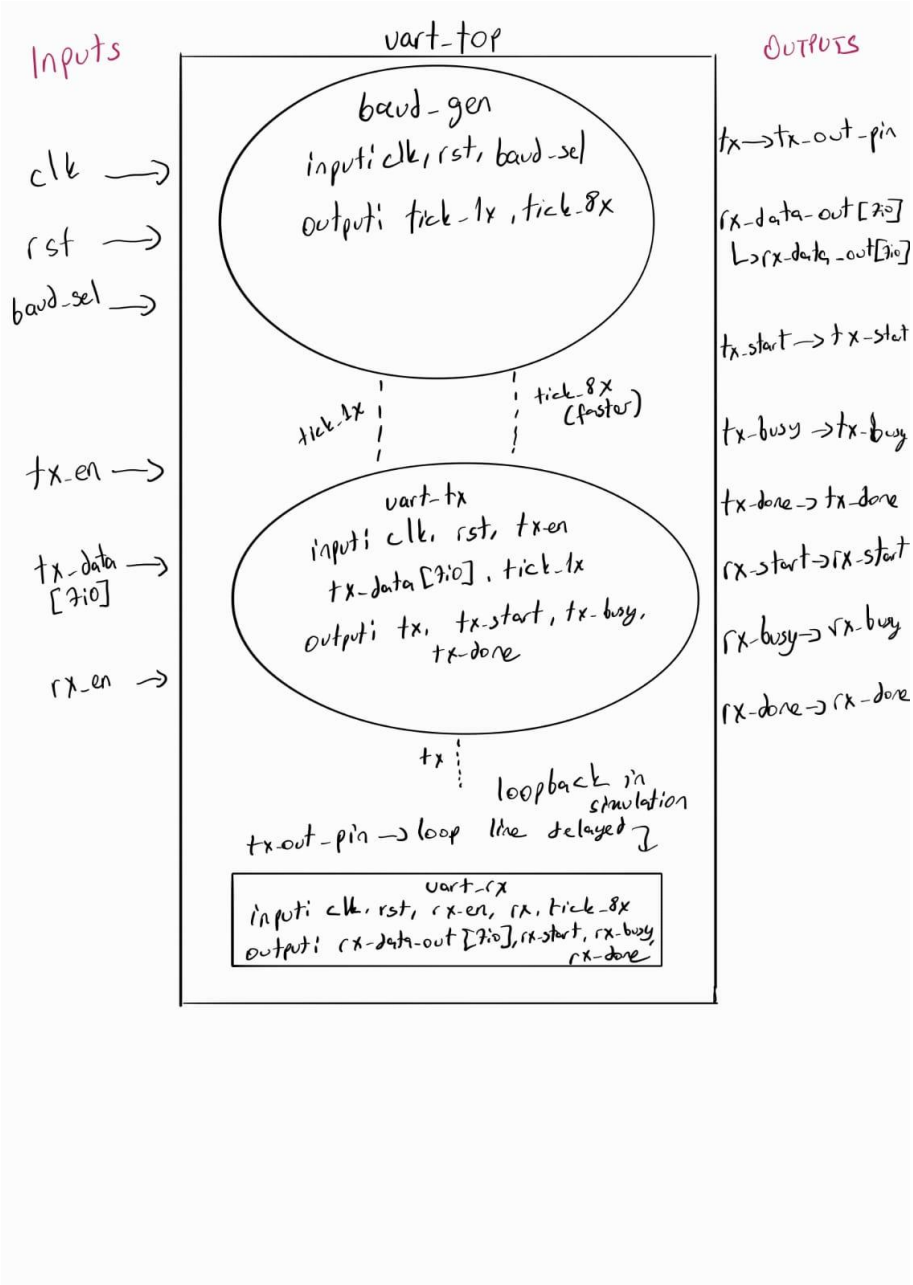

Figure 1.3: Block Diagram of Top Module

Examining the block diagram showing the hierarchical structure of the UART system, it is seen that the design centers around three main modules: the Baud Generator, UART TX (Transmitter), and UART RX (Receiver). The Baud Generator, which is the timing reference of the system, processes the clk, rst, and baud_sel inputs, generating tick_1x signals for the transmitter module and a faster tick_8x signal for the receiver module to enable precise sampling. In the diagram, the data flow is processed from left to right, and it is schematically shown that the serial tx signal from the uart_tx module is routed directly to the input of the uart_rx module via a loopback line in the simulation environment. This structure visualizes a closed-loop test setup where parallel data applied from the tx_data input is converted to serial and transmitted, then captured again by the receiver and transferred to the rx_data_out output.

