UART Implementation on FPGA with System Verilog

Nilakna D. Warushavithana

May 23, 2025

Contents

1 UART: Introduction **4**

2 Transmitter **5**

2.1 Introduction 5

2.2 RTL Code in System Verilog 6

2.3 State Machine: Transmitter 7

2.4 RTL design view 7

2.5 Verification Test Bench in System Verilog 8

2.6 Test bench simulation for Verification 9

3 Receiver **10**

3.1 Introduction 10

3.2 RTL Code in System Verilog 11

3.3 State Machine: Receiver 12

3.4 RTL design view 12

3.5 Verification Test Bench in System Verilog 13

3.6 Test bench simulation 14

4 Bidirectional UART Transceiver **15**

4.1 Introduction 15

4.2 RTL Code in System Verilog 16

4.3 RTL design view 16

4.4 Verification Test Bench in System Verilog 17

4.5 Test bench simulation 18

5 Implementation on DEO Nano FPGA **20**

5.1 Introduction 20

5.2 Pin plan 21

5.3 RTL Code of the top module 22

5.4 RTL design view 22

5.5 Results on FPGA 23

List of Figures

1 RTL design of transmitter module 6
2 Transmitter state machine 7
3 Transmitter module RTL 7
4 Test bench code of transmitter module 8
5 Transmitter test bench simulation waveforms for verification 9
6 RTL design of receiver module 11
7 Receiver state machine 12
8 Receiver module RTL 12
9 Test bench code of receiver module 13
10 Receiver test bench simulation waveforms for verification 14
11 RTL design of UART transceiver module 16
12 UART transceiver module block diagram 16
13 Test bench code of UART transceiver module 17
14 UART test bench simulation timing diagram waveforms 18
15 Vivado Simulation 19
16 Pin planner for DEO Nano Board 21
17 Top module RTL code for FPGA 22
18 UART top module block diagram 22
19 Sample data sending from UART TX to RX 23
20 RX is '1 when transmitter sends no data 24

1 UART: Introduction

The Universal Asynchronous Receiver-Transmitter (UART) represents one of the most fundamental and widely adopted serial communication protocols. This implementation demonstrates a complete UART communication system designed for FPGA deployment. The UART protocol is simple, reliable, and has minimal hardware requirements, making it suitable for many applications.

The UART protocol operates as an asynchronous serial communication standard; so, it does not need a shared clock signal between communicating devices. Data transmission occurs through a predetermined frame structure consisting of a start bit, data payload, optional parity bit, and stop bit(s). Since the clock is not shared, a timing agreement between the transmitter and receiver should be there. It is typically achieved through standardized baud rates such as 9600, 38400, 115200, or higher frequencies, depending on the requirements.

Protocol specifications for this implementation follow the standard 8N1 configuration: eight data bits, no parity bit, and one stop bit. Each transmitted frame begins with a start bit (logic low) that signals the beginning of data transmission, followed by eight data bits starting from the LSB, and concludes with a stop bit (logic high) that returns the communication line to its idle state. The idle condition maintains the transmission line at logic high, providing a clear reference for start bit detection.

This FPGA implementation utilizes a hierarchical design methodology, separating transmission and reception functionality into separate modules while maintaining protocol through parameter management and signal coordination. The design uses 50 MHz system clock on the DEO-Nano board with 115200 baud operation (434 clock cycles per bit period). The implementation incorporates digital design practices, including parameterized modules, state machine design, and robust reset handling to ensure reliable operation in practical deployment scenarios.

2 Transmitter

2.1 Introduction

The transmitter module implements the UART transmission using a finite state machine (FSM) with two primary states: IDLE and SEND. The module accepts parallel data input and serializes it for transmission according to the UART protocol standard. Upon receiving a valid send signal (`send_valid`), the transmitter creates the complete UART frame by adding a start bit (logic 0), the payload (8-bit data), and a stop bit (logic 1) into a 10-bit packet stored in `send_packet`.

The transmission happens through precise timing control using two counters: `clk_cnt` for pulse width timing and `data_cnt` for counting data bits to track bit position. The pulse width counter ensures each transmitted bit maintains the correct duration based on the configured baud rate (`PULSE_WIDTH` parameter). During transmission, the module right shifts the data packet, outputting each bit sequentially through the tx output line. The transmitter keeps the tx line at logic high during idle periods, maintaining standard UART specifications.

Finite state machine and the state transmissions ensure reliable communication. The module maintains `tx_ready` only during the IDLE state, preventing data corruption from early transmission requests. Clock considerations are managed by synchronous reset implementation, ensuring predictable behavior during system initialization. Our parameterized design allows for flexible configuration of word size, pulse width, and packet size, making it adaptable to various communication requirements while maintaining protocol requirements.

