ISTANBUL TECHNICAL UNIVERSITY
FACULTY OF ELECTRICAL –
ELECTRONICS

Digital System Design Applications

Project-2

Class

Class Lecturer: Sıddıka Berna Örs Yalçın

Class Assistants: Bilgi Görkem Yazgaç, Deniz Zakir Eroğlu

Term: 2025/2026 Fall Term

Student

Name-Surname: Mehmet Emin Delibaş

No: 040200092

E-mail: delibas20@itu.edu.tr

Due Date: 27/12/2025

UART PROTOCOL

1. Explanation

Universal Asynchronous Receiver Transmitter (UART) is a widely used hardware communication protocol that enables asynchronous serial data transfer between digital devices. Unlike synchronous communication methods, UART does not use a shared clock signal between the transmitter and the receiver. Instead, both sides operate with a predefined baud rate, which determines the duration of each transmitted bit.

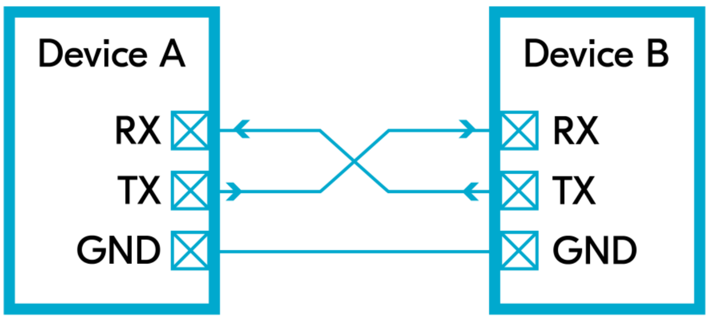
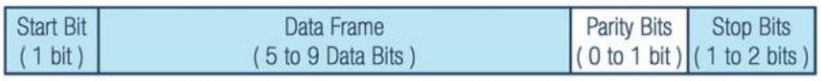


Figure 1: *Physical UART connection between two devices showing TX, RX and common ground lines.[1]*

Figure 1 shows the basic physical connection of two UART-enabled devices. Each device has a transmit (TX) and a receive (RX) line. The TX pin of one device is connected to the RX pin of the other device, while both systems share a common ground (GND). This cross-connection allows full-duplex communication between the two devices.
UART communication is based on framing the data with control bits. A general UART frame structure is illustrated in Figure 2.



Start Bit (1 bit)	Data Frame (5 to 9 Data Bits)	Parity Bits (0 to 1 bit)	Stop Bits (1 to 2 bits)

Figure 2: *General UART frame structure including start bit, data bits, optional parity bit and stop bit(s).[2]*

As shown in the figure, a UART frame typically consists of:

- one **start bit**,
- **5 to 8 data bits**,
- an optional **parity bit**, and
- one or two **stop bits**.

In this project, the frame format is selected as **1 start bit, 8 data bits, no parity, and 1 stop bit (8N1)**, which is one of the most commonly used configurations.

The timing behavior of a UART transmission is presented in Figure 3.

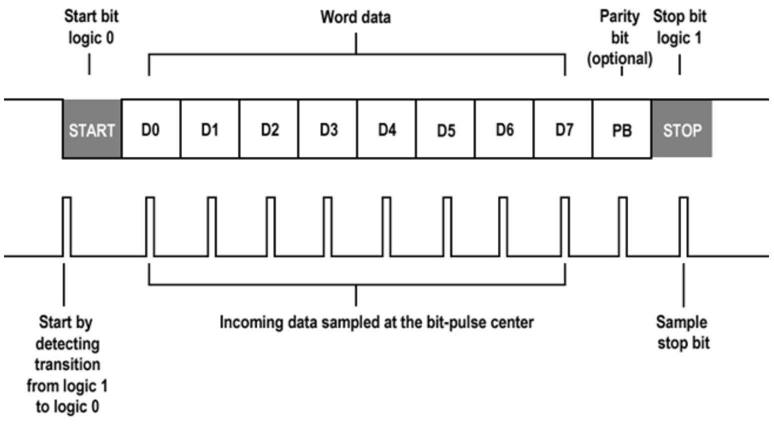


Figure 3: *UART timing diagram illustrating start bit detection and sampling of data bits at the center of each bit period.[3]*

During the idle state, the UART line is held at logic ‘1’. Communication begins when the transmitter pulls the line to logic ‘0’, indicating the start bit. After that, the data bits are transmitted starting from the least significant bit (LSB). The receiver detects the falling edge of the start bit and then samples the incoming data at the center of each bit period. After all data bits are received, the stop bit returns the line to logic ‘1’, marking the end of the frame.

To handle asynchronous timing, the receiver generally uses oversampling. In this project, the receiver uses an 8x baud tick as a timing reference and samples the serial input once per bit period after alignment to reconstruct each data bit.

Figure 4 illustrates how parallel data is converted into serial form in the UART transmitter.

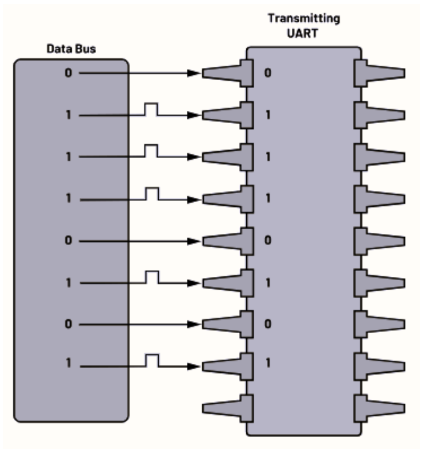


Figure 4: *Parallel-to-serial data conversion performed by the UART transmitter.[4]*

The transmitter takes an 8-bit parallel input from the data bus and shifts it out bit by bit according to the baud rate. Similarly, the UART receiver performs the inverse operation by reconstructing the serial data stream back into an 8-bit parallel word.

The reverse operation is performed in the UART receiver, as illustrated in Figure 5. The receiver collects the incoming serial data bits from the RX line and reconstructs them into an 8-bit parallel word on the data bus. Each bit is sampled at the appropriate time instant, and after all data bits are received, the complete parallel data is made available at the output of the receiver.

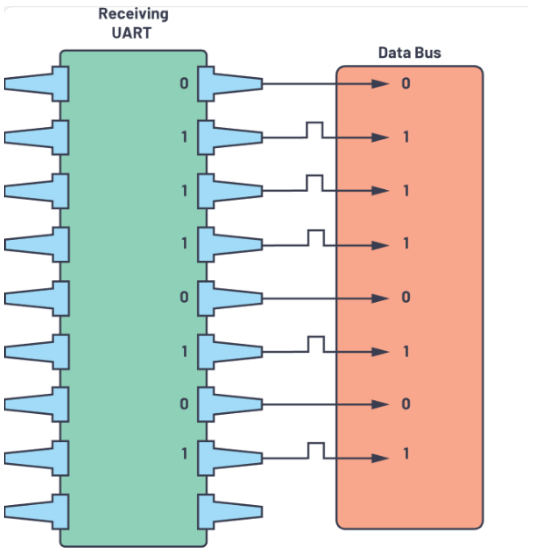


Figure 5: *Serial-to-parallel data reconstruction performed by the UART receiver.[5]*

Overall, UART offers a simple and efficient method for serial communication with minimal wiring requirements. Its asynchronous nature and configurable frame structure make it suitable for a wide range of digital and embedded system applications.

2. Algorithm

In this project, the UART driver is designed in a modular structure consisting of four main blocks: uart_tx, uart_rx, baud_gen, and uart_top. Each module is responsible for a specific task in the overall communication process.

The baud_gen module generates the required baud rate and its oversampled version from the system clock. This module provides timing references for both the transmitter and the receiver.

The uart_tx module implements the transmitter side of the UART protocol. It takes 8-bit parallel data as input and sends it serially according to the UART frame format using a finite state machine structure.

The uart_rx module performs the reverse operation. It receives serial data from the RX line, samples it using the oversampled baud clock, and reconstructs the original 8-bit parallel data at its output.

Finally, the uart_top module acts as the top-level integration block where the transmitter, receiver, and baud rate generator modules are instantiated and interconnected. It provides a single interface for external control signals and data input/output.

This modular design improves readability, reusability, and ease of debugging, while also allowing each block to be verified independently through simulation.

3. Baud Rate Generator Module

The UART design requires two timing references: the baud rate clock used for bit-to-bit progression, and an oversampled clock used by the receiver for reliable sampling. Therefore, the baud rate generator is designed to produce **baud** and **8x baud** timing pulses from the internal **100 MHz** system clock.
The module is parameterized by the baud rate value (**BAUD**) and generates a one-clock-cycle pulse called **tick_8x** every:

$$DIV_{8x} = \frac{f_{clk}}{8 \cdot f_{baud}} - 1$$

where $f_{clk} = 100\ MHz$and $f_{baud}$is selected as **115200** or **9600** in this project. Each time **tick_8x** is asserted, an internal modulo-8 counter is increased. When eight **tick_8x** pulses occur, the module produces a second one-clock-cycle pulse called **tick_baud**, which represents the actual baud timing.
An enable signal (**en**) is used to start/stop the divider operation. When **en = 0**, counters are cleared so that the timing restarts deterministically when enabled again. This behavior is especially useful for the receiver side, where sampling is aligned after detecting the start bit.
Verification:
The module is simulated for both baud rates. For **115200**, the expected divider for the oversampled tick is:

$$DIV_{8x} = \frac{100 \times 10^6}{8 \times 115200} - 1 \approx 107$$

For **9600**, the expected value is:

$$DIV_{8x} = \frac{100 \times 10^6}{8 \times 9600} - 1 \approx 1301$$

In the simulation waveform, **tick_8x** appears periodically according to the computed divider value, and **tick_baud** is observed once for every eight **tick_8x** pulses, confirming correct operation.

3.1. Design Source

```verilog
// Baud rate generator module
// Generates tick pulses for 8x oversampling and baud rate timing
module baud_gen #(parameter [16:0] freq = 115200)
(
    input  clk,                 // System clock (100 MHz)
    input  rst,                 // Active-high synchronous reset
    input  baud_en,             // Enable signal for baud generator
    output count_8x_ready,      // Pulse indicating 8x baud tick
    output count_baud_ready     // Pulse indicating baud tick (1x)
);

    // Internal clock frequency in Hz
    localparam integer clk_freq = 100000000; // 100 MHz

    // Registers to hold output pulses
    reg count_8x_ready_reg;
    reg count_baud_ready_reg;

    // Counter for generating 8x baud ticks
    reg [13:0] counter_8x;

    // Counter for dividing 8x ticks into 1 baud tick
    reg [2:0] counter_baud;

    // Sequential logic driven by system clock
    always @(posedge clk) begin
        if (rst == 1'b1) begin
            // Reset all counters and outputs
            counter_baud <= 3'b0;             // Baud tick counter reset
            count_baud_ready_reg <= 1'b0;     // Clear baud tick pulse
            counter_8x <= 13'b0;               // 8x tick counter reset
            count_8x_ready_reg <= 1'b0;       // Clear 8x tick pulse
        end
        else if (baud_en == 1'b1) begin
            // Baud generator active
            // Check if 8x counter reached terminal count
            if (counter_8x == ((clk_freq/(freq*8)) - 1)) begin
                counter_8x <= 13'b0;           // Reset 8x counter
                count_8x_ready_reg <= 1'b1; // Generate 8x tick pulse

                // Count 8 pulses to generate one baud tick
                if (counter_baud == 3'd7) begin
                    counter_baud <= 3'b0;             // Reset baud counter
                    count_baud_ready_reg <= 1'b1;  // Generate baud tick
pulse
                end
                else begin
                    counter_baud <= counter_baud + 1'b1; // Increment baud
counter
                    count_baud_ready_reg <= 1'b0;        // No baud pulse
yet
                end
            end
            else begin
                // Continue counting towards next 8x tick
                counter_8x <= counter_8x + 1'b1;
                count_8x_ready_reg <= 1'b0;
                count_baud_ready_reg <= 1'b0;
            end
        end
        else begin
            // When baud generator is disabled, clear counters and outputs
            count_8x_ready_reg <= 1'b0;
            counter_8x <= 13'b0;
            counter_baud <= 3'b0;
            count_baud_ready_reg <= 1'b0;
        end
    end

    // Assign registered outputs to module outputs
    assign count_8x_ready   = count_8x_ready_reg;
    assign count_baud_ready = count_baud_ready_reg;

endmodule
```

The baud_gen module generates timing pulses for UART operation by dividing the 100 MHz system clock to the desired baud rate given by the parameter freq. The counter counter_8x produces an oversampling pulse (count_8x_ready) at eight times the baud rate. A second counter, counter_baud, counts these pulses and generates a baud-rate pulse (count_baud_ready) every eight oversampling ticks. The module operates only when the enable signal baud_en is asserted. When baud_en is low or when reset is active, all counters and outputs are cleared. This structure provides accurate baud and oversampling timing for the UART transmitter and receiver.