2- UART TRANSMITTER

RTL CODE

```
`timescale 1ns / 1ps

module uart_tx (
    input  wire       clk,
    input  wire       rst,
    input  wire       tx_en,
    input  wire       tick_8x,
    input  wire [7:0] tx_data,
    output reg        tx_serial,
    output reg        busy,
    output reg        done
);


    localparam [1:0]
        IDLE  = 2'b00,
        START = 2'b01,
        DATA  = 2'b10,
        STOP  = 2'b11;

    // Registerlar
    reg [1:0] state_reg, state_next;
    reg [2:0] s_reg, s_next;
    reg [2:0] n_reg, n_next;
    reg [7:0] b_reg, b_next;
    reg       tx_reg, tx_next;


    always @(posedge clk or posedge rst) begin
        if (rst) begin
            state_reg <= IDLE;
            s_reg     <= 0;
            n_reg     <= 0;
            b_reg     <= 0;
            tx_reg    <= 1'b1;
        end else begin
            state_reg <= state_next;
            s_reg     <= s_next;
            n_reg     <= n_next;
            b_reg     <= b_next;
            tx_reg    <= tx_next;
        end
    end


    always @(*) begin
        state_next = state_reg;
        s_next     = s_reg;
        n_next     = n_reg;
        b_next     = b_reg;
        tx_next    = tx_reg;

        busy = 1'b1;
        done = 1'b0;

        case (state_reg)
            IDLE: begin
                tx_next = 1'b1;
                busy    = 1'b0;

                if (tx_en) begin
                    b_next = tx_data;
                    state_next = START;
                    s_next = 0;
                end
            end

            START: begin
                tx_next = 1'b0;

                if (tick_8x) begin
                    if (s_reg == 7) begin
```

```
                    state_next = DATA;
                    s_next = 0;
                    n_next = 0;
                end else begin
                    s_next = s_reg + 1;
                end
            end
        end

        DATA: begin
            tx_next = b_reg[0];

            if (tick_8x) begin
                if (s_reg == 7) begin
                    s_next = 0;
                    b_next = b_reg >> 1;

                    if (n_reg == 7) begin
                        state_next = STOP;
                    end else begin
                        n_next = n_reg + 1;
                    end
                end else begin
                    s_next = s_reg + 1;
                end
            end
        end

        STOP: begin
            tx_next = 1'b1;

            if (tick_8x) begin
                if (s_reg == 7) begin
                    state_next = IDLE;
                    done       = 1'b1;
                end else begin
                    s_next = s_reg + 1;
                end
            end
        end
    endcase
end


always @(*) tx_serial = tx_reg;

endmodule
```

TESTBENCH CODE

```
`timescale 1ns / 1ps


module tb_uart_tx;

    reg clk;
    reg rst;
    reg tx_en;
    reg tick_8x;
    reg [7:0] tx_data;

    wire tx_serial;
    wire busy;
    wire done;


    uart_tx uut (
        .clk(clk),
        .rst(rst),
        .tx_en(tx_en),
        .tick_8x(tick_8x),
        .tx_data(tx_data),
        .tx_serial(tx_serial),
        .busy(busy),
        .done(done)
    );
```

```
    // --- 1. Clock Üretimi (100 MHz) ---
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 10ns periyot
    end


    initial begin
        tick_8x = 0;
        forever begin
            repeat(4) @(posedge clk);
            tick_8x = 1;
            @(posedge clk);
            tick_8x = 0;
        end
    end


    initial begin

        rst = 1;
        tx_en = 0;
        tx_data = 0;


        #100;
        @(negedge clk);
        rst = 0;
        #100;

        $display("========================================");
        $display("   UART TX TEST STARTING (4 DATA)      ");
        $display("========================================");

      //(Binary: 01010101)
        send_packet(8'h55);

    //(Binary: 10101010)
        send_packet(8'hAA);

        // (Binary: 01000001)
        send_packet(8'h41);

        // (Binary: 11111111)
        send_packet(8'hFF);

        $display("========================================");
        $display("   ALL TEST COMPLETED   ");
        $display("========================================");
        $stop;
    end


    task send_packet(input [7:0] data_in);
        begin

            @(negedge clk);
            tx_data = data_in;
            tx_en = 1;
            $display("[Time: %t] Data is ready: 0x%h", $time, data_in);


            @(negedge clk);
            tx_en = 0;

            wait(busy == 1);

            wait(done == 1);
            $display("[Time: %t] Data sent: 0x%h (DONE is recieved)", $time, data_in);

            #200;
        end
    endtask

endmodule
```

Simulation Results

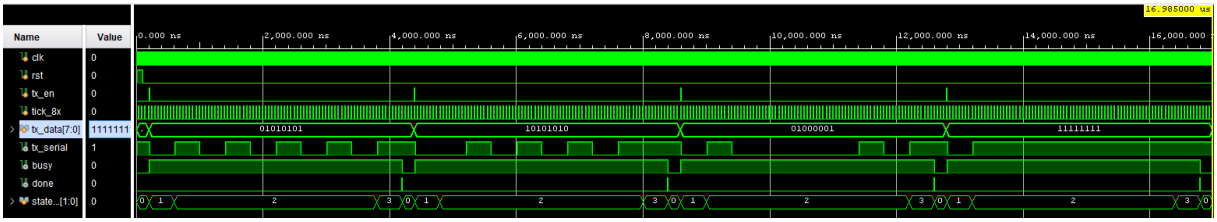

Figure 2.1: Simulation Result of UART Transmitter

Simulation Results (Zoomed)

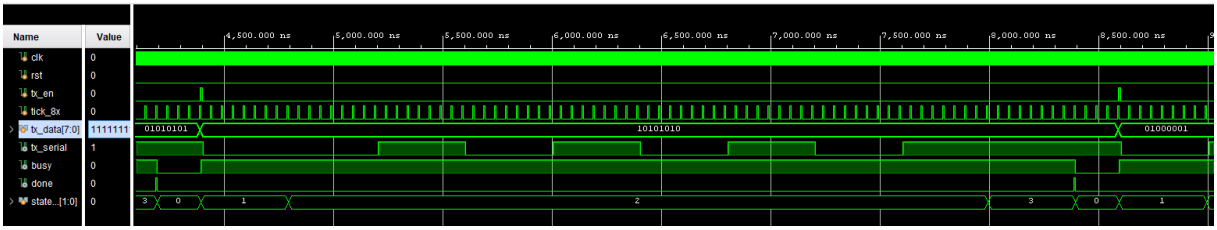

Figure 2.2: Zoomed Simulation Result of UART Transmitter

In the behavioral simulation performed to verify the functionality of the UART transmitter (uart_tx) module, synthetic clock and baud tick signals generated in the testbench environment were used. Within the simulation scenario, four different data packets were transmitted to the module: 0x55 (01010101), 0xAA (10101010), 0x41 (01000001), and 0xFF (11111111).