2.2 RTL Code in System Verilog

```
`timescale 1ns / 1ps

module transmitter #(
    parameter
    WORD_SIZE = 8,      // 8 bits per word
    PULSE_WIDTH = 4,    // PULSE_WIDTH = CLOCK_FREQ/BAUD, CLOCK_FREQ = 100000000, BAUD = 115200, CLK_PERIOD = 10
    PACKET_SIZE = 10    // start, word, stop
    )
    (
    input logic clk, rstn, send_valid,
    input logic [WORD_SIZE-1:0]data_bits,                       // input data bit for a word
    output logic tx_ready, tx
    );

    typedef enum logic [1:0] {IDLE, SEND} statetype;            // data type for states
    statetype state;                                            // state variable

    localparam PACKET_WIDTH_BITS = $clog2(PACKET_SIZE);         // for data count
    localparam PULSE_WIDTH_BITS = $clog2(PULSE_WIDTH);          // for clock count

    // initialize data packet to transmit
    logic [PACKET_SIZE-1:0] data_packet;
    logic [PACKET_SIZE-1:0] send_packet;

    // assign start bit and stop bit to data packet
    assign send_packet = {1'b1, data_bits, 1'b0};               //stop bit = 1, data, start bit = 0

    // initiate a tx register to update the bit to send
    logic tx_reg;

    // assign output bit
    assign tx = tx_reg;

    // counters for packet length in pulses and pulse length in clocks
    logic [PACKET_WIDTH_BITS-1:0] data_cnt;
    logic [PULSE_WIDTH_BITS-1:0] clk_cnt;

   // transmitter is ready to transmit in IDLE state
   assign tx_ready = (state==IDLE);

   // state machine
   always_ff @(posedge clk or negedge rstn) begin
       if (!rstn) begin
           state <= IDLE;
           data_packet <= '1;
           data_cnt <= 0;
           clk_cnt <= 0;
           tx_reg <= 1'b1;                                      // UART line is high when idle
           end
       else case (state)
           IDLE: if (send_valid) begin
               state <= SEND;                                   // if send is valid move to send state
               data_packet <= send_packet;                      // copy the packet to data
               data_cnt <= 0;                                   // reset counter
               tx_reg <= send_packet[0];                        // set the first bit immediately
               end

           SEND: begin
               if (clk_cnt == PULSE_WIDTH-1) begin
                   clk_cnt <= 0;                                // reset the clock count

                   if (data_cnt == PACKET_SIZE-1) begin         // if all data in packet is sent, finish and IDLE
                       data_packet <= '1;
                       data_cnt <= 0;
                       state <= IDLE;
                       tx_reg <= 1'b1;                          // Set line high when returning to idle
                       end
                   else begin                                   // if not finished with data yet, send them
                       data_cnt <= data_cnt + 1;                // count the bits
                       data_packet <= (data_packet >> 1);       // shift the data register
                       tx_reg <= data_packet[1];                // update the sending bit to the next bit
                       end
                   end
               else clk_cnt <= clk_cnt + 1;                     // if pulse is not complete count the clocks
               end
           endcase
   end

endmodule
```