3.2. Testbench Code

```
// Testbench for the baud rate generator module
// This testbench verifies the generation of 8x and baud-rate tick pulses
module baud_gen_tb();

    // Baud rate parameter (17 bits are enough to represent 115200)
    parameter [16:0] freq = 115200;

    // Testbench signals
    reg clk = 1'b0;           // System clock
    reg rst = 1'b0;           // Reset signal
    reg baud_en = 1'b0;       // Enable signal for baud generator

    // Outputs from DUT
    wire count_8x_ready;      // 8x oversampling tick pulse
    wire count_baud_ready;    // Baud-rate tick pulse

    // Instantiate the DUT (Device Under Test)
    baud_gen #(
        .freq(freq)
    ) uut (
        .clk(clk),                        // Connect clock
        .rst(rst),                        // Connect reset
        .baud_en(baud_en),                // Connect enable
        .count_8x_ready(count_8x_ready), // Connect 8x tick output
        .count_baud_ready(count_baud_ready) // Connect baud tick output
    );

    // Generate a 100 MHz clock (10 ns period, toggle every 5 ns)
    always begin
        #5 clk = ~clk;
    end

    // Stimulus process
    initial begin
        // Initial state: apply reset and keep module disabled
        rst = 1'b1;
        baud_en = 1'b0;

        // Wait for 10 ns and deassert reset
        #10;
        rst = 1'b0;

        // Enable the baud generator
        baud_en = 1'b1;

        // Let the generator run long enough to observe output pulses
        #11000;

        // Disable the baud generator
        baud_en = 1'b0;

        // Wait for a short time and finish the simulation
        #200;
        $finish;
    end

endmodule
```

The baud_gen_tb module is a testbench designed to verify the functionality of the baud_gen module. It generates a 100 MHz clock by toggling the clk signal every 5 ns and applies reset and enable stimuli to the design under test.
At the beginning of the simulation, the reset signal rst is asserted to initialize the internal counters of the baud generator. After a short delay, the reset is deasserted and the enable signal baud_en is set high to activate the baud generator. The testbench then allows the module to run for a sufficient duration so that both the

oversampling tick (count_8x_ready) and the baud-rate tick (count_baud_ready) can be observed.
After this interval, the enable signal is deasserted to verify that the module stops generating output pulses when disabled. Finally, the simulation is terminated. This testbench confirms that the baud generator produces correct timing pulses and responds properly to reset and enable control signals.

3.3. Simulation Waveform

For 115200Hz:

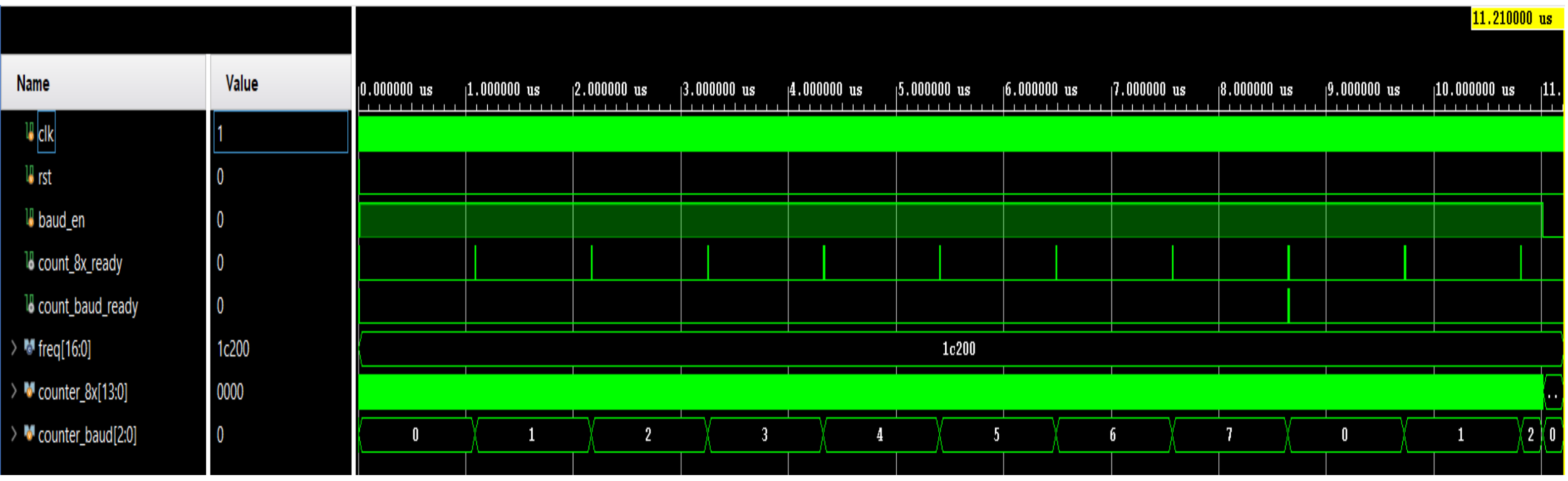


Figure 6: *Waveform of the baud_gen module showing generation of count_8x_ready and count_baud_ready pulses for a baud rate of 115200 bps under a 100 MHz clock..*

For 9600Hz:

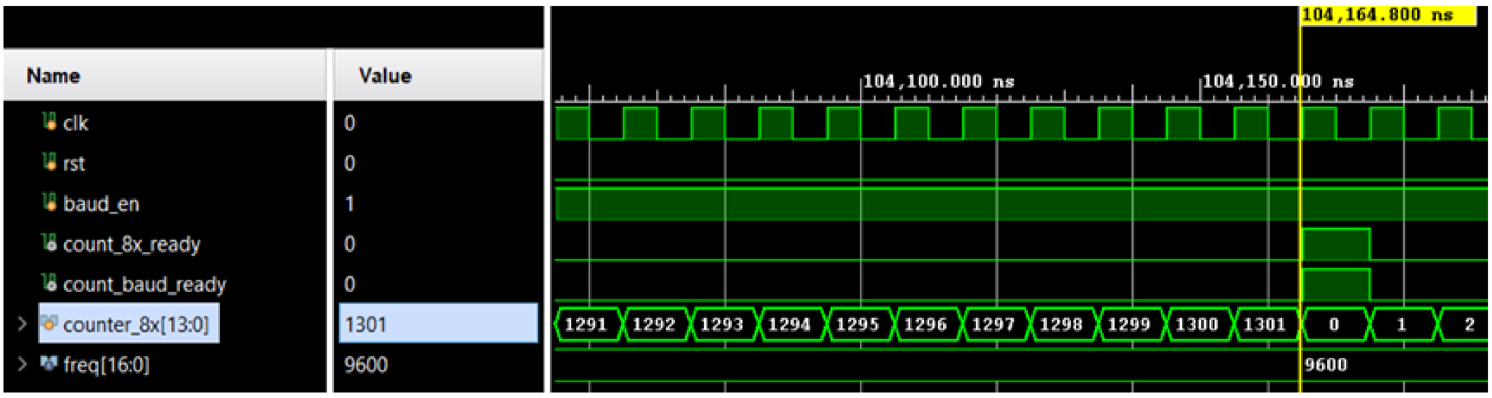


Figure 7: *Simulation waveform of baud_gen for 9600 baud.*

3.4. Post-implementation Timing Simulation

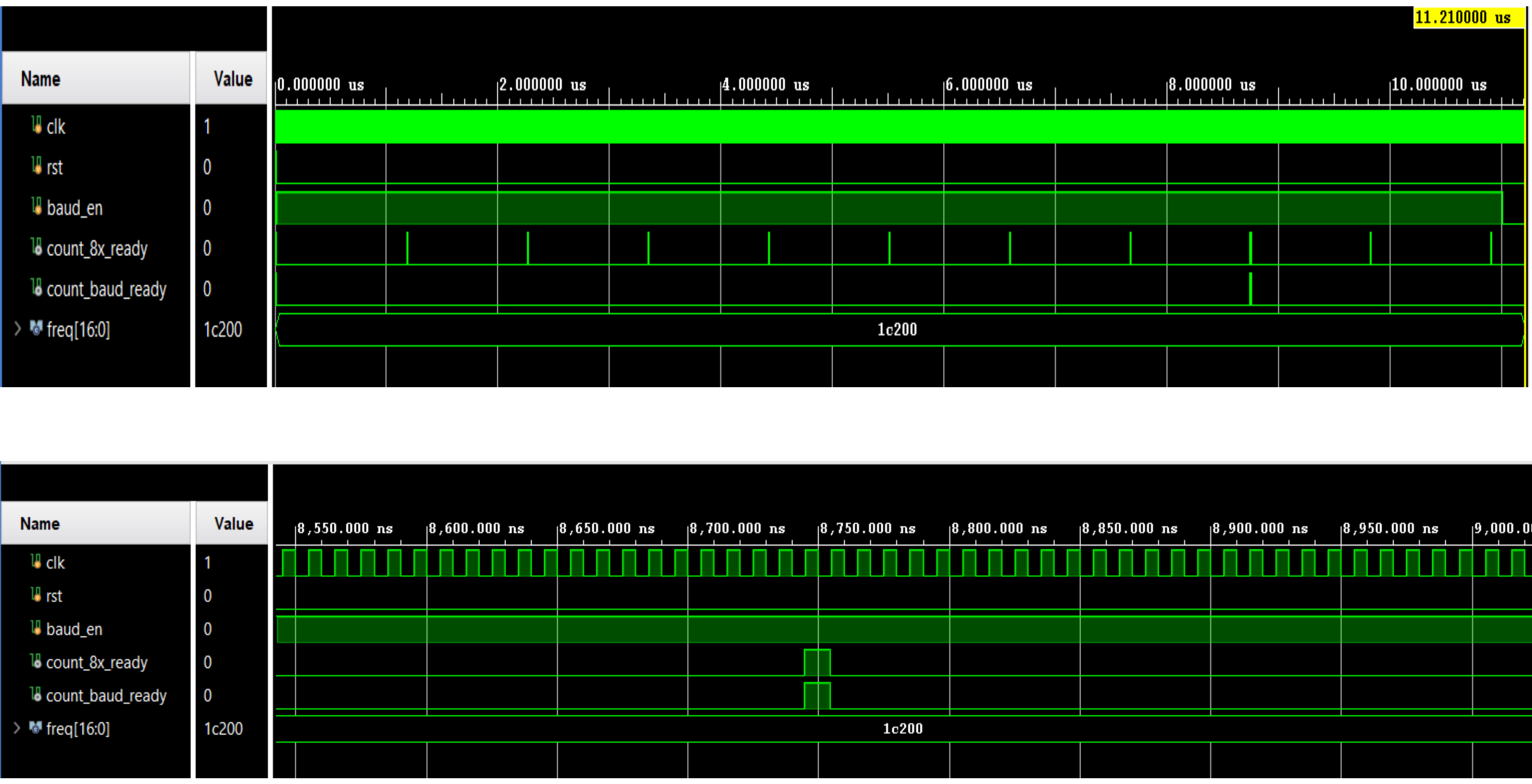
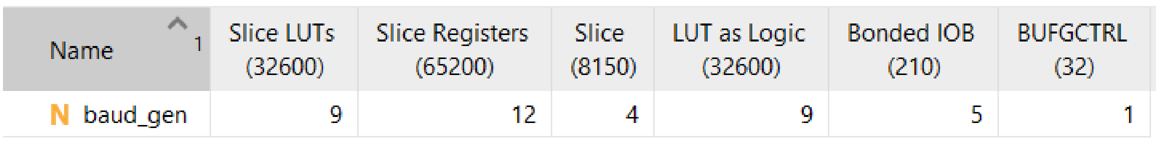
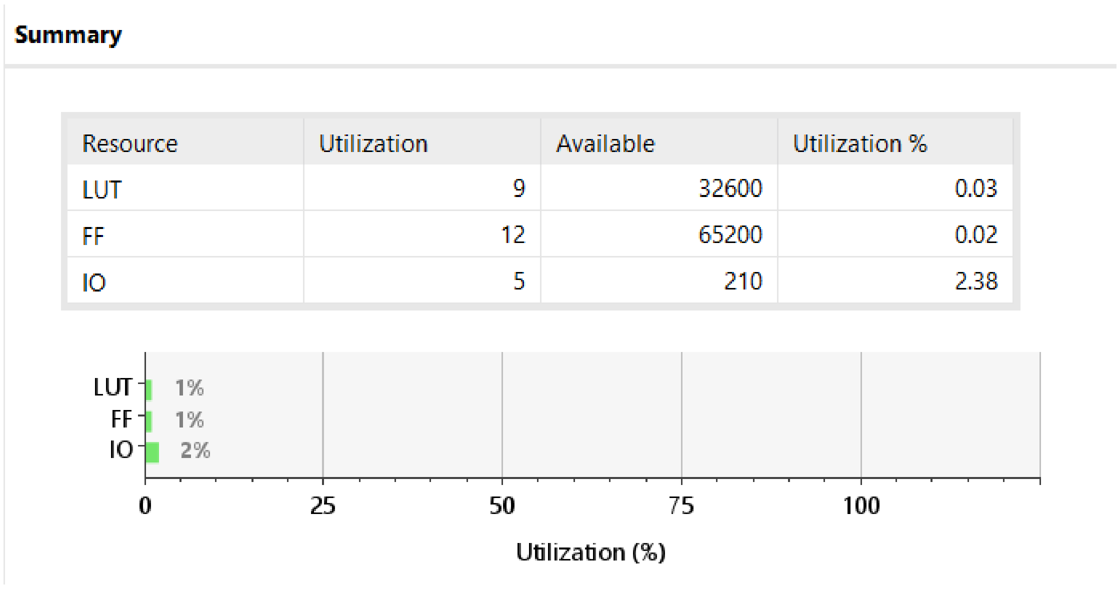


Figure 8: *Post-implementation timing simulation waveform of the baud rate generator, showing the generation of count_8x_ready and count_baud_ready pulses synchronized to the 100 MHz system clock, confirming correct baud and oversampling tick timing under implementation delays.*