Examination of the resulting waveform shows that triggering the tx_en signal activates the busy flag, and the Finite State Machine (FSM) exits the IDLE (0) state, subsequently operating the START (1), DATA (2), and STOP (3) states. On the tx_serial line, the Start bit (logic-0), followed by the data bits, and finally the Stop bit (logic-1) were observed with correct timing for each data packet. The generation of a "done" signal at the end of the process and the module returning to the IDLE state confirms that the parallel data has been successfully converted to serial format.

TIMING SUMMARY

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	7,250 ns	Worst Hold Slack (WHS):	0,216 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:	25	Total Number of Endpoints:	25	Total Number of Endpoints:	18

All user specified timing constraints are met.

Figure 2.3: Timing Summary of UART Transmitter

Examination of the timing report obtained after Vivado Implementation shows that the design fully meets all the constraints set for the target 100 MHz (10 ns period) system clock frequency. The Worst Negative Slack (WNS) value obtained from the setup analysis was calculated as 7.250 ns. The fact that this value is positive and the Total Negative Slack (TNS)

value is 0.000 ns indicates that there is no timing violation in the circuit. As a result, the design operates stably at 100 MHz with a safe timing margin.

UTILIZATION REPORT

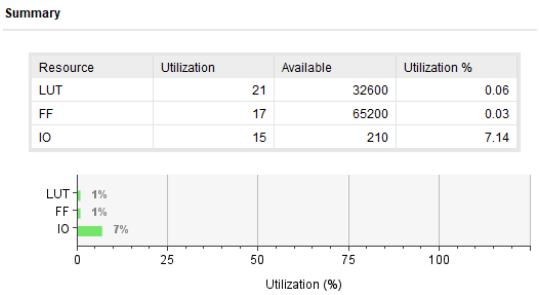

Figure 2.4: Utilization Report of UART Transmitter

Utilization Report is declared above.

POST-IMPLEMENTATION TIMING SIMULATION

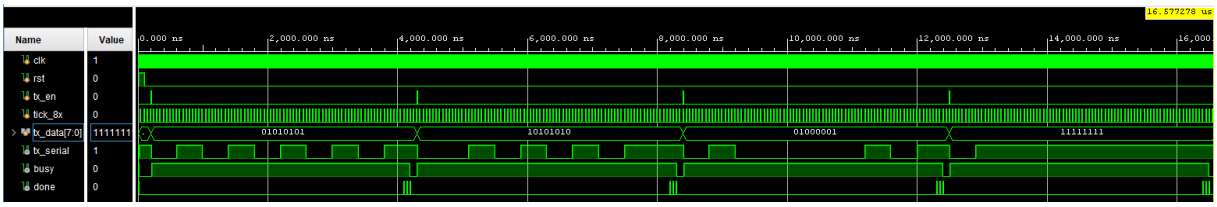

Figure 2.5: Post-Implementation Timing Simulation of UART Transmitter

In the post-implementation simulation, instantaneous instabilities and transient errors (glitches) were observed in the signals due to path and gate delays caused by physical layout. However, the positive WNS value (7.250 ns) in the timing analysis confirms that these delays are damped until the next clock cycle and that the system operates reliably.

3- BAUD GENERATOR

RTL CODE

```
`timescale 1ns / 1ps
module baud_gen (
    input wire clk,
    input wire rst,
    input wire baud_select,
    output reg tick_1x,
    output reg tick_8x
);


    localparam DIV_9600   = 1302; // 100MHz / (9600 * 8)
    localparam DIV_115200 = 108;  // 100MHz / (115200 * 8)


    reg [10:0] counter;
    reg [2:0]  sub_counter;
    reg [10:0] limit;


    always @* begin
```

```
        case (baud_select)
            1'b0: limit = DIV_9600;   // 9600 seçimi
            1'b1: limit = DIV_115200; // 115200 seçimi
        endcase
    end

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            counter <= 0;
            tick_8x <= 0;
        end else begin
            // 8x Tick Üretimi
            if (counter >= limit - 1) begin
                counter <= 0;
                tick_8x <= 1'b1;
            end else begin
                counter <= counter + 1;
                tick_8x <= 1'b0;
            end
        end
    end


    always @(posedge clk or posedge rst) begin
        if (rst) begin
            sub_counter <= 0;
            tick_1x <= 0;
        end else begin
            tick_1x <= 0;

            if (tick_8x) begin
                if (sub_counter == 7) begin
                    sub_counter <= 0;
                    tick_1x <= 1'b1;
                end else begin
                    sub_counter <= sub_counter + 1;
                end
            end
        end
    end

endmodule
```

TESTBENCH CODE

```
`timescale 1ns / 1ps

module tb_baud_gen;

    // Sinyaller
    reg clk;
    reg rst;
    reg baud_select;

    wire tick_1x;
    wire tick_8x;

    // Modül Bağlantısı
    baud_gen uut (
        .clk(clk),
        .rst(rst),
        .baud_select(baud_select),
        .tick_1x(tick_1x),
        .tick_8x(tick_8x)
    );

    // Saat Üretimi (100 MHz -> 10ns periyot)
    always #5 clk = ~clk;

    initial begin
        // Başlangıç
        clk = 0;
        rst = 1;
        baud_select = 1; // Önce Hızlı Mod (115200) ile başlayalım
```

```
        #100;
        rst = 0;

        // --- TEST 1: 115200 Baud (Hızlı) ---
        $display("--- Test 1: 115200 Baud ---");

        #100000;


        // --- TEST 2: 9600 Baud (Yavaş) ---
        $display("--- Test 2: 9600 Baud ---");
        baud_select = 0;
        rst = 1;
        #20;
        rst = 0;


        #1000000;

        $display("--- Test Completed ---");
        $stop;
    end

endmodule
```