Figure 1: RTL design of transmitter module

2.3 State Machine: Transmitter

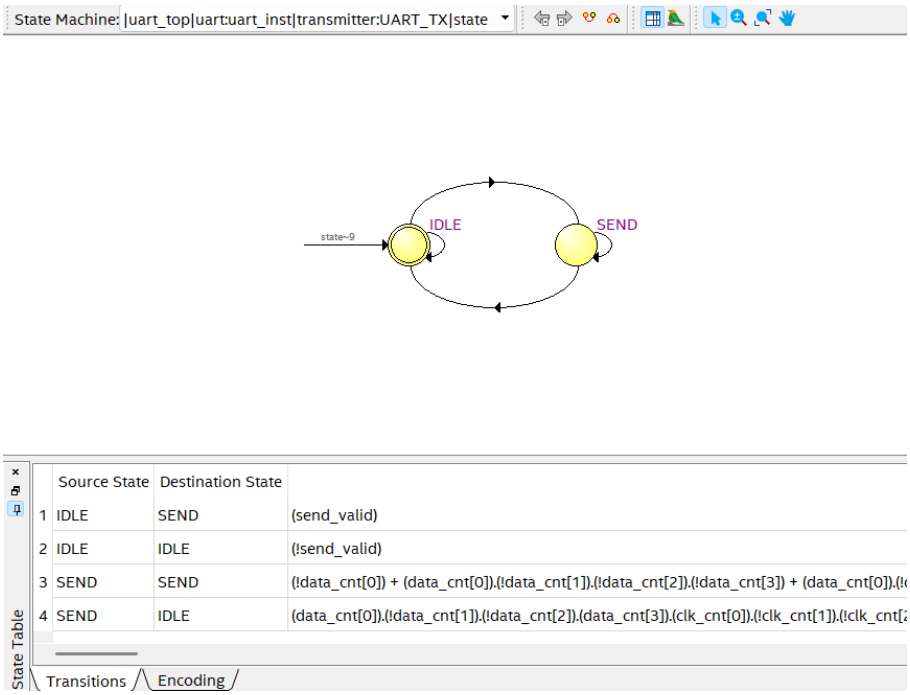


Figure 2: Transmitter state machine

2.4 RTL design view

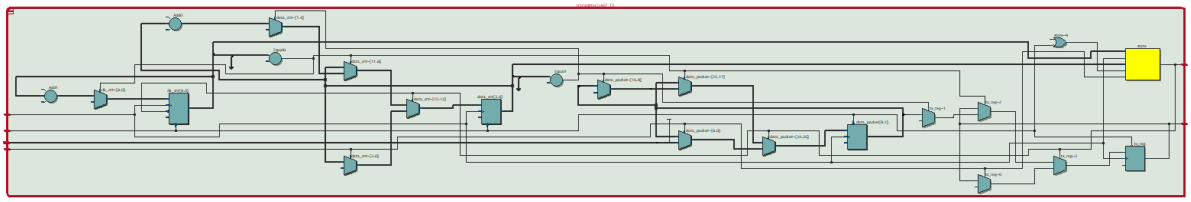


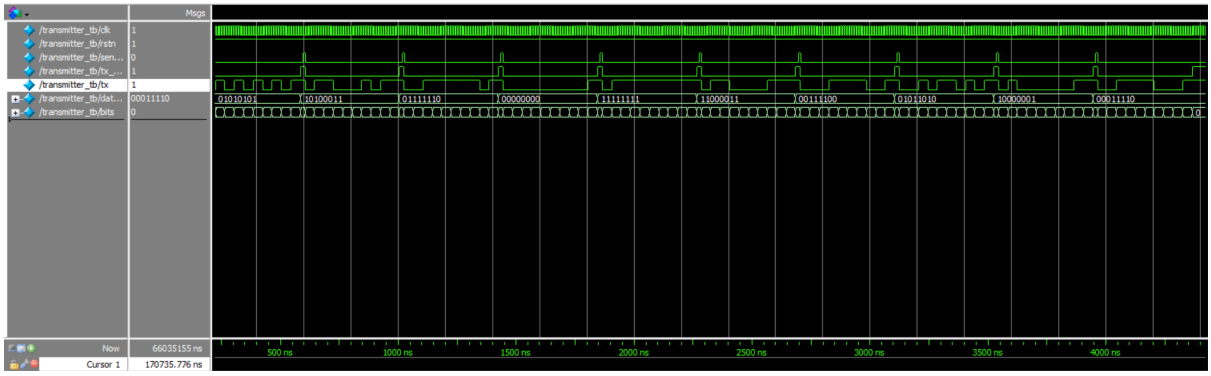
Figure 3: Transmitter module RTL

2.5 Verification Test Bench in System Verilog

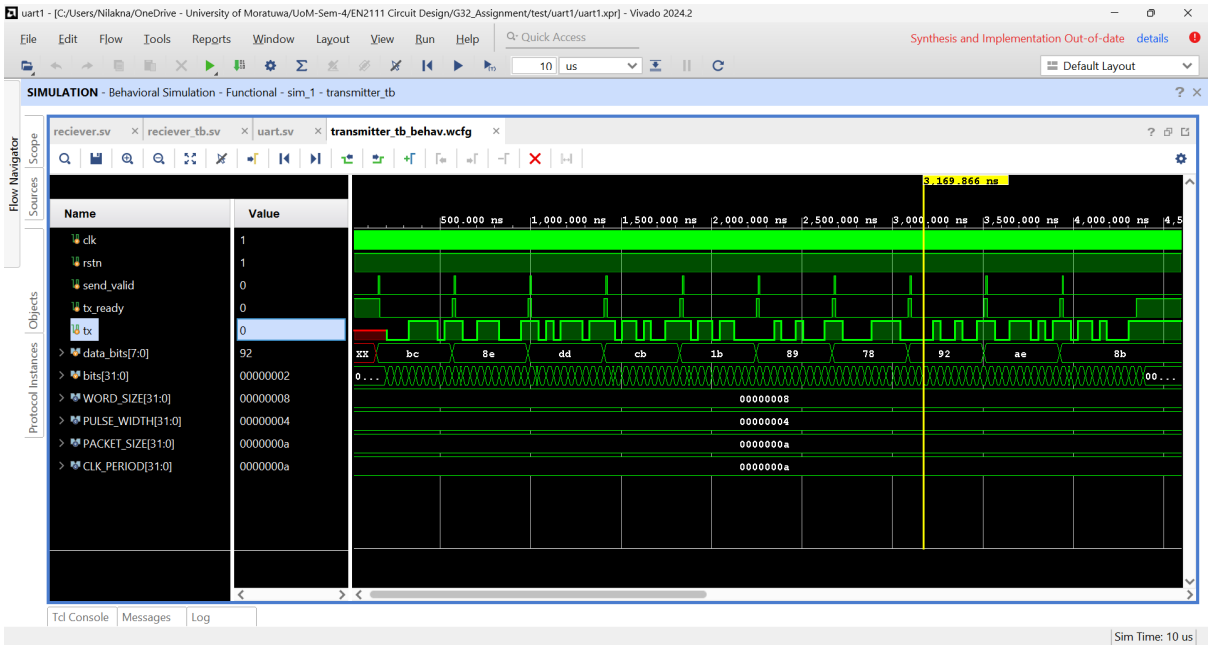
```
`timescale 1ns / 1ps
module transmitter_tb;
//    timeunit 1ns; timeprecision 1ps;

    // parameters
    localparam  WORD_SIZE = 8,  // 8 bits per word
                PULSE_WIDTH = 4, // PULSE_WIDTH = CLOCK_FREQ/BAUD, CLOCK_FREQ = 100000000, BAUD = 115200, CLK_PERIOD = 10
                PACKET_SIZE = 10, // start, word, stop
                CLK_PERIOD = 10; // 100Mhz

    // signals in test bench
    logic   clk = 0,
            rstn = 0, // start in reset mode
            send_valid = 0,
            tx_ready,
            tx;
    logic [WORD_SIZE-1:0] data_bits;               // data to/before transmit
    logic [WORD_SIZE-1:0] test_values [0:9];       // words for 10 test cases

    // instantiate the transmitter
    transmitter #(  .WORD_SIZE(WORD_SIZE),
                    .PULSE_WIDTH(PULSE_WIDTH),
                    .PACKET_SIZE(PACKET_SIZE))
                dut (
                    .clk(clk),
                    .rstn(rstn),
                    .send_valid(send_valid),
                    .data_bits(data_bits),
                    .tx_ready(tx_ready),
                    .tx(tx)
                );

    // clock generation
    always #(CLK_PERIOD/2) clk <= !clk;      // 5 = CLK_PERIOD/2

    // generate test cases
    initial begin
        test_values[0] = 8'h55;
        test_values[1] = 8'hA3;
        test_values[2] = 8'h7E;
        test_values[3] = 8'h00;
        test_values[4] = 8'hFF;
        test_values[5] = 8'hC3;
        test_values[6] = 8'h3C;
        test_values[7] = 8'h5A;
        test_values[8] = 8'h81;
        test_values[9] = 8'h1E;
    end
    // test sequence
    initial begin
        $dumpfile ("dump.vcd"); $dumpvars;

        // reset
        #20 rstn = 1;       // remove reset  after 20 ns (2 clks)
        repeat(5) @(posedge clk) #1;


        // send test cases (10 words)
        for (int i = 0; i < 10; i = i+1) begin
        // // generate random 10 test cases (10 words)
        // repeat (10) begin

            //repeat ($urandom_range(1,20)) @(posedge clk);
            repeat(10) @(posedge clk);

            wait (tx_ready == 1);   // wait till tx is ready. ready in IDLE

            // data_bits = $urandom(); // test pattern [7:0] recieve is 0,01001001,1
            data_bits = test_values[i];

            @(posedge clk) #1 send_valid = 1;     // start sending
            @(posedge clk) #1 send_valid = 0;     // send signal pulse ends

            #10 $display("sending 0,%b,1", data_bits);
            end
        end

    // count uart bits - for waveform
    int bits;
    initial forever begin
        bits = 0;
        wait (!tx);
        for (int j = 0; j < PACKET_SIZE; j = j+1) begin
            bits += 1;
            repeat (PULSE_WIDTH) @(posedge clk);
            end
        end

endmodule
```