3.5. Utilization & Timing Summary Report

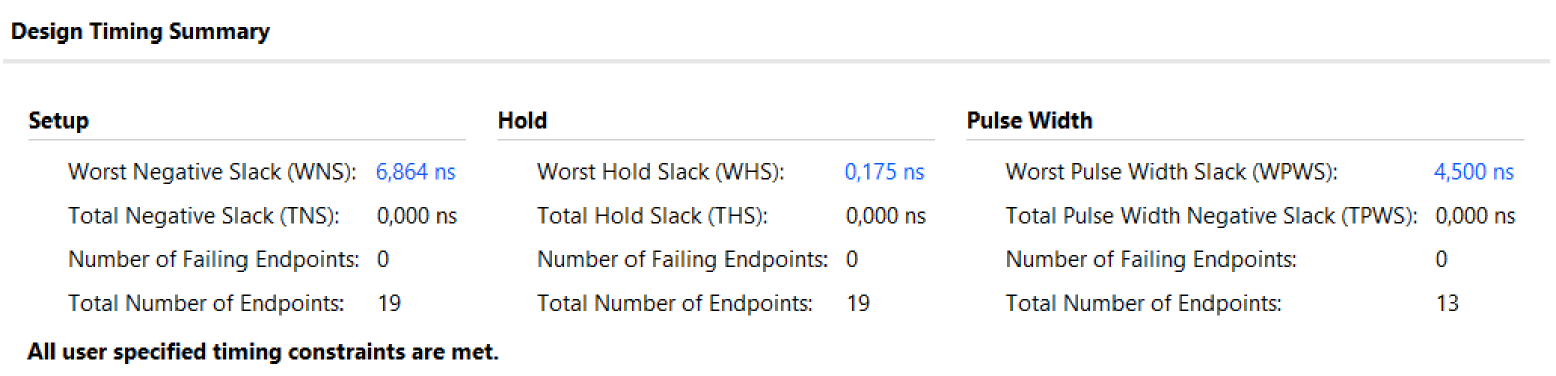


Name	Slice LUTs (32600)	Slice Registers (65200)	Slice (8150)	LUT as Logic (32600)	Bonded IOB (210)	BUFGCTRL (32)
N baud_gen	9	12	4	9	5	1



Summary

Resource	Utilization	Available	Utilization %
LUT	9	32600	0.03
FF	12	65200	0.02
IO	5	210	2.38



Design Timing Summary

Setup		**Hold**		**Pulse Width**	
Worst Negative Slack (WNS):	6,864 ns	Worst Hold Slack (WHS):	0,175 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:	19	Total Number of Endpoints:	19	Total Number of Endpoints:	13

All user specified timing constraints are met.

4. UART Transmitter

The uart_tx module is responsible for transmitting an 8-bit parallel input as a serial UART frame. The frame format used in this project is 1 start bit, 8 data bits (LSB first), and 1 stop bit. The transmitter is implemented using a finite state machine (FSM) with four states: IDLE, START, DATA, STOP.

- IDLE: The TX line is held at logic ‘1’. The module waits for tx_en to become high. When tx_en is asserted, the input byte is latched into an internal register and the baud generator is enabled (baud_en = 1). A one-cycle start pulse is generated to indicate the beginning of a transmission.
- START: The transmitter drives the line low (tx_out = 0) for exactly one baud period to send the start bit.
- DATA: The transmitter sends the data bits sequentially. The least significant bit is transmitted first. The bit progression is controlled by the tick_baud signal (a one-cycle pulse from the baud rate generator). At each tick_baud, the next bit is output and the internal shift register is updated. After 8 bits, the FSM transitions to STOP.
- STOP: The line is driven high (tx_out = 1) for one baud period as the stop bit. After that, the transmitter disables the baud generator, generates a one-cycle done pulse, and returns to IDLE.
  During the transmission process, the busy signal remains high to indicate that the module is actively sending a frame.

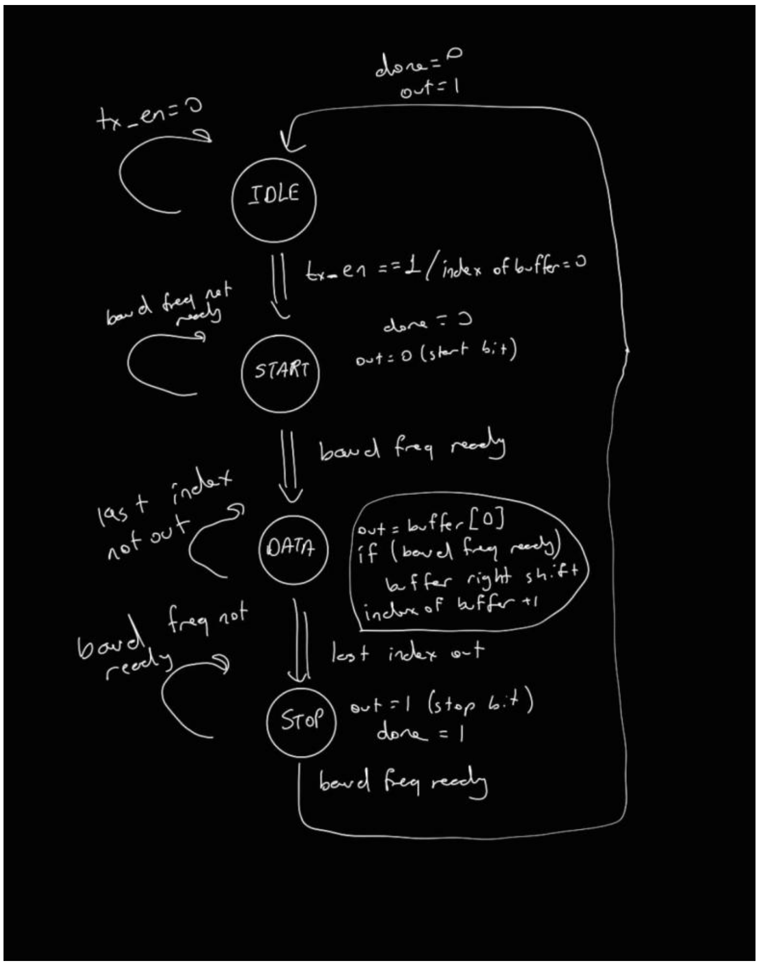
4.1. Design Source

The uart_tx module implements a UART transmitter using a finite state machine. It serializes 8-bit parallel input data into a standard UART frame consisting of one start bit, eight data bits (LSB first), and one stop bit. The transmission is synchronized with an external baud rate generator through the tx_count_baud_ready signal. Control signals start, busy, and done indicate the transmission status.

```
module uart_tx #(
    parameter [16:0] freq = 115200   // Baud rate parameter (not used
directly here)
)(
    input  wire       clk,            // System clock
    input  wire       rst,            // Synchronous reset
    input  wire [7:0] d_in,           // Parallel data to transmit
    input  wire       tx_en,          // Transmit enable signal

    // Interface to baud rate generator
    input  wire       tx_count_baud_ready, // Baud tick: one pulse per bit
time
    output reg        tx_baud_en,            // Enable signal for baud
generator

    output wire       seri_out,       // UART serial output line
    output reg        start,          // Indicates start of transmission
    output reg        busy,           // High while transmitting
    output wire       done            // One-cycle pulse when transmission
completes
);

    reg done_reg;                     // Internal done flag
    reg seri_out_reg;                // Registered serial output
    reg [2:0] bit_in;                 // Bit counter (0 to 7)
    reg [7:0] buffer;                 // Shift register for transmit data

    // FSM state encoding
    localparam IDLE  = 2'b00;         // Idle state
    localparam START = 2'b01;         // Start bit state
    localparam DATA  = 2'b10;         // Data bits state
    localparam STOP  = 2'b11;         // Stop bit state

    reg [1:0] state;                  // Current FSM state

    // Output assignments
    assign seri_out = seri_out_reg;
    assign done     = done_reg;

    // Main FSM and control logic
    always @(posedge clk) begin
        if (rst) begin
            // Reset all internal registers and outputs
            state        <= IDLE;
            done_reg     <= 1'b0;
            start        <= 1'b0;
            busy         <= 1'b0;
            seri_out_reg <= 1'b1;  // UART line is idle high
            tx_baud_en   <= 1'b0;
            bit_in       <= 3'd0;
            buffer       <= 8'h00;
        end else begin
            case (state)

                // -------------------------------
                // IDLE: Wait for tx_en to start
                // -------------------------------
                IDLE: begin
                    done_reg     <= 1'b0;
                    seri_out_reg <= 1'b1;   // Keep line high
                    tx_baud_en   <= 1'b0;   // Disable baud generator
                    busy         <= 1'b0;
                    start        <= 1'b0;
                    if (tx_en) begin
                        state  <= START;   // Go to start bit
                        buffer <= d_in;    // Load data into buffer
                        bit_in <= 3'd0;    // Reset bit counter
                    end
                end

                // --------------------------------
                // START: Send start bit (0)
                // --------------------------------
                START: begin
                    start        <= 1'b1;
                    busy         <= 1'b1;
                    seri_out_reg <= 1'b0;   // Start bit = 0
                    tx_baud_en   <= 1'b1;   // Enable baud generator

                    if (tx_count_baud_ready) begin
                        state  <= DATA;      // Move to data bits on baud
tick
                    end
                end

                // --------------------------------
                // DATA: Send 8 data bits (LSB first)
                // --------------------------------
                DATA: begin
                    start        <= 1'b0;
                    busy         <= 1'b1;
                    seri_out_reg <= buffer[0]; // Output LSB

                    if (tx_count_baud_ready) begin
                        if (bit_in == 3'd7) begin
                            state  <= STOP;   // After last bit, go to stop
                            bit_in <= 3'd0;
                        end else begin
                            buffer <= buffer >> 1; // Shift right
                            bit_in <= bit_in + 1'b1;
                        end
                    end
                end

                // --------------------------------
                // STOP: Send stop bit (1)
                // --------------------------------
                STOP: begin
                    busy         <= 1'b1;
                    seri_out_reg <= 1'b1;   // Stop bit = 1

                    if (tx_count_baud_ready) begin
                        state      <= IDLE; // Return to idle
                        tx_baud_en <= 1'b0; // Disable baud generator
                        busy       <= 1'b0;
                        done_reg   <= 1'b1; // Pulse done
                    end else begin
                        done_reg <= 1'b0;
                    end
                end

                default: state <= IDLE;
            endcase
        end
```

```
    end

endmodule
```

4.2. Finite State Machine



4.3. Simulation Source

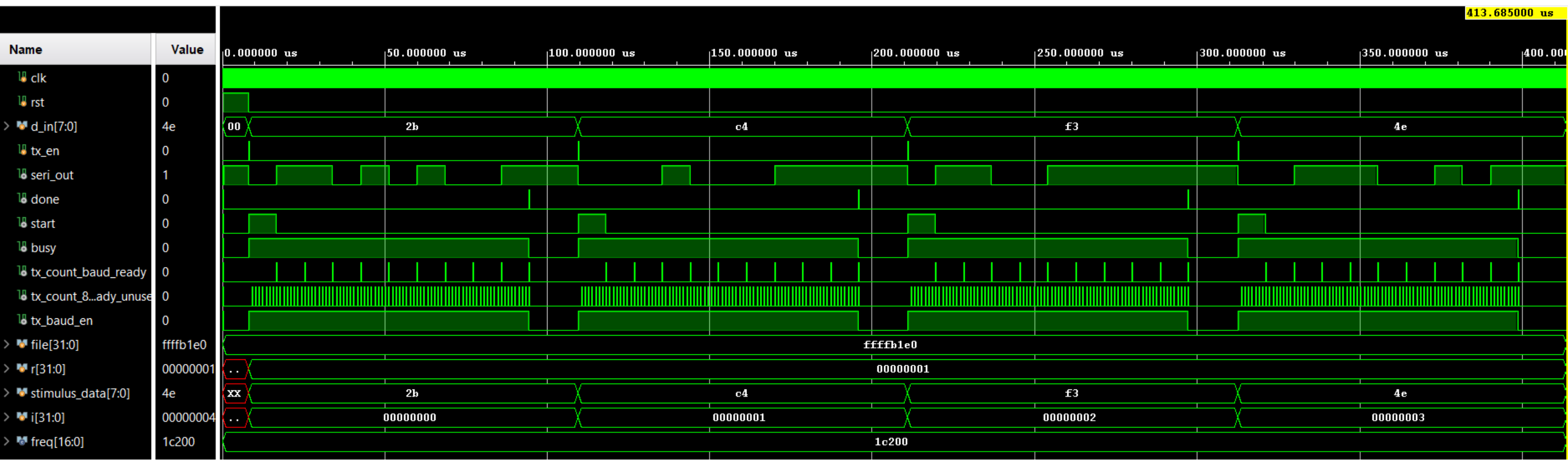
```
module uart_tx_tb;


    // Parameters

    // Baud rate parameter (17 bits required for 115200)
    parameter [16:0] freq = 115200;


    // Testbench Signals

    reg         clk   = 1'b0;       // System clock
    reg         rst   = 1'b0;       // Reset
    reg [7:0]   d_in  = 8'h00;      // Data input to TX
    reg         tx_en = 1'b0;       // Transmit enable


    // Outputs from DUT
```

```verilog
wire seri_out;              // Serial UART output
wire done;                  // Transmission done pulse
wire start;                 // Transmission start indicator
wire busy;                  // TX busy flag

// Baud_gen interface signals for revised uart_tx
wire tx_count_baud_ready;        // Baud tick from baud_gen
wire tx_count_8x_ready_unused;  // 8x tick (unused for TX)
wire tx_baud_en;                // Enable signal from TX to baud_gen

// File I/O variables
integer file, r;
reg [7:0] stimulus_data;    // Data read from file
integer i;


// 100 MHz clock generation (10 ns period)

always #5 clk = ~clk;

// Baud generator instance
// Generates baud ticks for uart_tx
baud_gen #(
    .freq(freq)
) u_baud_tx (
    .clk              (clk),
    .rst              (rst),
    .baud_en          (tx_baud_en),          // Enabled by TX module
    .count_8x_ready   (tx_count_8x_ready_unused), // Not used in TX
    .count_baud_ready (tx_count_baud_ready) // Baud tick for TX
);


// DUT: Revised uart_tx (external baud_gen)
uart_tx #(
    .freq(freq)
) uut (
    .clk                 (clk),
    .rst                 (rst),
    .d_in                (d_in),
    .tx_en               (tx_en),

    .tx_count_baud_ready (tx_count_baud_ready),
    .tx_baud_en          (tx_baud_en),

    .seri_out            (seri_out),
    .start               (start),
    .busy                (busy),
    .done                (done)
);
// Main stimulus process
initial begin
    // Open stimulus file
    // The file must be located in the simulation run directory
    file = $fopen("stimulus.txt", "r");
    if (file == 0) begin
        $display("ERROR: Cannot open stimulus.txt");
        $finish;
    end
```

```
        // Apply reset
        rst   = 1'b1;
        d_in = 8'h00;
        tx_en = 1'b0;
        #8000;                 // Hold reset for 8 us
        rst   = 1'b0;

        // Read and transmit 4 bytes from file
        for (i = 0; i < 4; i = i + 1) begin
            r = $fscanf(file, "%b\n", stimulus_data);
            if (r == 1) begin
                // Apply input data
                d_in = stimulus_data;

                // Generate a one-clock pulse on tx_en
                tx_en = 1'b1;
                #10;
                tx_en = 1'b0;

                // Wait until transmission is complete
                wait (done == 1'b1);

                // Display transmitted data
                $display("TX Sent[%0d] = %b (0x%0h)", i, stimulus_data,
stimulus_data);

                // Wait before sending next byte
                #15000;
            end
        end

        // Close file and finish simulation
        $fclose(file);
        $display("UART TX file-based test completed.");
        $finish;
    end

endmodule
```