Simulation Result

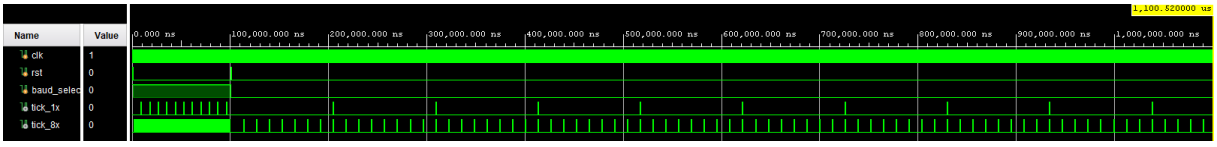

Figure 3.1: Simulation Result of Baud Generator

Simulation Result (9600 MBaud (Zoomed))

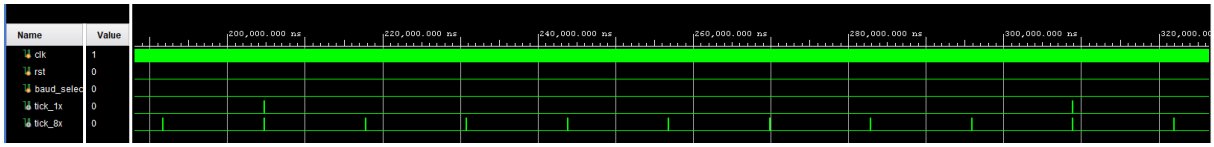

Figure 3.2: Simulation Result of 9600 MBaud Baud Generator

Simulation Result (115200 MBaud(Zoomed))

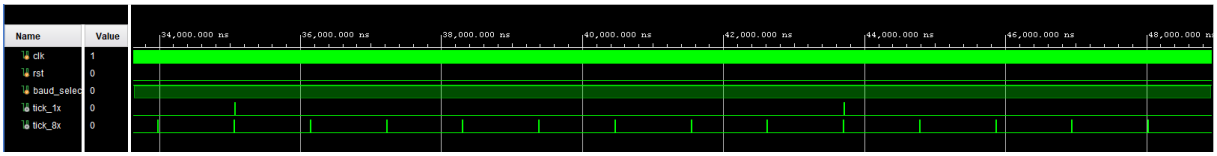

Figure 3.3: Simulation Result of 115200 MBaud Baud Generator

When the simulation waveform of the Baud Generator module is examined, it is seen that the baud_select input directly controls the system's timing reference. In the first stage of the simulation, when baud_select is in logic-1 position, closely spaced tick signals suitable for a baud rate of 115200 (108) are generated by keeping the internal counter limit low; when the input is pulled to logic-0, the limit value is increased (1302) to 9600 baud, and it is observed that the signal period widens significantly. Due to the 'oversampling' technique applied to increase noise immunity in UART communication, there is a fixed frequency division

relationship between the tick_8x and tick_1x signals. Thanks to the auxiliary counter structure in the module, exactly 1 tick_1x (data transmission) pulse is generated for every 8 tick_8x (sampling) pulses produced, and it is confirmed in the simulation graph that 8 tick_8x pulses always fit between two tick_1x signals.

Utilization Report

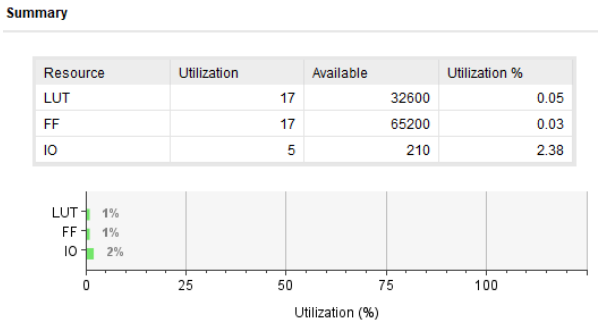

Figure 3.4: Utilization Report of Baud Generator

Utilization Report for Baud Generator is declared above.

Timing Report

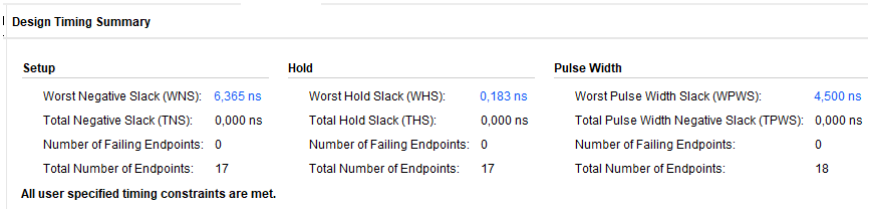

Figure 3.5: Timing Report of Baud Generator

Timing analysis of the design showed that the specified constraints were successfully met. According to setup analysis data, the Worst Negative Slack (WNS) value was 6.365 ns. The positive WNS value and the Total Negative Slack (TNS) value of 0.000 ns indicate that there is no setup time violation in the circuit. These results confirm that the design operates stably with safe timing margins at the target clock frequency (100 MHz).

POST IMPLEMENTATION TIMING

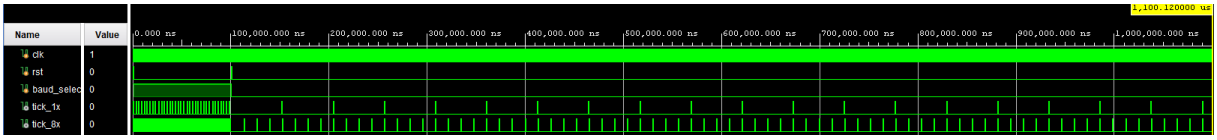

Figure 3.5: Post-Implementation Timing Simulation of Baud Generator

Analysis of the post-implementation timing simulation waveforms revealed no significant delays or glitches in the output signals that would disrupt data transmission or cause

erroneous logic levels. It was confirmed that the signals were generated cleanly and stably within the targeted timing constraints.