Figure 4: Test bench code of transmitter module

2.6 Test bench simulation for Verification



(a) Simulation in Quartus Prime - Modelsim Altera



(b) Simulation in Vivado

Figure 5: Transmitter test bench simulation waveforms for verification

3 Receiver

3.1 Introduction

The receiver module implements an FSM with 4 states: IDLE, START, DATA, and STOP, to decode the incoming UART serial data stream. The design implements timing recovery and bit sampling to ensure reliable data reception even with clock differences between transmitter and receiver systems. The module continuously monitors the rx input line for the start bit transition from idle high to active low.

Upon detecting a start bit, the receiver transitions to the START state and initiates sampling at the middle of the bit by counting to half the pulse width period. Then it transitions to the DATA state, where it sequentially captures 8 data bits and storing them in the `data_bits` register with indexing to maintain bit order. Each bit is sampled after a full pulse width duration to ensure synchronization with the transmitter's timing.

The receiver's timing implementation has 2 counters: `clk_cnt` for bit duration timing and `data_cnt` for tracking received data bits within each word. Error handling is done through state timeout and reset procedures. The module validates frame structure with the stop bit in the STOP state before returning to IDLE. The `rx_valid` signal indicates successful frame reception and data availability, enabling us to process received data. Parameterization supports various word sizes and baud rates while maintaining protocol integrity.

3.2 RTL Code in System Verilog

```
`timescale 1ns / 1ps
module reciever #(
    parameter   WORD_SIZE = 8,   // 8 bits per word
                PULSE_WIDTH = 4, // PULSE_WIDTH = CLOCK_FREQ/BAUD, CLOCK_FREQ = 100000000, BAUD = 115200, CLK_PERIOD = 10
                PACKET_SIZE = 10 // start, word, stop
    )(
    input logic clk, rstn, rx,
    output logic rx_valid,
    output logic [WORD_SIZE-1:0]data_bits
    );

    typedef enum logic [1:0] {IDLE, START, DATA, STOP} statetype; // define state data type
    statetype state;                                              // create state variable

    localparam PULSE_WIDTH_BITS = $clog2(PULSE_WIDTH);            // for clock count
    localparam WORD_SIZE_BITS = $clog2(WORD_SIZE);                // for bit count in word

    // counters for pulse and word
    logic [PULSE_WIDTH_BITS-1:0] clk_cnt;
    logic [WORD_SIZE_BITS-1:0] data_cnt;

    // reciever is ready to recieve in IDLE state
    assign rx_valid = (state==IDLE);

    // state machine
    always_ff @(posedge clk or negedge rstn) begin

        if (!rstn) begin
            state <= IDLE;
            data_bits <= '0;
            data_cnt <= 0;
            clk_cnt <= 0;
            end

        else case (state)
            IDLE: begin
                if (rx==0) state <= START;                        // when we get start bit, go to start state
                end

            START: begin
                if (clk_cnt == PULSE_WIDTH/2-1) begin

                    clk_cnt <= 0;                                 // reset clock for data
                    data_cnt <= 0;                                // reset data count
                    data_bits <= '0;                              // reset data_bits register
                    state <= DATA;                                // halfway through start bit, switch to data state

                    end
                else clk_cnt += 1;                                // if not increase clock count
                end

            DATA: begin
                if (clk_cnt == PULSE_WIDTH-1) begin
                    clk_cnt <= 0;                                 // reset the pulse count clock
                    data_bits[data_cnt] <= rx;                    // append the recieved bit rx

                    if (data_cnt == WORD_SIZE-1) begin
                        data_cnt <= 0;                            // reset data bit count for word
                        state <= STOP;                            // go to STOP state
                        end
                    else data_cnt += 1;                           // if word isnt finised, sount data bits

                    end
                else clk_cnt += 1;                                // if not increase pulse clock count
                end

            STOP: begin
                if (clk_cnt == PULSE_WIDTH-1) begin
                    state <= IDLE;                                // go to IDLE state after this
                    clk_cnt <= 0;                                 // reset pulse clock
                    end
                else clk_cnt += 1;                                // if not increase pulse clock count

                end
            endcase
        end

endmodule
```