I took 4 random data into a text file as each one 8 bits. I used them in the simulation and checked it.
Here they are:
00101011
11000100
11110011
01001110

4.4. Behavioral Simulation Waveform



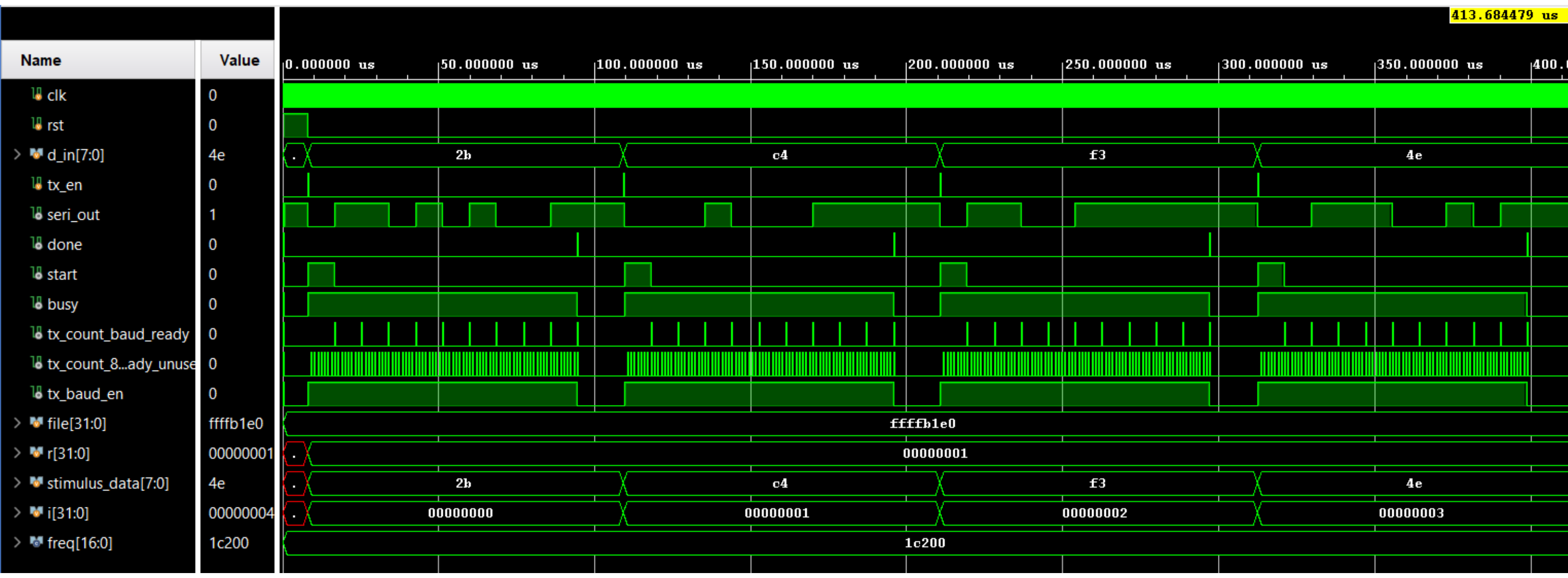
Name	Value
clk	0
rst	0
d_in[7:0]	4e
tx_en	0
seri_out	1
done	0
start	0
busy	0
tx_count_baud_ready	0
tx_count_8..ady_unuse	0
tx_baud_en	0
file[31:0]	ffffb1e0
r[31:0]	00000001
stimulus_data[7:0]	4e
i[31:0]	00000004
freq[16:0]	1c200

Figure 9: *Simulation waveform of the UART transmitter showing the serialization of input bytes (2B, C4, F3, 4E) into seri_out, along with the control signals tx_en, busy, and done, synchronized by the baud rate tick from baud_gen.*

2b = $0010\ 1011_2$ → LSB first: 1 1 0 1 0 1 0 0
c4 = $1100\ 0100_2$ → LSB first: 0 0 1 0 0 0 1 1
f3 = $1111\ 0011_2$ → LSB first: 1 1 0 0 1 1 1 1
4e = $0100\ 1110_2$ → LSB first: 0 1 1 1 0 0 1 0

In the transmitter simulation, the input bytes 0x2B, 0xC4, 0xF3 and 0x4E are applied through the stimulus file. As shown in Figure X, each value is transmitted with a start bit, followed by 8 data bits in LSB-first order, and a stop bit. The busy signal stays high during the frame, while done pulses at the end of each transmission, confirming the correct operation of the UART transmitter.

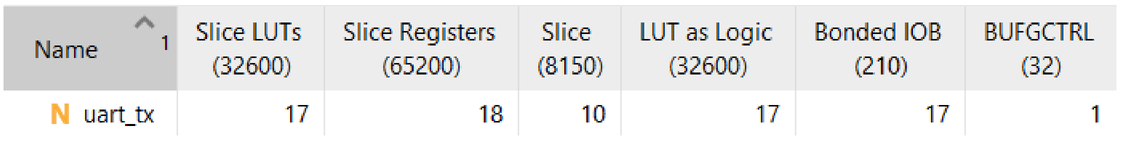
4.5. Post-implementation Timing Waveform



Name	Value
clk	0
rst	0
d_in[7:0]	4e
tx_en	0
seri_out	1
done	0
start	0
busy	0
tx_count_baud_ready	0
tx_count_8...ady_unuse	0
tx_baud_en	0
file[31:0]	ffffb1e0
r[31:0]	00000001
stimulus_data[7:0]	4e
i[31:0]	00000004
freq[16:0]	1c200

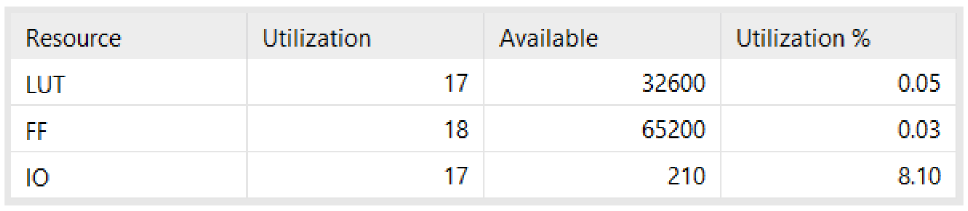
Figure 10*: Post-implementation timing simulation waveform of the UART transmitter, showing correct serialization of the input bytes (2B, C4, F3, 4E) on seri_out under realistic timing delays, and validating proper operation of control signals (tx_en, busy, done) with the generated baud rate ticks.*

4.6. Utilization & Timing Summary Report

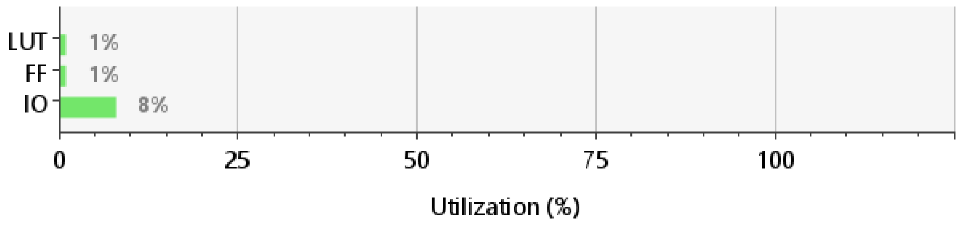


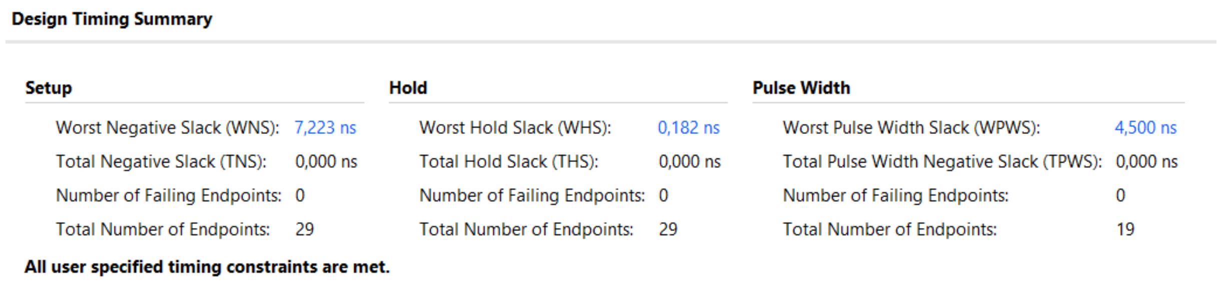
Name	Slice LUTs (32600)	Slice Registers (65200)	Slice (8150)	LUT as Logic (32600)	Bonded IOB (210)	BUFGCTRL (32)
N uart_tx	17	18	10	17	17	1

Summary



Resource	Utilization	Available	Utilization %
LUT	17	32600	0.05
FF	18	65200	0.03
IO	17	210	8.10





Design Timing Summary

Setup

Worst Negative Slack (WNS):	7,223 ns
Total Negative Slack (TNS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	29

Hold

Worst Hold Slack (WHS):	0,182 ns
Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	29

Pulse Width

Worst Pulse Width Slack (WPWS):	4,500 ns
Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	19

All user specified timing constraints are met.

5. UART Receiver

5.1. Design Source

```
module uart_rx #(
    parameter [16:0] freq = 115200
) (
    input  wire        clk,
    input  wire        rst,
    input  wire        d_in,      // Serial RX input line
    input  wire        rx_en,     // Receiver enable

    // Interface to external baud generator
    input  wire        rx_count_8x_ready,    // 8x oversampling tick
    input  wire        rx_count_baud_ready,  // Baud-rate tick
    output reg         rx_baud_en,            // Enable signal to baud_gen

    output reg   [7:0] d_out,    // Received parallel data
    output reg         start,    // Start of reception indicator
    output reg         busy,     // Receiver busy flag
    output wire        done      // Reception done pulse
);

    reg done_reg;
    // ----------------------------------------
    // Sampling and bit counters
    // ----------------------------------------
    reg [2:0] bit_in;          // Bit index (0 to 7)
    reg [2:0] sample_cnt;      // Oversample counter (0 to 7)
    reg [3:0] ones_cnt;        // (Unused here) previously for majority
voting
    reg [2:0] half_cnt;        // Counts 4 ticks for half-bit delay (T/2)

    reg [7:0] shift_reg;       // Shift register to store received bits

    // ----------------------------------------
    // FSM state encoding
    // ----------------------------------------
    localparam IDLE  = 2'b00;
    localparam START = 2'b01;
    localparam DATA  = 2'b10;
    localparam STOP  = 2'b11;

    reg [1:0] state;

    assign done = done_reg;

    // ----------------------------------------
    // UART RX state machine
    // ----------------------------------------
    always @(posedge clk) begin
        if (rst) begin
            // Reset all registers and go to IDLE
            state      <= IDLE;
            d_out      <= 8'h00;
            shift_reg  <= 8'h00;
            start      <= 1'b0;
            busy       <= 1'b0;
            done_reg   <= 1'b0;
            rx_baud_en <= 1'b0;

            bit_in     <= 3'd0;
            sample_cnt <= 3'd0;
            ones_cnt   <= 4'd0;
            half_cnt   <= 3'd0;

        end else begin
            // Default: done is a one-cycle pulse
            done_reg <= 1'b0;

            case (state)
                // --------------------------------
                // IDLE: Wait for start bit
                // --------------------------------
                IDLE: begin
                    start      <= 1'b0;
                    busy       <= 1'b0;
                    rx_baud_en <= 1'b0;

                    bit_in     <= 3'd0;
                    sample_cnt <= 3'd0;
                    ones_cnt   <= 4'd0;
                    half_cnt   <= 3'd0;

                    // Detect falling edge (start bit)
                    if (rx_en && (d_in == 1'b0)) begin
                        state       <= START;
                        start       <= 1'b1;
                        busy        <= 1'b1;
                        rx_baud_en <= 1'b1;   // Enable baud generator
                        half_cnt    <= 3'd0;
                        sample_cnt <= 3'd0;
                    end
                end

                // --------------------------------
                // START: Validate start bit at T/2
                // --------------------------------
                START: begin
                    start <= 1'b1;
                    busy  <= 1'b1;

                    if (rx_count_8x_ready) begin
                        if (half_cnt < 3'd3) begin
                            // Wait half bit time (4 ticks)
                            half_cnt <= half_cnt + 1'b1;
                        end else begin
                            // Sample in the middle of start bit
                            if (d_in == 1'b0) begin
                                // Valid start bit
                                state       <= DATA;
                                start       <= 1'b0;
                                bit_in      <= 3'd0;
                                sample_cnt <= 3'd0;
                            end else begin
                                // False start, return to IDLE
                                state       <= IDLE;
                                start       <= 1'b0;
                                busy        <= 1'b0;
                                rx_baud_en <= 1'b0;
                            end
                        end
                    end
                end

                // --------------------------------
                // DATA: Receive 8 data bits (LSB first)
                // --------------------------------
                DATA: begin
                    busy <= 1'b1;

                    if (rx_count_8x_ready) begin
                        if (sample_cnt == 3'd7) begin
                            // One bit period elapsed -> sample bit
                            shift_reg <= { d_in, shift_reg[7:1] };
                            sample_cnt <= 3'd0;

                            if (bit_in == 3'd7) begin
                                state  <= STOP;   // All bits received
                                bit_in <= 3'd0;
                            end else begin
                                bit_in <= bit_in + 1'b1;
                            end
                        end else begin
                            sample_cnt <= sample_cnt + 1'b1;
                        end
                    end
                end

                // --------------------------------
                // STOP: Wait for stop bit and finish
                // --------------------------------
                STOP: begin
                    busy <= 1'b1;

                    if (rx_count_baud_ready) begin
                        d_out      <= shift_reg;  // Output received byte
                        done_reg   <= 1'b1;       // Pulse done
                        busy       <= 1'b0;
                        rx_baud_en <= 1'b0;       // Disable baud generator
                        state      <= IDLE;
                    end
                end

                default: state <= IDLE;
            endcase
        end
    end

endmodule
```