4- UART RECEIVER

RTL CODE

```
`timescale 1ns / 1ps

module uart_rx(
    input wire clk,
    input wire rst,
    input wire rx,
    input wire rx_en,
    input wire tick_8x,

    output reg [7:0] rx_data,
    output reg rx_start,
    output reg rx_busy,
    output reg rx_done
    );

    // Durumlar
    localparam [1:0]
        IDLE  = 2'b00,
        START = 2'b01,
        DATA  = 2'b10,
        STOP  = 2'b11;

    reg [1:0] state;
    reg [2:0] tick_cnt;
    reg [2:0] bit_cnt;
    reg [1:0] vote_cnt;
    reg [7:0] shift_reg;


    reg rx_sync1, rx_sync2;
    wire rx_in = rx_sync2;

    always @(posedge clk) begin
        if (rst) begin
            rx_sync1 <= 1'b1;
            rx_sync2 <= 1'b1;
        end else begin
            rx_sync1 <= rx;
            rx_sync2 <= rx_sync1;
        end
    end


    always @(posedge clk) begin
        if (rst) begin
            state     <= IDLE;
            tick_cnt  <= 0;
            bit_cnt   <= 0;
            vote_cnt  <= 0;
            shift_reg <= 0;
            rx_data   <= 0;
            rx_busy   <= 0;
            rx_done   <= 0;
            rx_start  <= 0;
        end else begin

            rx_done  <= 0;
            rx_start <= 0;

            case (state)

                IDLE: begin
                    rx_busy <= 0;
                    tick_cnt <= 0;
                    bit_cnt <= 0;
```

```
                    if (rx_en && rx_in == 1'b0) begin
                        state    <= START;
                        rx_busy  <= 1;
                        rx_start <= 1;
                    end
                end


                START: begin
                    if (tick_8x) begin

                        if (tick_cnt == 3) begin
                            if (rx_in == 1'b1) begin
                                state   <= IDLE;
                                rx_busy <= 0;
                            end
                        end

                        if (tick_cnt == 7) begin
                            state    <= DATA;
                            tick_cnt <= 0;
                            vote_cnt <= 0;
                        end else begin
                            tick_cnt <= tick_cnt + 1;
                        end
                    end
                end

                DATA: begin
                    if (tick_8x) begin

                        if (tick_cnt == 3 || tick_cnt == 4 || tick_cnt == 5) begin
                            if (rx_in == 1'b1)
                                vote_cnt <= vote_cnt + 1;
                        end

                        if (tick_cnt == 7) begin
                            tick_cnt <= 0;
                            shift_reg <= { (vote_cnt >= 2), shift_reg[7:1] };

                            vote_cnt <= 0;

                            if (bit_cnt == 7)
                                state <= STOP;
                            else
                                bit_cnt <= bit_cnt + 1;
                        end else begin
                            tick_cnt <= tick_cnt + 1;
                        end
                    end
                end

                STOP: begin
                    if (tick_8x) begin
                        if (tick_cnt == 7) begin
                            state   <= IDLE;
                            rx_done <= 1;
                            rx_data <= shift_reg;
                            rx_busy <= 0;
                        end else begin
                            tick_cnt <= tick_cnt + 1;
                        end
                    end
                end
            endcase
        end
    end

endmodule
```

TESTBENCH CODE

```
`timescale 1ns / 1ps
```

```
module tb_uart_rx;

    reg clk;
    reg rst;
    reg rx;
    reg rx_en;
    reg tick_8x;

    wire [7:0] rx_data;
    wire rx_start;
    wire rx_busy;
    wire rx_done;

    uart_rx uut (
        .clk(clk),
        .rst(rst),
        .rx(rx),
        .rx_en(rx_en),
        .tick_8x(tick_8x),
        .rx_data(rx_data),
        .rx_start(rx_start),
        .rx_busy(rx_busy),
        .rx_done(rx_done)
    );


    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end


    localparam TICK_PERIOD = 400; // 400ns

    initial begin
        tick_8x = 0;
        forever begin
            #(TICK_PERIOD);
            tick_8x = 1;
            #10;
            tick_8x = 0;
        end
    end

    initial begin
        rst = 1;
        rx_en = 1;
        rx = 1;
        #200;
        rst = 0;
        #500;

        // TEST 1: Normal Veri (0x55 - 01010101)
        $display("TEST 1: 0x55 Gonderiliyor...");
        uart_send_byte(8'h55);
        #2000; // Paketler arası boşluk

        // TEST 2: Faz Farklı Veri (0xAA - 10101010)

        #353;
        $display("TEST 2: 0xAA Gonderiliyor (Faz farkli)...");
        uart_send_byte(8'hAA);
        #2000;

        // TEST 3: 0xC3 (11000011)
        #127;
        $display("TEST 3: 0xC3 Gonderiliyor...");
        uart_send_byte(8'hC3);
        #2000;

        // TEST 4: 0x18 (00011000)
        #88;
        $display("TEST 4: 0x18 Gonderiliyor...");
        uart_send_byte(8'h18);
        #2000;

        $stop;
```

```
    end


    task uart_send_byte(input [7:0] data);
        integer i;
        begin
            // 1. Start Bit (Low)
            rx = 0;
            #(TICK_PERIOD * 8 + 10); // 1 Bit süresi bekle (tick_period * 8)

            // 2. Data Bits (LSB First)
            for (i=0; i<8; i=i+1) begin
                rx = data[i];
                #(TICK_PERIOD * 8 + 10);
            end

            // 3. Stop Bit (High)
            rx = 1;
            #(TICK_PERIOD * 8 + 10);
        end
    endtask

endmodule
```