Figure 6: RTL design of receiver module

3.3 State Machine: Receiver

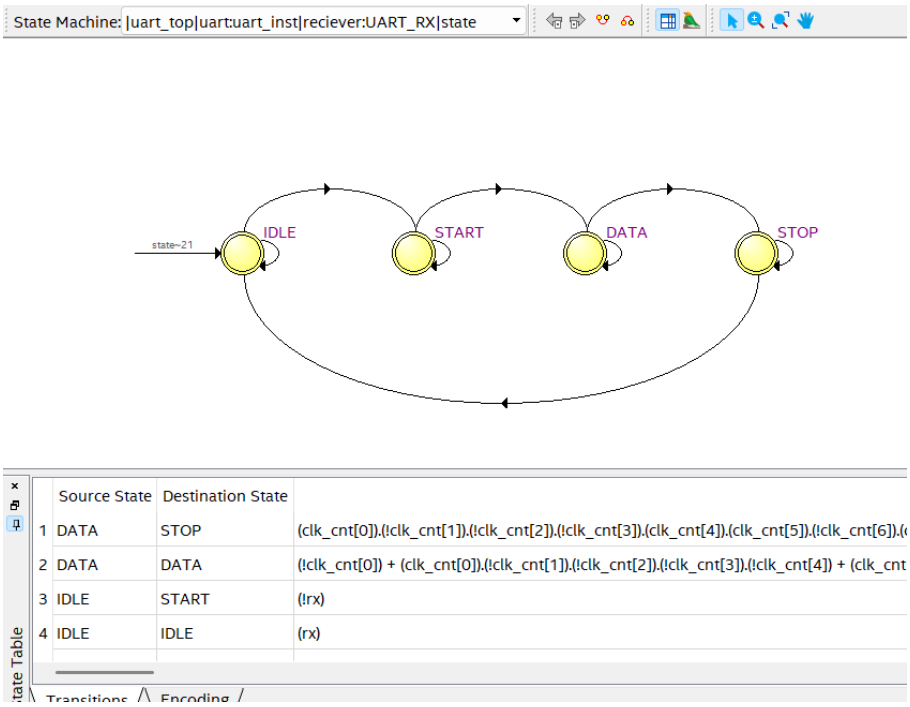


Figure 7: Receiver state machine

3.4 RTL design view

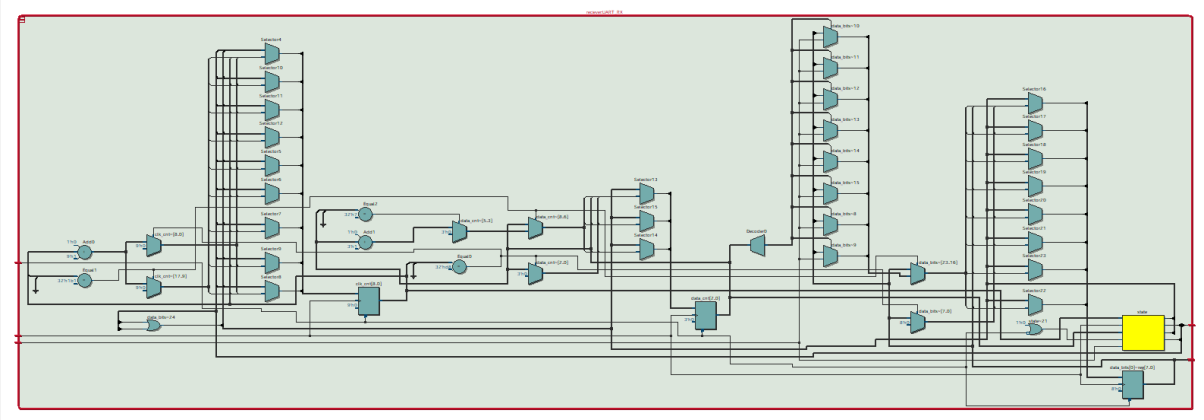


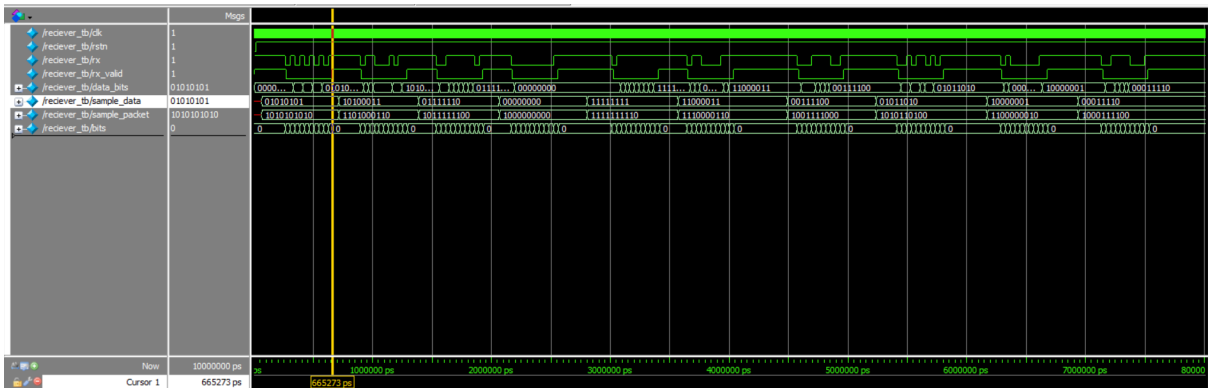
Figure 8: Receiver module RTL

3.5 Verification Test Bench in System Verilog

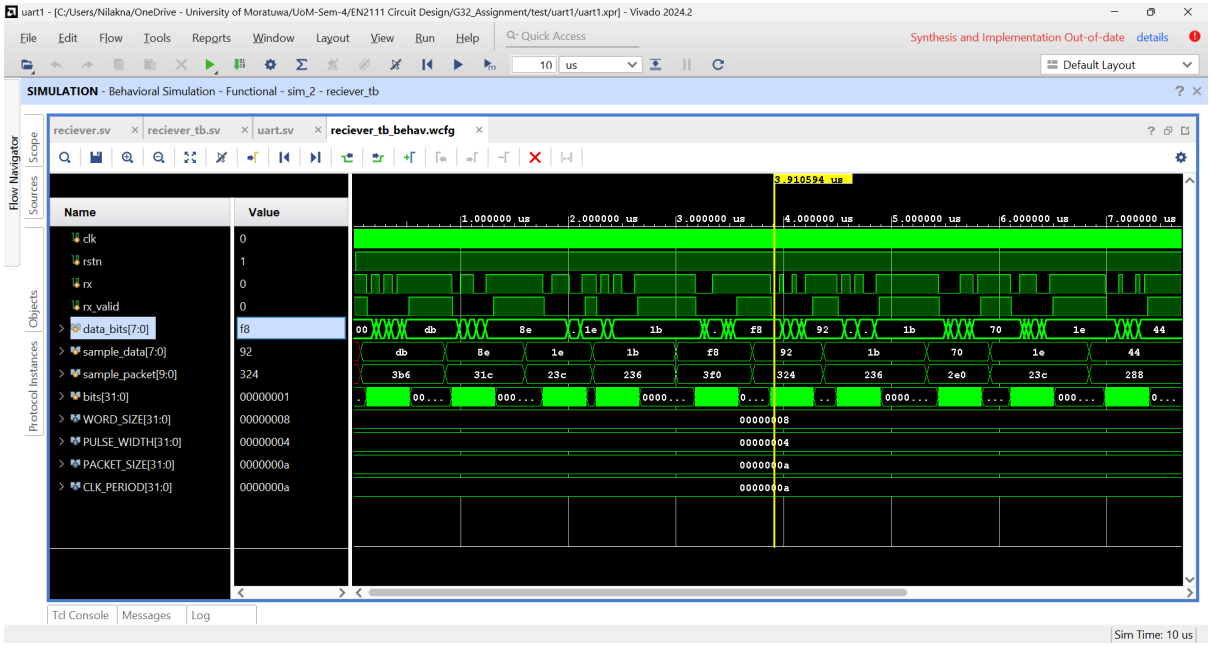
```
`timescale 1ns / 1ps
module reciever_tb;
    //parameters
    localparam  WORD_SIZE = 8,  // 8 bits per word
                PULSE_WIDTH = 4, // PULSE_WIDTH = CLOCK_FREQ/BAUD, CLOCK_FREQ = 100000000, BAUD = 115200, CLK_PERIOD = 10
                PACKET_SIZE = 10, // start, word, stop
                CLK_PERIOD = 10; // 100Mhz

    // signals
    logic   clk = 0,
            rstn = 0, // start in reset
            rx = 1,
            rx_valid;
    logic [WORD_SIZE-1:0] data_bits;

    // test bench data and packet simul
    logic [WORD_SIZE-1:0] sample_data;
    logic [WORD_SIZE-1+2:0] sample_packet;
    logic [WORD_SIZE-1:0] test_values [0:9];       // words for 10 test cases


    // instantiate the reciever
    reciever #(   .WORD_SIZE(WORD_SIZE),
                  .PULSE_WIDTH(PULSE_WIDTH),
                  .PACKET_SIZE(PACKET_SIZE))
                dut (
                  .clk(clk),
                  .rstn(rstn),
                  .data_bits(data_bits),
                  .rx(rx),
                  .rx_valid(rx_valid)
                );

    // clock generation
    always #(CLK_PERIOD/2) clk <= !clk;      // 5 = CLK_PERIOD/2
    // generate test cases
    initial begin
        test_values[0] = 8'h55;
        test_values[1] = 8'hA3;
        test_values[2] = 8'h7E;
        test_values[3] = 8'h00;
        test_values[4] = 8'hFF;
        test_values[5] = 8'hC3;
        test_values[6] = 8'h3C;
        test_values[7] = 8'h5A;
        test_values[8] = 8'h81;
        test_values[9] = 8'h1E;
    end

    // test sequence
    initial begin