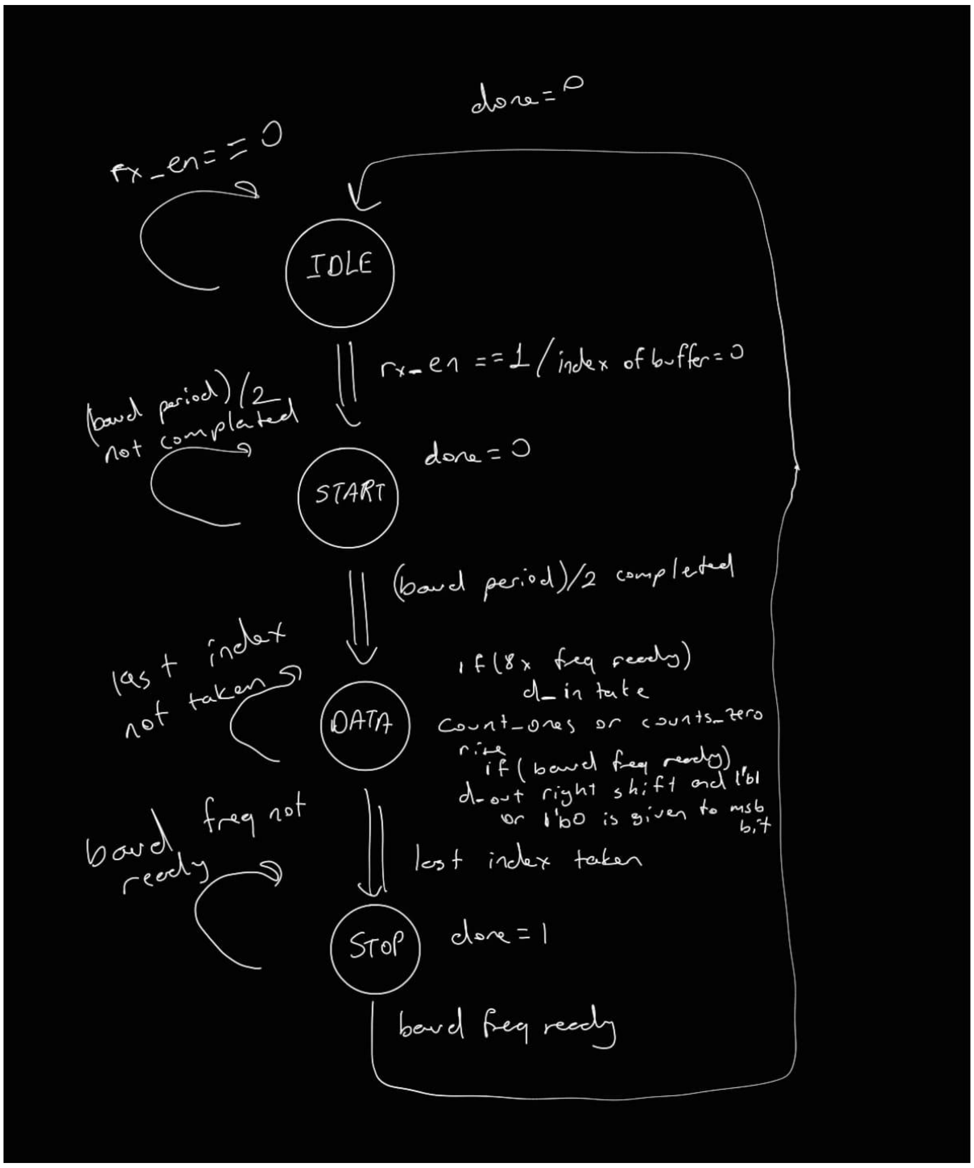
The UART receiver is designed as a finite state machine with IDLE, START, DATA and STOP states. Since the receiver was implemented before the top module, the baud rate generator is instantiated inside the receiver to provide the required baud and 8x baud timing signals. In the final top-level design, this module is later moved and shared at the top level.

In the IDLE state, the receiver waits until rx_en becomes logic high. When enabled and a falling edge is detected on the input, the FSM switches to the START state. In this state, the receiver waits for half of the baud period (T/2) to align sampling to the center of the start bit. This half-bit delay is measured by counting four ticks of the 8x baud clock.

After T/2, the FSM enters the DATA state. Here, the receiver uses the 8x baud tick as a timing reference and waits for one full bit period (eight ticks) for each data bit. At the end of each bit period, the serial input d_in is sampled once and shifted into an internal register. This process is repeated until all eight data bits are collected, as indicated by the bit counter.

After all data bits are received, the FSM switches to the STOP state. In this state, the receiver waits for one baud period corresponding to the stop bit interval and then generates a one-cycle done pulse to indicate that a byte has been received. Finally, the FSM returns to the IDLE state and waits for the next frame.

5.2. Finite State Machine



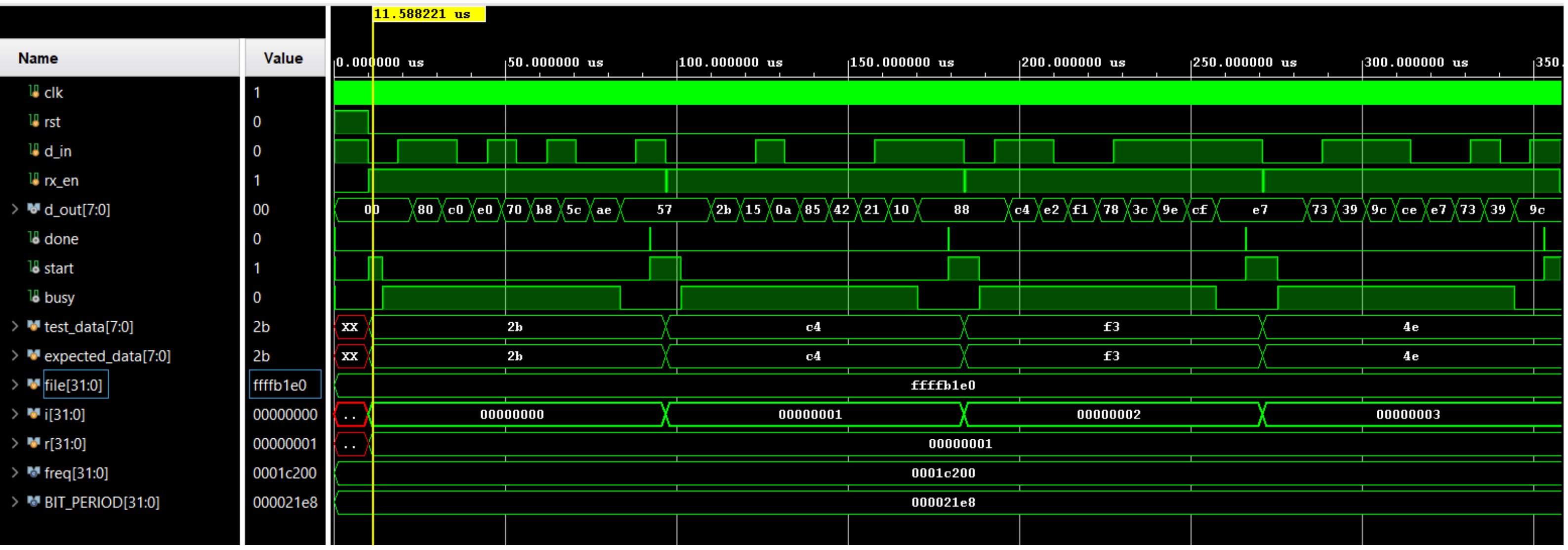
done = 0
rx_en == 0
IDLE
rx_en == 1 / index of buffer = 0
(baud period)/2 not completed
START
done = 0
(baud period)/2 completed
last index not taken
DATA
if (8x freq ready) d_in take count_ones or counts_zero rise
if (baud freq ready) d_out right shift and 1'b1 or 1'b0 is given to msb bit
last index taken
baud freq not ready
STOP
done = 1
baud freq ready

5.3. Simulation Source

```
module uart_rx_tb;

    // ----------------------------------------
    // Parameters
    // ----------------------------------------
    parameter [16:0] freq = 115200;              // UART baud rate
(parameter to DUT)
    localparam integer CLK_PERIOD     = 10;      // 100 MHz clock period in
ns
    localparam integer BIT_PERIOD     = 8680;    // Bit time in ns for
115200 baud
    localparam integer TICK_8X_PERIOD = BIT_PERIOD/8; // 8x oversampling
tick period

    // ----------------------------------------
    // Testbench Signals
    // ----------------------------------------
    reg clk  = 1'b0;          // System clock
    reg rst  = 1'b0;          // Reset
    reg d_in = 1'b1;          // UART RX line (idle = high)
    reg rx_en = 1'b0;         // Receiver enable

    // ----------------------------------------
    // "baud_gen" handshake signals (generated by TB)
    // These emulate the baud_gen outputs used by uart_rx
    // ----------------------------------------
    reg  rx_count_8x_ready   = 1'b0;  // 8x tick pulse
    reg  rx_count_baud_ready = 1'b0;  // baud tick pulse (1x)
    wire rx_baud_en;                 // enable from DUT to start/stop tick
generation

    // ----------------------------------------
    // Outputs from DUT
    // ----------------------------------------
    wire [7:0] d_out;         // Received byte
    wire done;                // Done pulse from receiver
    wire start;               // Start detection indicator
    wire busy;                // Receiver busy flag

    // ----------------------------------------
    // Comparison signals (test_data vs expected_data)
    // ----------------------------------------
    reg [7:0] test_data;      // Byte read from file and transmitted on d_in
    reg [7:0] expected_data; // Expected received value (should match
d_out)

    // ----------------------------------------
    // File I/O variables
    // ----------------------------------------
    integer file, r;
    integer i;

    // ----------------------------------------
    // Clock Generation (100 MHz)
    // ----------------------------------------
    always #(CLK_PERIOD/2) clk = ~clk;

    // ----------------------------------------
    // Instantiate DUT (uart_rx)
    // ----------------------------------------
    uart_rx #(
        .freq(freq)
    ) uut (
        .clk                 (clk),
        .rst                 (rst),
        .d_in                (d_in),
        .rx_en               (rx_en),
        .rx_count_8x_ready   (rx_count_8x_ready),
        .rx_count_baud_ready (rx_count_baud_ready),
        .rx_baud_en          (rx_baud_en),
        .d_out               (d_out),
        .start               (start),
        .busy                (busy),
        .done                (done)
    );

    // ------------------------------------------------------------
    // Generate rx_count_8x_ready pulses when rx_baud_en is high
    // Also generate rx_count_baud_ready once every 8 pulses
    // This emulates a baud generator used by the RX module.
    // ----------------------------------------------------------
    integer tick8_cnt = 0;

    initial begin
        rx_count_8x_ready   = 1'b0;
        rx_count_baud_ready = 1'b0;
        tick8_cnt           = 0;

        forever begin
            // Wait for the 8x tick interval
            #(TICK_8X_PERIOD);

            if (rx_baud_en) begin
                // Create a pulse long enough so the DUT will not miss it
                rx_count_8x_ready = 1'b1;
                #(CLK_PERIOD);
                rx_count_8x_ready = 1'b0;

                // After 8x pulses, create one baud tick pulse
                if (tick8_cnt == 7) begin
                    tick8_cnt = 0;
                    rx_count_baud_ready = 1'b1;
                    #(CLK_PERIOD);
                    rx_count_baud_ready = 1'b0;
                end else begin
                    tick8_cnt = tick8_cnt + 1;
                end
            end else begin
                // If RX disables baud generation, reset tick
counters/pulses
                tick8_cnt           = 0;
                rx_count_8x_ready   = 1'b0;
                rx_count_baud_ready = 1'b0;
            end
        end
    end

    // ----------------------------------------
    // Task: Send one UART frame on d_in
    // Frame format: start(0) + 8 data bits LSB-first + stop(1)
    // ----------------------------------------
    task send_uart_byte;
        input [7:0] data;
        begin
            // Start bit
            d_in = 1'b0;
            #(BIT_PERIOD);

            // Data bits (LSB first)
            for (i = 0; i < 8; i = i + 1) begin
                d_in = data[i];
                #(BIT_PERIOD);
            end

            // Stop bit
            d_in = 1'b1;
            #(BIT_PERIOD);
        end
    endtask

    // ----------------------------------------
    // Main test sequence
    // - Apply reset
    // - Enable RX
    // - Read bytes from stimulus.txt
    // - Send each byte as a UART frame
    // ----------------------------------------
    initial begin
        // Initial values
        d_in  = 1'b1;
        rst   = 1'b1;
        rx_en = 1'b0;

        test_data     = 8'h00;
        expected_data = 8'h00;

        // Hold reset briefly
        #100;
        rst  = 1'b0;
        rx_en = 1'b1;

        // Idle line before first frame (helps receiver start clean)
        #(3*BIT_PERIOD);

        // Open stimulus file (must exist in simulation run directory)
        file = $fopen("stimulus.txt", "r");
        if (file == 0) begin
            $display("ERROR: Cannot open stimulus.txt");
            $finish;
        end

        // Read until end-of-file; stimulus file expected to be binary (%b)
        while (!$feof(file)) begin
            r = $fscanf(file, "%b\n", test_data);

            if (r == 1) begin
                // Set expected value for comparison when done asserts
                expected_data = test_data;

                // Send UART frame
                $display("Sending byte: bin=%b hex=%h", test_data,
test_data);
                send_uart_byte(test_data);

                // Small gap between frames
                #(3*BIT_PERIOD);
            end
        end

        // Close file and finish simulation
        $fclose(file);
        #20000;
        $finish;
    end

    // ----------------------------------------
    // Monitor: Compare received output with expected_data
    // Prints PASS/FAIL whenever done pulse occurs
    // ----------------------------------------
    always @(posedge clk) begin
        if (done) begin
            if (d_out == expected_data)
                $display("PASS @%t : d_out=%h matches expected=%h", $time,
d_out, expected_data);
            else
                $display("FAIL @%t : d_out=%h expected=%h", $time, d_out,
expected_data);
        end
    end

endmodule
```