Simulation Result

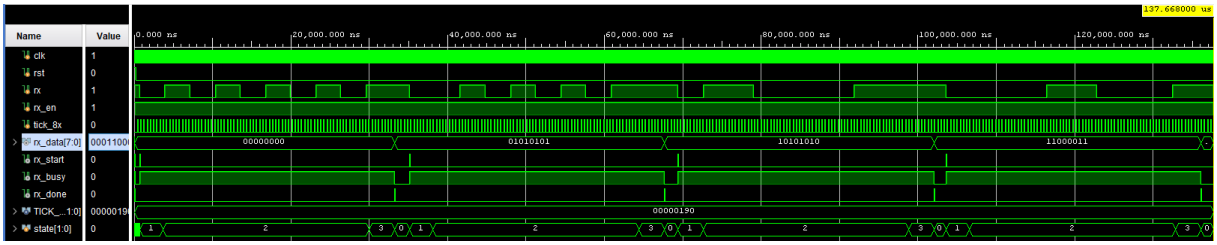

Figure 4.1: Simulation Result of UART Receiver

Simulation Result (Zoomed)

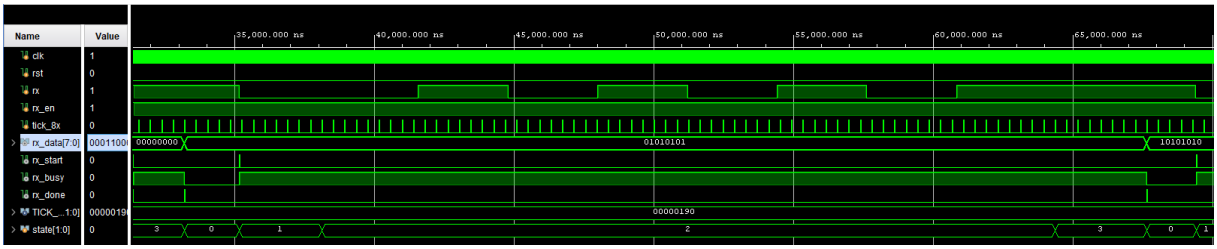

Figure 4.2: Zoomed Simulation Result of UART Receiver

In the behavioral simulation of the UART receiver (uart_rx) module, four different data packets with values of 0x55, 0xAA, 0xC3, and 0x18 were tested to verify its asynchronous data reception capability. These serial data packets, generated in the testbench environment and sent with random time delays (phase shifts) between packets, were processed by the module using an 8x oversampling technique. Examination of the simulation waveform confirmed that the serial bit stream on the rx line was successfully converted to parallel data under FSM control (Start-Data-Stop), the values sent at the rx_data output (e.g., 00011000 in the last packet, i.e., 0x18) were generated without errors, and the rx_done flag was activated after each successful reception.

Utilization Report

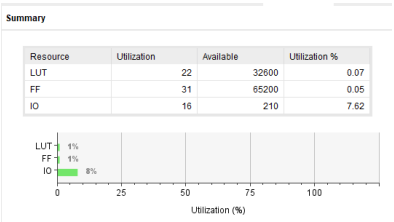

Figure 4.3: Utilization Report of UART Receiver

Utilization Report for UART Receiver is declared above.

Timing Summary

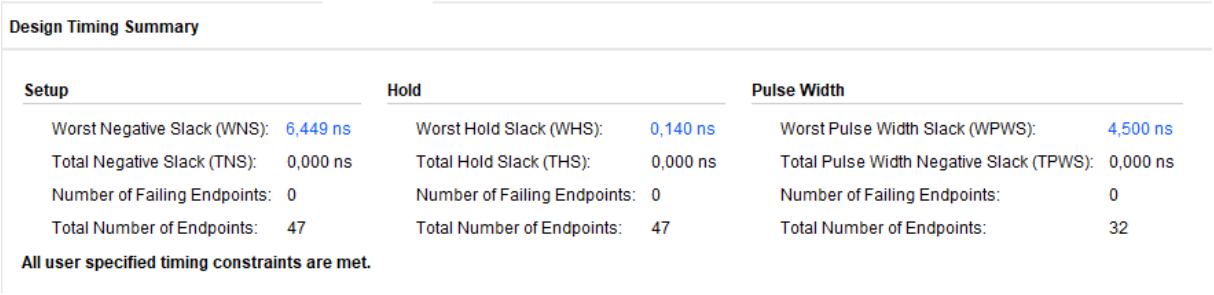

Figure 4.4: Timing Summary of UART Receiver

Examination of the timing summary report obtained after the physical implementation of the design shows that the system fully meets the targeted 100 MHz (10 ns period) operating frequency constraints. The Worst Negative Slack (WNS) value obtained in the setup timing analysis is 6.449 ns, and the Total Negative Slack (TNS) value is measured as 0.000 ns. The positive WNS value and the zero failing endpoints confirm that there are no timing violations in the circuit and that the design operates safely and stably at the predicted speed.