        $dumpfile ("dump.vcd"); $dumpvars;

        // reset
        #20 rstn = 1;       // remove reset  after 20 ns (2 clks)
        repeat(5) @(posedge clk) #1;


        // send test cases (10 words)
        for (int i = 0; i < 10; i = i+1) begin
        // generate random 10 test cases (10 words)
        // repeat (10) begin
            // generate sample data stream
            // sample_data = $urandom(); // test pattern [7:0] recieve is 0,01001001,1
            sample_data = test_values[i];
            sample_packet = {1'b1, sample_data, 1'b0}; // packet format


            //  send these to the reciever
            repeat ($urandom_range(1,20)) @(posedge clk);
            for (int i = 0; i < WORD_SIZE+2; i = i+1)
                repeat (PULSE_WIDTH) @(posedge clk) #1 rx <= sample_packet[i];      // latch bit from the packet

            // for debugging
            #10 $display("should recieve %b", sample_data);

            // add some delay
            repeat ($urandom_range(1,50)) @(posedge clk);            end
        end
    // count uart bits - for waveform
    int bits;
    initial forever begin
        bits = 0;
        wait (!rx);
        for (int j = 0; j < PACKET_SIZE; j = j+1) begin
            bits += 1;
            repeat (PULSE_WIDTH) @(posedge clk);
            end
        end

endmodule
```

Figure 9: Test bench code of receiver module

3.6 Test bench simulation



(a) Simulation in Quartus Prime - Modelsim Altera



(b) Simulation in Vivado

Figure 10: Receiver test bench simulation waveforms for verification

4 Bidirectional UART Transceiver

4.1 Introduction

The UART transceiver module serves as the top-level module that integrates the transmitter and receiver into a complete UART communication interface. This hierarchical design approach allows modularity, code reusability, and simplified testing. The module instantiates both transmitter and receiver sub-modules with matching parameter passing.