The UART receiver is designed as a finite state machine with the states **IDLE**, **START**, **DATA**, and **STOP**. Since the receiver was implemented before the top module, the baud-rate generator is instantiated inside the receiver to provide both the baud and 8x baud timing signals. In the final top-level design, this module is later moved to the top level and shared between the transmitter and receiver.
In the **IDLE** state, the receiver remains inactive until rx_en is asserted. When enabled and a falling edge is detected on the serial input, indicating the start bit, the FSM switches to the **START** state. In this state, the receiver waits for half of the baud period (T/2) in order to align the sampling instant to the center of the start bit. This half-bit delay is measured by counting four ticks of the 8x baud clock.
After T/2, the FSM enters the **DATA** state. Here, the receiver uses the 8x baud tick as a timing reference and waits for one full bit period (eight ticks) for each data bit. At the end of each bit period, the serial input d_in is sampled once and shifted into an internal register. This process is repeated until all eight data bits are collected, as indicated by the bit counter.
After all data bits are received, the FSM switches to the **STOP** state. In this state, the receiver waits for one baud period corresponding to the stop bit interval and then generates a one-cycle done pulse to indicate that a byte has been received. Finally, the FSM returns to the **IDLE** state and waits for the next frame.

5.4. Simulation Waveform



Name	Value
clk	1
rst	0
d_in	0
rx_en	1
d_out[7:0]	00
done	0
start	1
busy	0
test_data[7:0]	2b
expected_data[7:0]	2b
file[31:0]	ffffb1e0
i[31:0]	00000000
r[31:0]	00000001
freq[31:0]	0001c200
BIT_PERIOD[31:0]	000021e8

In the receiver simulation, serial data frames are applied to the d_in input using the testbench while rx_en is asserted. As shown in Figure X, the receiver detects the falling edge of the start bit and waits for half a bit duration (T/2) in order to align sampling in time. Then, in the DATA state, the receiver uses the 8x baud tick as a timing reference and samples the serial input **once per bit period** (after 8 ticks) to reconstruct each data bit, shifting the sampled value into an internal register until 8 data bits are collected.
After the 8-bit word is reconstructed, the receiver waits one baud period corresponding to the stop bit and produces a one-cycle done pulse. The parallel output d_out matches the expected values (0x2B, 0xC4, 0xF3, and 0x4E), verifying correct operation of the UART receiver.

5.5. Post-implementation Timing Simulation Waveform

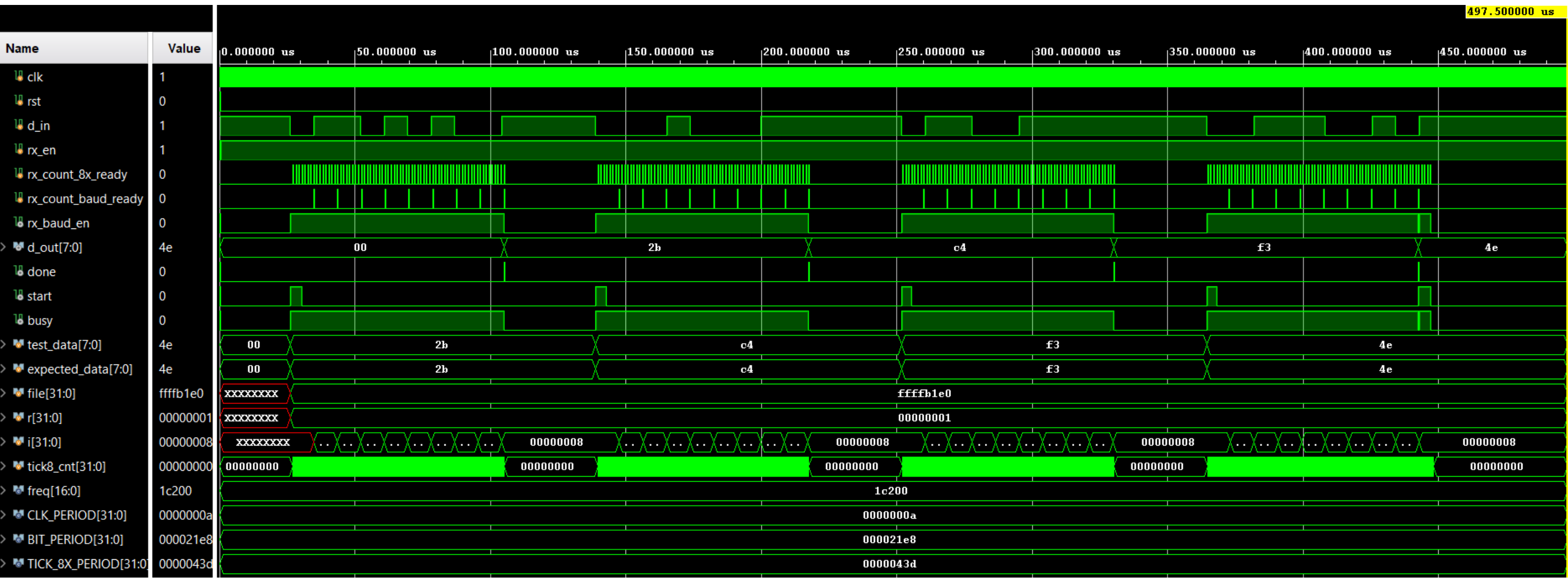
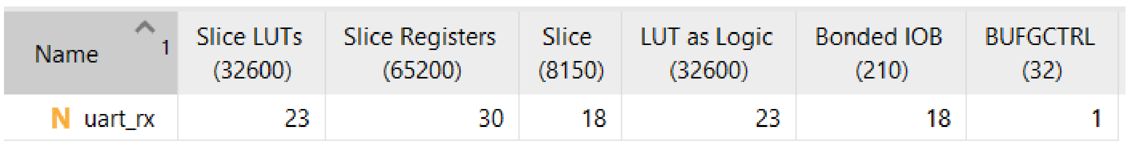
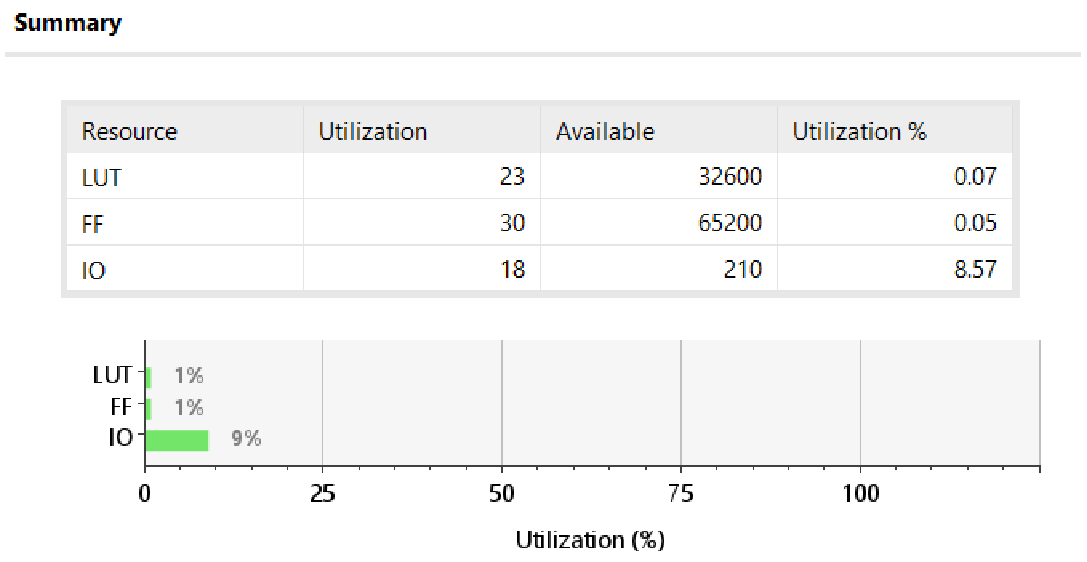


Figure 12: *Post-implementation timing simulation waveform of the UART receiver, showing the reception of serial frames on d_in, oversampling using rx_count_8x_ready, and correct reconstruction of the parallel output d_out. The control signals start, busy, and done confirm proper frame detection and completion under realistic implementation delays.*

5.6. Utilization & Timing Summary Reports

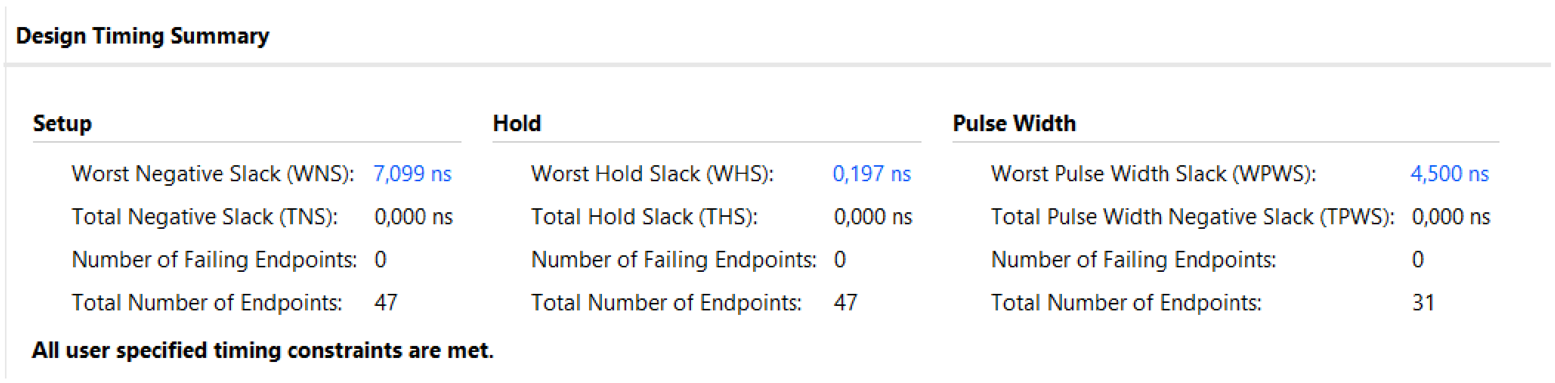


Name	Slice LUTs (32600)	Slice Registers (65200)	Slice (8150)	LUT as Logic (32600)	Bonded IOB (210)	BUFGCTRL (32)
N uart_rx	23	30	18	23	18	1



Summary

Resource	Utilization	Available	Utilization %
LUT	23	32600	0.07
FF	30	65200	0.05
IO	18	210	8.57



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	7,099 ns	Worst Hold Slack (WHS):	0,197 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:	47	Total Number of Endpoints:	47	Total Number of Endpoints:	31

All user specified timing constraints are met.