POST IMPLEMENTATION TIMING SIMULATION

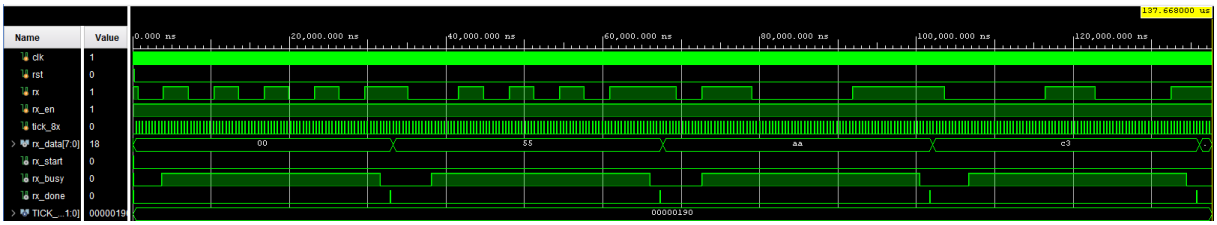

Figure 4.4: Post-Implementation Timing Result of UART Receiver

When the post-implementation timing simulation was examined, it was observed that, unlike the ideal behavioral simulation, there were shifts of the microsecond order in the timing of the rx_busy and rx_done control signals. This situation, caused by physical routing and logic gate delays within the FPGA, causes the signals to be updated with a certain delay, not immediately after the relevant clock edge. However, these natural physical delays remained within the timing constraints of the design and did not negatively affect the data acquisition accuracy.

5- TOP MODULE

RTL CODE

```
`timescale 1ns / 1ps

module uart_top (
    input wire clk,
    input wire rst,
    input wire baud_select,

    input wire tx_en,
    input wire [7:0] tx_data,
    output wire tx,
    output wire tx_busy,
    output wire tx_done,


    input wire rx,
    input wire rx_en,
    output wire [7:0] rx_data,
    output wire rx_busy,
    output wire rx_done
);

    wire tick_1x_w;
    wire tick_8x_w;

    wire tx_start_unused;
    wire rx_start_unused;

    // ------------------------------------------------------------------
    // 1. BAUD RATE GENERATOR INSTANCE
    // ------------------------------------------------------------------
    baud_gen inst_baud_gen (
        .clk(clk),
        .rst(rst),
        .baud_select(baud_select),
        .tick_1x(tick_1x_w),
        .tick_8x(tick_8x_w)
    );

    // ------------------------------------------------------------------
    // 2. UART TRANSMITTER INSTANCE
    // ------------------------------------------------------------------
uart_tx inst_uart_tx (
        .clk(clk),
        .rst(rst),
        .tx_en(tx_en),
        .tick_8x(tick_8x_w),
        .tx_data(tx_data),
        .tx_serial(tx),
        .busy(tx_busy),
        .done(tx_done)

    );;

    // ------------------------------------------------------------------
    // 3. UART RECEIVER INSTANCE
    // ------------------------------------------------------------------
    uart_rx inst_uart_rx (
        .clk(clk),
        .rst(rst),
        .rx(rx),
        .rx_en(rx_en),
        .tick_8x(tick_8x_w),
        .rx_data(rx_data),
        .rx_start(rx_start_unused),
        .rx_busy(rx_busy),
        .rx_done(rx_done)
    );

endmodule
```

TESTBENCH CODE

```
`timescale 1ns / 1ps

module tb_uart_top;

    reg clk;
    reg rst;
    reg baud_select;

    // TX (Verici) Girişleri
    reg tx_en;
    reg [7:0] tx_data_in;

    // RX (Alıcı) Çıkışları
    wire [7:0] rx_data_out;
    wire rx_done;

    // Durum Sinyalleri
    wire tx_busy, tx_done;
    wire rx_busy;

    // Loopback Hattı
    wire serial_loop;

    // --- UUT ---
    uart_top uut (
        .clk(clk),
        .rst(rst),
        .baud_select(baud_select),
        .tx_en(tx_en),
        .tx_data(tx_data_in),
        .tx(serial_loop),
        .tx_busy(tx_busy),
        .tx_done(tx_done),
        .rx(serial_loop),
        .rx_en(1'b1),
        .rx_data(rx_data_out),
        .rx_busy(rx_busy),
        .rx_done(rx_done)
    );

    // Clock (100 MHz - 10ns)
    always #5 clk = ~clk;

    reg [7:0] test_vectors [0:3];
    integer i;

    initial begin

        test_vectors[0] = 8'h55;
        test_vectors[1] = 8'hAA;
        test_vectors[2] = 8'h12;
        test_vectors[3] = 8'hD4;

        clk = 0;
        rst = 1;
        baud_select = 0; // 9600 Baud   1 for
        tx_en = 0;
        tx_data_in = 0;


        #200;
        @(negedge clk);
        rst = 0;
        #100;

        $display("========================================");
        $display("   UART LOOPBACK TESTI (TIMING FIX)");
        $display("   (Inputs driven at NEGEDGE)");
        $display("========================================");

        for (i = 0; i < 4; i = i + 1) begin

            @(negedge clk);
            tx_data_in = test_vectors[i];
```

```
            tx_en = 1;
            $display("[%0t] SENDING (Tx): %h", $time, tx_data_in);

            @(negedge clk);
            tx_en = 0;

            wait(rx_done == 1);

            #1000;


            if (rx_data_out === test_vectors[i])
                $display("    -> [SUCCESSFULL] Rx Read: %h", rx_data_out);
            else
                $display("    -> [ERROR]      Expected: %h, Read: %h", test_vectors[i],
rx_data_out);


            #20000;
        end

        $display("========================================");
        $display("    TEST COMPLETED");
        $display("========================================");
        $stop;
    end

endmodule
```