Signal routing within the transceiver creates independent transmit and receive data paths while sharing clock and reset common control signals. The design maintains separate data buses (`data_bits_tx` and `data_bits_rx`) to implement bidirectional communication without interference. Control signals handle `send_valid, tx_ready, and rx_valid` to coordinate data flow and indicate system status.

The transceiver supports full-duplex operation, allowing simultaneous transmission and reception operations for bidirectional communication protocols. This design approach enables easy modification of individual transmission or reception characteristics due to its modularity and parametric implementation.

4.2 RTL Code in System Verilog

```
`timescale 1ns / 1ps
module uart #(
    parameter
        WORD_SIZE = 8,  // 8 bits per word
        PULSE_WIDTH = 4, // PULSE_WIDTH = CLOCK_FREQ/BAUD, CLOCK_FREQ = 100000000, BAUD = 115200, CLK_PERIOD = 10
        PACKET_SIZE = 10 // start, word, stop
    )(
    input logic clk, rstn,
                send_valid,
                rx,
    input logic [WORD_SIZE-1:0] data_bits_tx,
    output logic [WORD_SIZE-1:0] data_bits_rx,
    output logic rx_valid,
                tx_ready, tx
    );

    // instantiate the transmitter
    transmitter #(
            .WORD_SIZE(WORD_SIZE),
            .PULSE_WIDTH(PULSE_WIDTH),
            .PACKET_SIZE(PACKET_SIZE)
    ) UART_TX (
            .clk(clk),
            .rstn(rstn),
            .send_valid(send_valid),
            .data_bits(data_bits_tx),
            .tx_ready(tx_ready),
            .tx(tx)
            );

    // instantiate the reciever
    reciever #(
            .WORD_SIZE(WORD_SIZE),
            .PULSE_WIDTH(PULSE_WIDTH),
            .PACKET_SIZE(PACKET_SIZE)
    ) UART_RX (
            .clk(clk),
            .rstn(rstn),
            .data_bits(data_bits_rx),
            .rx(rx),
            .rx_valid(rx_valid)
            );

endmodule
```

Figure 11: RTL design of UART transceiver module

4.3 RTL design view

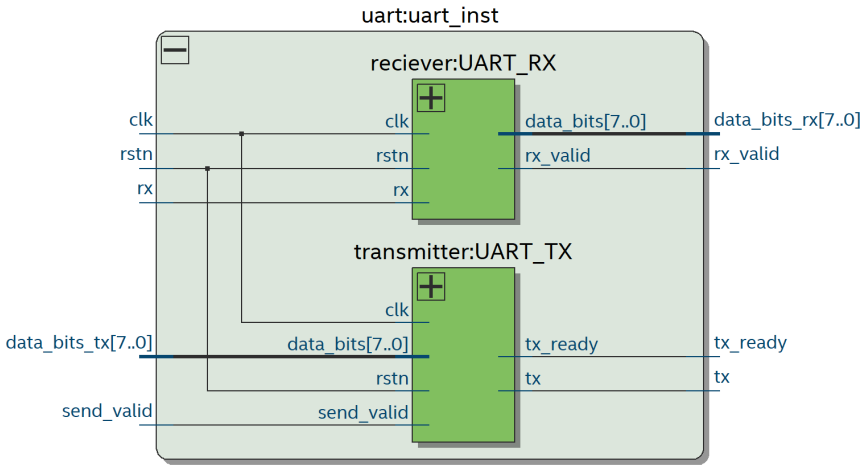


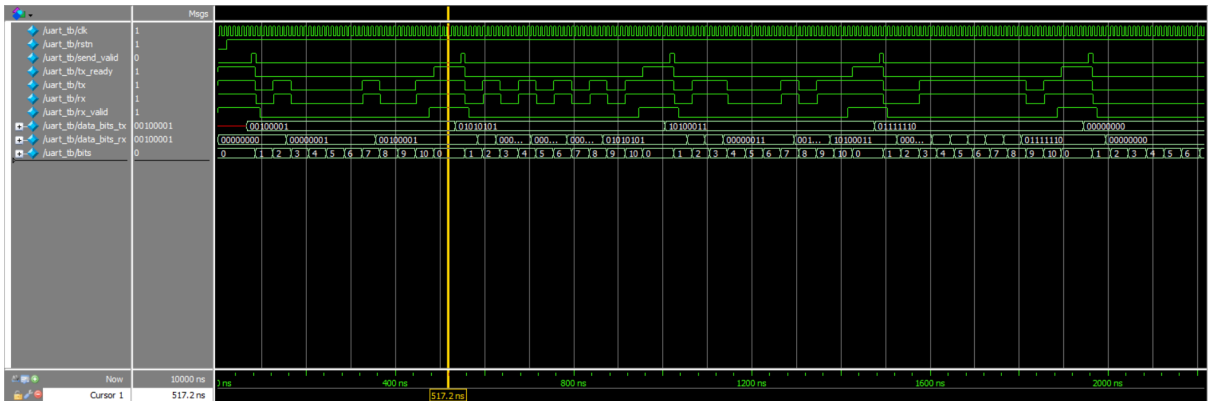
Figure 12: UART transceiver module block diagram