6. TOP Module

6.1. Design Source

```
// Top-level UART module
// Integrates one baud generator, one UART transmitter, and one UART
receiver
// TX and RX share the same baud_gen instance as required by the design
module uart_top #(
    parameter [16:0] freq = 115200      // UART baud rate
)(
    input  wire       clk,              // System clock (e.g., 100 MHz)
    input  wire       rst,              // Global reset

    // ----------------------------------------
    // Control signals
    // ----------------------------------------
    input  wire       tx_en,             // Enable transmission
    input  wire       rx_en,             // Enable reception

    // ----------------------------------------
    // Parallel TX input
    // ----------------------------------------
    input  wire [7:0] tx_d_in,           // Byte to be transmitted

    // ----------------------------------------
    // Serial lines
    // ----------------------------------------
    input  wire       rx_serial_in,      // Serial input from UART line
    output wire       tx_serial_out,     // Serial output to UART line

    // ----------------------------------------
    // Parallel RX output
    // ----------------------------------------
    output wire [7:0] rx_d_out,          // Received byte

    // ----------------------------------------
    // Status signals - TX
    // ----------------------------------------
    output wire       tx_start,          // TX start indicator
    output wire       tx_busy,           // TX busy flag
    output wire       tx_done,           // TX done pulse

    // ----------------------------------------
    // Status signals - RX
    // ----------------------------------------
    output wire       rx_start,          // RX start indicator
    output wire       rx_busy,           // RX busy flag
    output wire       rx_done            // RX done pulse
);

    // ----------------------------------------
    // Internal wiring between baud_gen, TX, and RX
    // ----------------------------------------
    wire count_8x_ready;         // 8x oversampling tick from baud_gen
    wire count_baud_ready;      // Baud-rate tick from baud_gen

    wire tx_baud_en;            // TX requests baud generator enable
    wire rx_baud_en;            // RX requests baud generator enable

    // Enable baud_gen when either TX or RX is active
    wire baud_en_top;
    assign baud_en_top = tx_baud_en | rx_baud_en;

    // ----------------------------------------
    // Single baud generator instance
    // Shared by both transmitter and receiver
    // ----------------------------------------
    baud_gen #(.freq(freq)) u_baud (
        .clk              (clk),
        .rst              (rst),
        .baud_en          (baud_en_top),
        .count_8x_ready   (count_8x_ready),
        .count_baud_ready (count_baud_ready)
    );

    // ----------------------------------------
    // UART Transmitter instance
    // ----------------------------------------
    uart_tx #(.freq(freq)) u_tx (
        .clk                 (clk),
        .rst                 (rst),
        .d_in                (tx_d_in),
        .tx_en               (tx_en),
        .tx_count_baud_ready (count_baud_ready),
        .tx_baud_en          (tx_baud_en),
        .seri_out            (tx_serial_out),
        .start               (tx_start),
        .busy                (tx_busy),
        .done                (tx_done)
    );

    // ----------------------------------------
    // UART Receiver instance
    // ----------------------------------------
    uart_rx #(.freq(freq)) u_rx (
        .clk                 (clk),
        .rst                 (rst),
        .d_in                (rx_serial_in),
        .rx_en               (rx_en),
        .rx_count_8x_ready   (count_8x_ready),
        .rx_count_baud_ready (count_baud_ready),
        .rx_baud_en          (rx_baud_en),
        .d_out               (rx_d_out),
        .start               (rx_start),
        .busy                (rx_busy),
        .done                (rx_done)
    );

endmodule
```

The uart_top module integrates the UART transmitter, UART receiver, and a single shared baud rate generator into a complete UART communication system. The transmitter and receiver operate independently but share the same baud generator, which is enabled whenever either the transmitter or the receiver requests baud timing (baud_en_top = tx_baud_en | rx_baud_en). Parallel input data are serialized by the transmitter and driven onto the tx_serial_out line, while incoming serial data on rx_serial_in are decoded by the receiver and presented as parallel output on rx_d_out. Status signals indicate the start, busy, and completion of both transmission and reception, allowing external control and monitoring of UART operations.

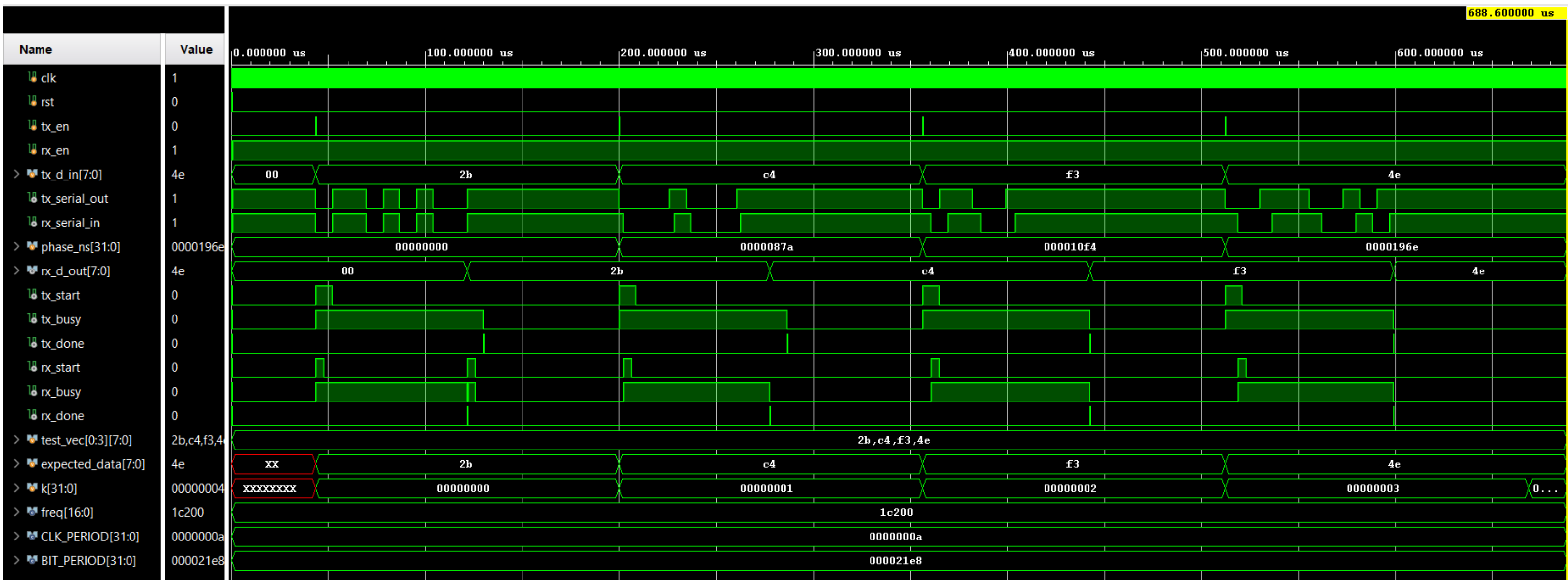
6.2. Simulation Source

```verilog
// Testbench for uart_top
// Performs loopback testing by connecting TX output to RX input
// Applies different phase delays and checks received data automatically
module uart_top_tb;

    // ----------------------------------------
    // Parameters
    // ----------------------------------------
    parameter [16:0] freq = 115200;      // UART baud rate
    localparam integer CLK_PERIOD = 10;  // 100 MHz clock period (ns)
    localparam integer BIT_PERIOD = 8680;// Bit period for 115200 baud (ns)

    // ----------------------------------------
    // Clock and reset
    // ----------------------------------------
    reg clk = 1'b0;
    reg rst = 1'b0;

    // ----------------------------------------
    // TX / RX control and data
    // ----------------------------------------
    reg tx_en = 1'b0;            // Transmit enable
    reg rx_en = 1'b0;            // Receive enable
    reg [7:0] tx_d_in = 8'h00;   // Parallel TX input data

    // ----------------------------------------
    // Serial connection
    // ----------------------------------------
    wire tx_serial_out;          // Serial output from TX

    // RX input is delayed version of TX output
    wire rx_serial_in;
    integer phase_ns;           // Programmable phase delay in ns

    // ----------------------------------------
    // Outputs / status from DUT
    // ----------------------------------------
    wire [7:0] rx_d_out;         // Parallel RX output
    wire tx_start, tx_busy, tx_done;
    wire rx_start, rx_busy, rx_done;

    // ----------------------------------------
    // Test vectors and expected data
    // ----------------------------------------
    reg [7:0] test_vec [0:3];    // Test bytes to transmit
    reg [7:0] expected_data;    // Expected RX output
    integer k;

    // ----------------------------------------
    // Clock generation (100 MHz)
    // ----------------------------------------
    always #(CLK_PERIOD/2) clk = ~clk;

    // ----------------------------------------
    // Apply phase delay between TX and RX (loopback)
    // ----------------------------------------
    assign #(phase_ns) rx_serial_in = tx_serial_out;

    // ----------------------------------------
    // DUT: uart_top instance
    // ----------------------------------------
    uart_top #(.freq(freq)) dut (
        .clk           (clk),
        .rst           (rst),
        .tx_en         (tx_en),
        .rx_en         (rx_en),
        .tx_d_in       (tx_d_in),
        .rx_serial_in  (rx_serial_in),
        .tx_serial_out(tx_serial_out),
        .rx_d_out      (rx_d_out),
        .tx_start      (tx_start),
        .tx_busy       (tx_busy),
        .tx_done       (tx_done),
        .rx_start      (rx_start),
        .rx_busy       (rx_busy),
        .rx_done       (rx_done)
    );

    // ----------------------------------------
    // Task: Send one byte through TX
    // tx_en is asserted for exactly one full clock cycle
    // and aligned to clock edges so uart_tx cannot miss it
    // ----------------------------------------
    task send_byte;
        input [7:0] b;
        begin
            $display("send_byte called: %h at %t", b, $time);

            // Drive data and assert tx_en before a rising edge
            @(negedge clk);
            tx_d_in <= b;
            tx_en   <= 1'b1;

            // Keep tx_en high for one full clock cycle
            @(negedge clk);
            tx_en   <= 1'b0;

            $display("tx_en deasserted at %t", $time);
        end
    endtask

    // ----------------------------------------
    // Main stimulus process
    // ----------------------------------------
    initial begin
        // Initialize test vectors
        test_vec[0] = 8'h2B;
        test_vec[1] = 8'hC4;
        test_vec[2] = 8'hF3;
        test_vec[3] = 8'h4E;

        // Initial conditions
        rst = 1'b1;
        tx_en = 1'b0;
        rx_en = 1'b0;
        phase_ns = 0;

        // Hold reset for some cycles
        #(20*CLK_PERIOD);
        rst = 1'b0;

        // Enable receiver
        rx_en = 1'b1;

        // Small idle gap before first transmission
        #(5*BIT_PERIOD);

        // Loop through all test vectors
        for (k = 0; k < 4; k = k + 1) begin
            // Change phase between TX and RX to test robustness
            phase_ns = (k * (BIT_PERIOD/4)) % BIT_PERIOD;
            expected_data = test_vec[k];

            $display("---- Frame %0d : sending %h, phase_ns=%0d ----",
                      k, expected_data, phase_ns);

            // Start transmission
            send_byte(expected_data);

            // Wait long enough for TX and RX to complete
            #(15*BIT_PERIOD);

            // Compare received data with expected value
            if (rx_d_out == expected_data)
                $display("PASS: rx_d_out=%h matches expected=%h @%t",
                          rx_d_out, expected_data, $time);
            else
                $display("FAIL: rx_d_out=%h expected=%h @%t",
                          rx_d_out, expected_data, $time);

            // Gap before next frame
            #(3*BIT_PERIOD);
        end

        // Finish simulation
        #20000;
        $finish;
    end

endmodule
```

The uart_top_tb testbench generates the system clock and reset, and applies four predefined test bytes (0x2B, 0xC4, 0xF3, 0x4E) to the UART transmitter through an internal test vector array. For each transmitted frame, the receiver output is compared against the expected value and pass/fail messages are reported.

6.3. Simulation Waveform

6.3.1. Behavioral Simulation:



Name	Value
clk	1
rst	0
tx_en	0
rx_en	1
tx_d_in[7:0]	4e
tx_serial_out	1
rx_serial_in	1
phase_ns[31:0]	0000196e
rx_d_out[7:0]	4e
tx_start	0
tx_busy	0
tx_done	0
rx_start	0
rx_busy	0
rx_done	0
test_vec[0:3][7:0]	2b,c4,f3,4
expected_data[7:0]	4e
k[31:0]	00000004
freq[16:0]	1c200
CLK_PERIOD[31:0]	0000000a
BIT_PERIOD[31:0]	000021e8

Figure 13: *Behavioral simulation waveform of the uart_top module showing loopback operation, where transmitted bytes (0x2B, 0xC4, 0xF3, 0x4E) are correctly received at the RX output with corresponding start, busy, and done signals.*

Figure 13 shows the behavioral simulation results of the uart_top module configured in loopback mode. The input bytes are applied to the transmitter through tx_d_in, and transmission is initiated by asserting tx_en. For each test case, the transmitter generates a UART frame consisting of a start bit, eight data bits, and a stop bit, which appears on the serial line tx_serial_out. This serial stream is directly fed into the receiver input. When a complete frame is received, the receiver reconstructs the parallel byte at rx_d_out and asserts a one-cycle rx_done signal. The waveform demonstrates that for all transmitted values (0x2B, 0xC4, 0xF3, and 0x4E), the received data matches the transmitted data, confirming correct end-to-end operation of the UART loopback system..