Simulation Results

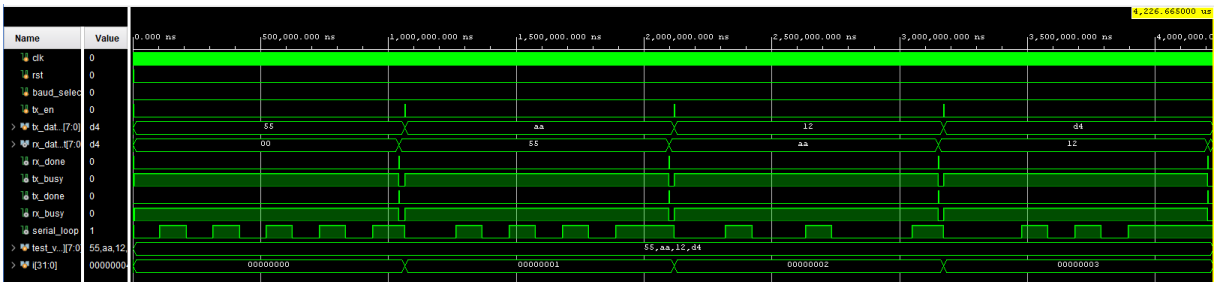

Figure 5.1: Simulation Result of Top Module

TCL Console Output

```
# run all
    -> [SUCCESSFULL] Rx Read: 55
  [1062810000] SENDING (Tx): aa
    -> [SUCCESSFULL] Rx Read: aa
  [2117430000] SENDING (Tx): 12
    -> [SUCCESSFULL] Rx Read: 12
  [3172050000] SENDING (Tx): d4
    -> [SUCCESSFULL] Rx Read: d4
  ========================================
    TEST COMPLETED
  ========================================
```

Figure 5.2: TCL Console Output for Top Module

In the test scenario created to perform a top-level simulation of the UART communication system, four different data packets with values of 0x55, 0xAA, 0x12, and 0xD4 were applied to the system. Examination of the simulation waveform shows that the serial data transmitted by the transmitter (TX) to the serial_loop line is successfully captured by the receiver (RX) module and transmitted to the rx_data output without error. The one-to-one matching of the

transmitted tx_data values and the received rx_data values (with a delay), and the synchronized generation of the tx_done and rx_done signals at the end of each packet transfer, proves that the system operates in a stable loopback without data loss.

TIMING REPORT

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5,651 ns	Worst Hold Slack (WHS):	0,154 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:	83	Total Number of Endpoints:	83	Total Number of Endpoints:	60

All user specified timing constraints are met.

Figure 5.3: Timing Report of Top Module

Examination of the timing summary report obtained after the physical implementation of the entire system (Top Module) reveals that the design fully complies with the targeted 100 MHz (10 ns period) operating frequency constraints. The analysis showed a Worst Negative Slack (WNS) value of 5.651 ns, a Total Negative Slack (TNS) value of 0.000 ns, and the absence of any failing endpoints, indicating no setup or hold timing violations in the circuit. These data confirm that the UART system operates stably and reliably at the predicted speed within its integrated structure.

Utilization Report

Summary

Resource	Utilization	Available	Utilization %
LUT	54	32600	0.17
FF	59	65200	0.09
IO	27	210	12.86

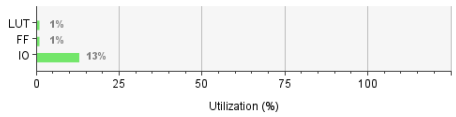

Figure 5.4: Utilization Report of Top Module

Utilization Report for Top Module is declared above.

POST-IMPLEMENTATION TIMING SIMULATION RESULTS

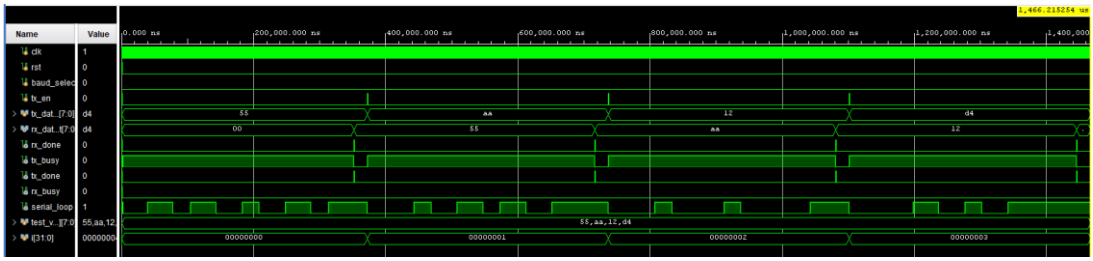

Figure 5.5: Post-Implementation Timing Simulation of Top Module

In the post-implementation timing simulation of the UART Top module, significant time shifts were observed in signal transitions due to unavoidable propagation delays caused by physical placement and routing. However, despite these physical delays, the system synchronization remained intact, and all data packets 0x55, 0xAA, 0x12, and 0xD4 sent in the test scenario were correctly sampled by the receiver module and transmitted to the rx_data output without error. The results demonstrate that the design is tolerant to delays under real hardware conditions and operates while maintaining data integrity.

REFERENCES

1- **M. Barr and A. Massa, Programming Embedded Systems, 2nd ed., O'Reilly Media, 2006. (Asynchronous serial communication and embedded system interfaces)**

2- **S. Kilts, Advanced FPGA Design: Architecture, Implementation, and Optimization, Wiley-IEEE Press, 2007. (FPGA-based FSM design and timing-aware implementations)**

3- **N. Weste and D. Harris, CMOS VLSI Design: A Circuits and Systems Perspective, 4th ed., Addison-Wesley, 2011. (Timing constraints, setup/hold analysis, and clocked systems)**

4- **Microchip Technology Inc., "UART Communication Module User Guide," Technical Reference Manual, 2020. (UART frame structure, start/stop bit timing, receiver sampling)**

5- **Xilinx Inc., "Vivado Design Suite User Guide: Simulation (UG900)"**