6.3.2. Post-implementation Timing Simulation

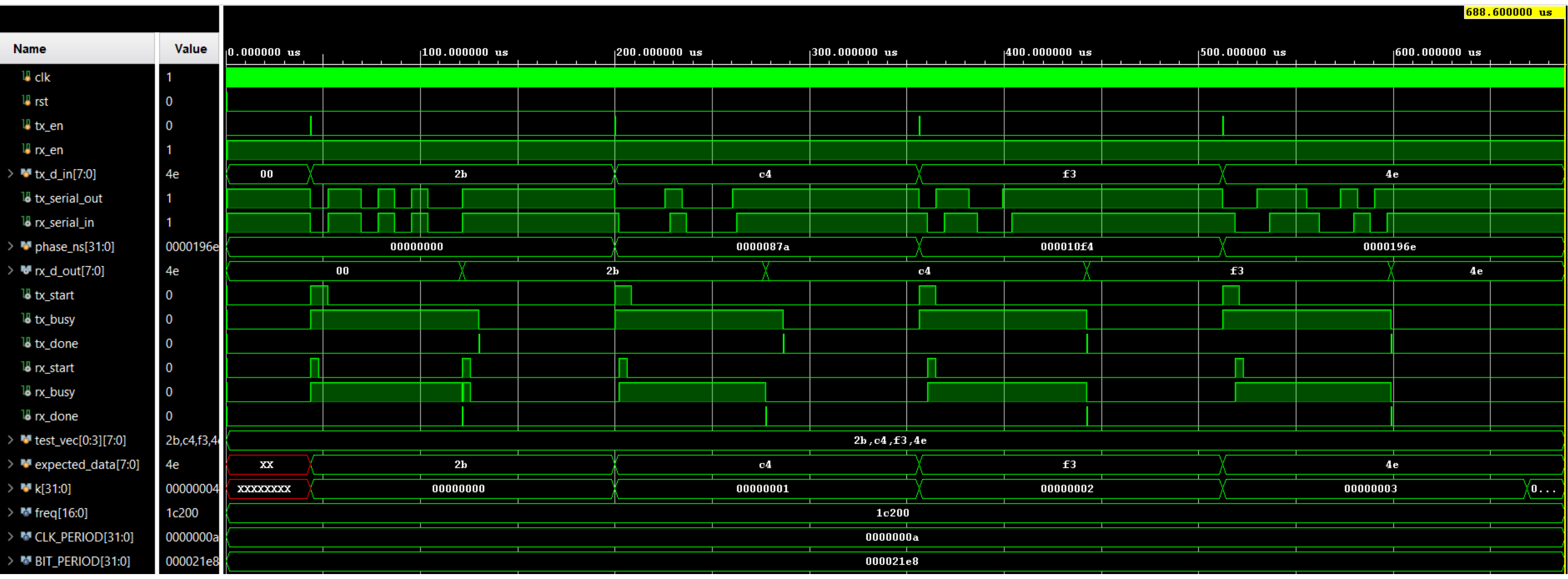
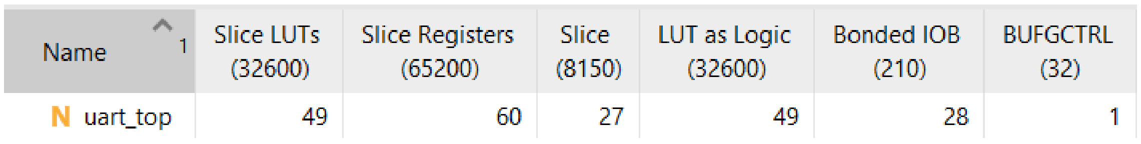


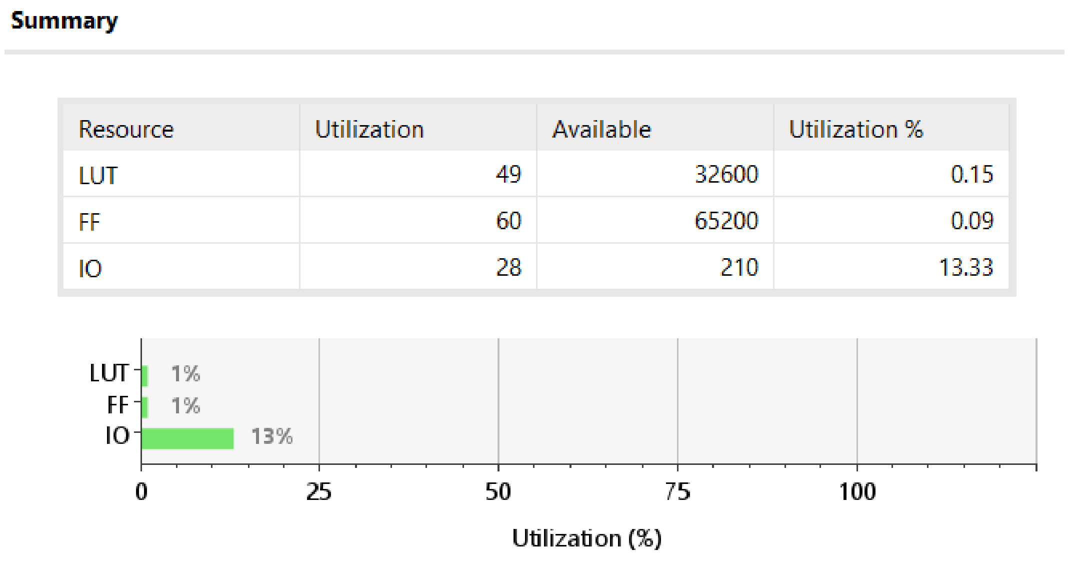
Figure 14: *Post-implementation timing simulation of the UART top module showing correct loopback operation with gate-level delays.*

Figure 14 shows the post-implementation timing simulation results of the uart_top module after place-and-route. In this simulation, the synthesized and routed netlist is used together with timing delays, providing a realistic representation of hardware behavior. Despite the presence of propagation delays, the transmitter successfully generates UART frames on the serial line tx_serial_out, and the receiver correctly reconstructs the transmitted bytes at rx_d_out. The tx_done and rx_done signals are asserted as one-cycle pulses for each frame, indicating proper completion of transmission and reception. The results demonstrate that the UART top module operates correctly not only at the behavioral level but also after implementation with actual timing effects.

6.4. Utilization & Timing Summary Reports

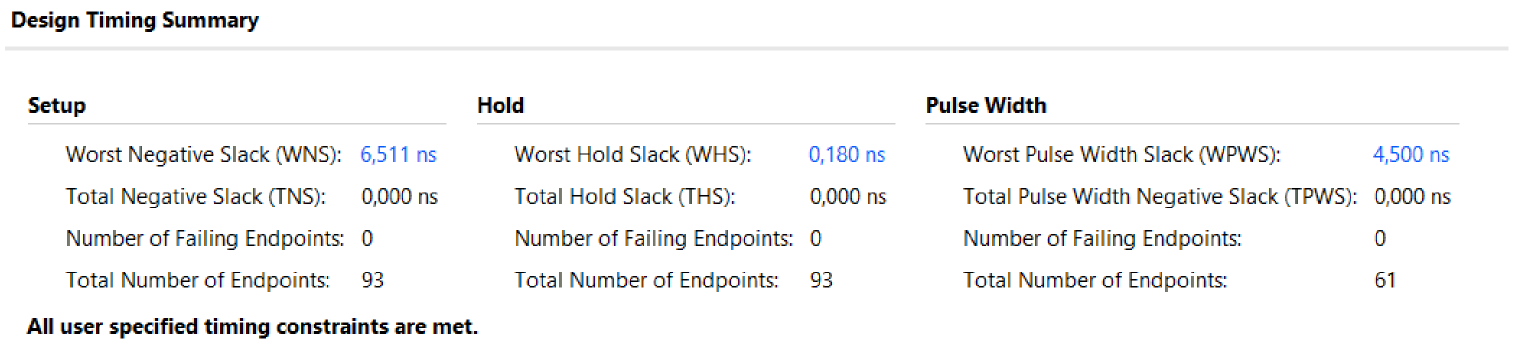


Name	Slice LUTs (32600)	Slice Registers (65200)	Slice (8150)	LUT as Logic (32600)	Bonded IOB (210)	BUFGCTRL (32)
uart_top	49	60	27	49	28	1



Summary

Resource	Utilization	Available	Utilization %
LUT	49	32600	0.15
FF	60	65200	0.09
IO	28	210	13.33



Design Timing Summary

Setup

Worst Negative Slack (WNS):	6,511 ns
Total Negative Slack (TNS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	93

Hold

Worst Hold Slack (WHS):	0,180 ns
Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	93

Pulse Width

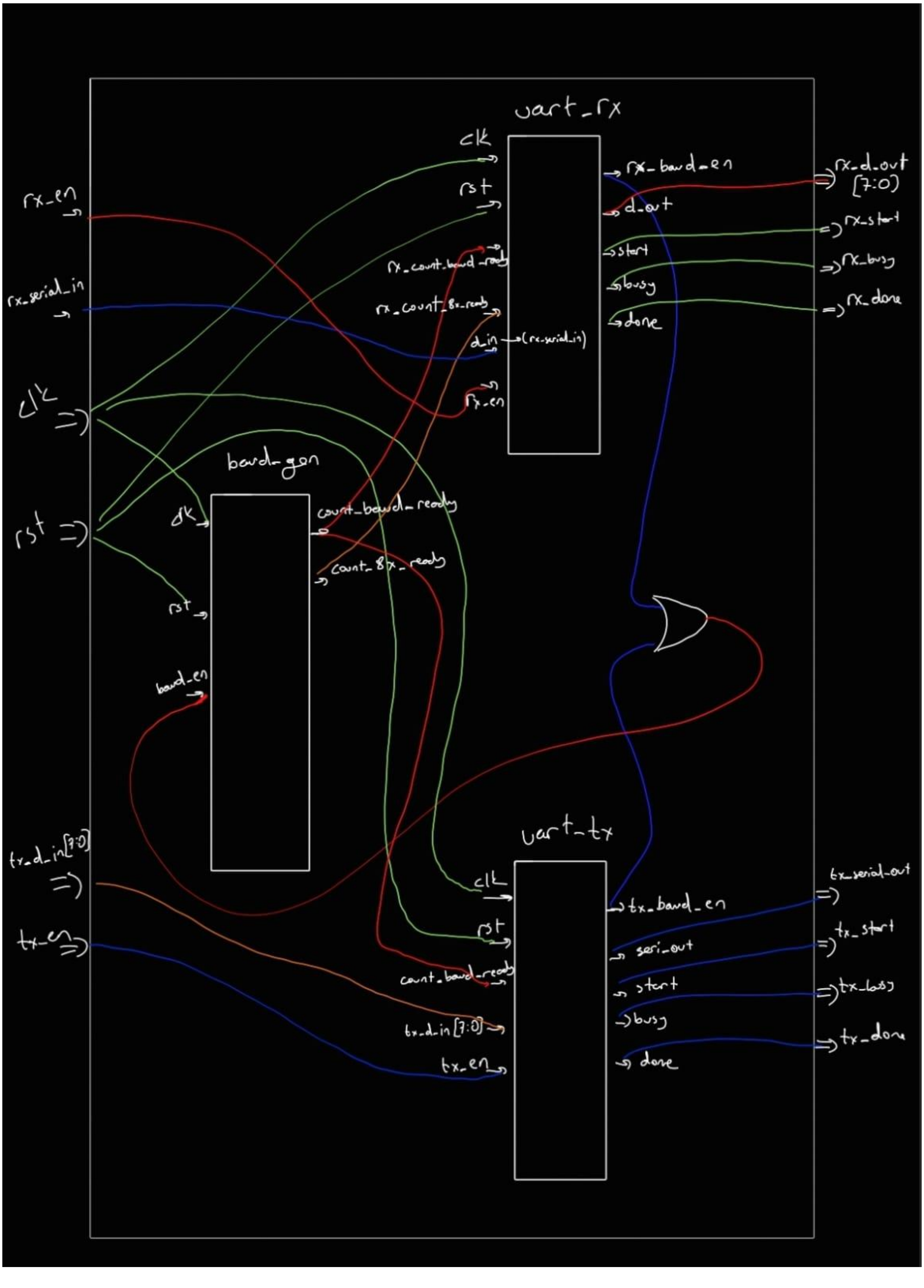
Worst Pulse Width Slack (WPWS):	4,500 ns
Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	61

All user specified timing constraints are met.

Our WHS value is 6.511ns. Therefore, 10 – 6.511= 3.489 ns. Maximum clock frequency is:

$$F_{max} \approx \frac{1}{3.489ns} \approx 286.6\ MHz$$

6.5. Block Diagram



7. References

[1] Nordic Semiconductor, “UART protocol,” *nRF Connect SDK Fundamentals - Serial Communication (UART)*, Academy.nordicsemi.com, 2025. [Online]. Available: https://academy.nordicsemi.com/courses/nrf-connect-sdk-fundamentals/lessons/lesson-4-serial-communication-uart/topic/uart-protocol/ [Accessed: Dec. 28, 2025].

[2] J. Wong, “Software UART through emulated GPIO pins,” *JimmyWongIoT.com*, Oct. 23, 2022. [Online]. Available: https://jimmywongiot.com/2022/10/23/software-uart-through-emulated-gpio-pins/ [Accessed: Dec. 28, 2025].

[3] Seeed Studio, “UART Communication Protocol and How It Works,” *Seeed Studio Blog*, Sep. 8, 2022. [Online]. Available: https://www.seeedstudio.com/blog/2022/09/08/uart-communication-protocol-and-how-it-works/ [Accessed: Dec. 28, 2025].

[4] Article World, “Fundamentals of UART Communication,” *ArticleWorld.com*, [Online]. Available: https://aticleworld.com/fundamentals-of-uart-communication/ [Accessed: Dec. 28, 2025].

[5] Analog Devices, “UART — A Hardware Communication Protocol,” *Analog Dialogue*, [Online]. Available: https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html [Accessed: Dec. 28, 2025].