4.4 Verification Test Bench in System Verilog

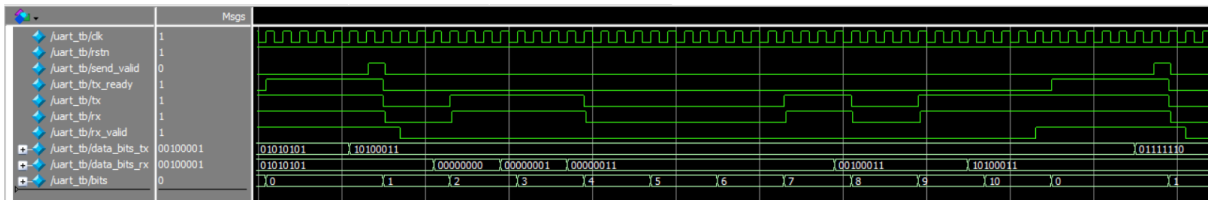
```
`timescale 1ns / 1ps

module uart_tb;
    // parameters
    localparam  WORD_SIZE = 8,                      // 8 bits per word
                PULSE_WIDTH = 4,                    // PULSE_WIDTH = CLOCK_FREQ/BAUD, CLOCK_FREQ = 100000000, BAUD = 115200, CLK_PERIOD = 10
                PACKET_SIZE = 10,                   // start, word, stop
                CLK_PERIOD = 10;                    // 100MHz

    // signals in test bench
    logic   clk = 0,
            rstn = 0,                               // start in reset mode
            send_valid = 0,
            tx_ready,
            tx,
            rx = 1,
            rx_valid;

    logic [WORD_SIZE-1:0] data_bits_tx;             // data to transmit
    logic [WORD_SIZE-1:0] data_bits_rx;             // data recieved
    logic [WORD_SIZE-1:0] test_values [0:9];        // words for 10 test cases

    // clock generation
    always #(CLK_PERIOD/2) clk <= !clk;             // 10 ns period

    // Instantiate UART module
    uart #(
        .WORD_SIZE(WORD_SIZE),
        .PULSE_WIDTH(PULSE_WIDTH),
        .PACKET_SIZE(PACKET_SIZE)
    ) dut (
        .clk(clk),
        .rstn(rstn),
        .send_valid(send_valid),
        .rx(rx),
        .data_bits_tx(data_bits_tx),
        .data_bits_rx(data_bits_rx),
        .rx_valid(rx_valid),
        .tx_ready(tx_ready),
        .tx(tx)
    );
    // generate test cases
    initial begin
        test_values[0] = 8'h55;
        test_values[1] = 8'hA3;
        test_values[2] = 8'h7E;
        test_values[3] = 8'h00;
        test_values[4] = 8'hFF;
        test_values[5] = 8'hC3;
        test_values[6] = 8'h3C;
        test_values[7] = 8'h5A;
        test_values[8] = 8'h81;
        test_values[9] = 8'h1E;
    end

    // connecting tx and rx.
    always @(posedge clk) begin
        #1 rx = tx;
        end


    // test sequence
    initial begin
        $dumpfile ("dump.vcd"); $dumpvars;

        // reset
        #20 rstn = 1;                               // remove reset  after 20 ns (2 clks)
        repeat(5) @(posedge clk) #1;

        // test case - sending single word
        wait (tx_ready);                            // wait till tx is ready. ready in IDLE
        data_bits_tx = 8'b00100001;
        @(posedge clk) #1 send_valid = 1;           // start sending
        @(posedge clk) #1 send_valid = 0;           // send signal pulse ends

        #10 $display("sending %b", data_bits_tx);   // display the sending message (debugging)

       @(posedge rx_valid);                         // wait until rx is valid
       @(posedge clk);                              // wait for data to be fully latched
       #1 $display("Received: %b", data_bits_rx);   // display the recieved message (debugging)


        // send test cases (10 words)
        for (int i = 0; i < 10; i = i+1) begin
        //repeat (10) begin

            repeat (5) @(posedge clk);              // add a delay
            // repeat ($urandom_range(1,20)) @(posedge clk); // random delay. worked in vivado. not supported in quartus.

            wait (tx_ready);                        // wait till tx is ready. ready in IDLE

            data_bits_tx = test_values[i];
            // data_bits_tx = $urandom(); // test pattern [7:0] recieve is 0,01001001,1. random test cases. worked in vivado. not supported in quar


            @(posedge clk) #1 send_valid = 1;       // start sending
            @(posedge clk) #1 send_valid = 0;       // send signal pulse ends

            #10 $display("sending %b", data_bits_tx); // display the sending message (debugging)

            @(posedge rx_valid);                    // wait until rx is valid
            @(posedge clk);                         // wait for data to be fully latched
            #1 $display("Received: %b", data_bits_rx);// display the recieved message (debugging)
            end

        end // test sequence

    // count uart bits - for waveform
    int bits;
    initial forever begin
        bits = 0;
        wait (!tx);
        for (int j = 0; j < PACKET_SIZE; j = j+1) begin
            bits += 1;
            repeat (PULSE_WIDTH) @(posedge clk);
            end
        end
endmodule
```

Figure 13: Test bench code of UART transceiver module

4.5 Test bench simulation



(a) Capture of all transmissions (Modelsim Altera)



(b) Zoomed in transmission of one word (Modelsim Altera)

```
# sending 00100001
# Received: 00100001
# sending 01010101
# Received: 01010101
# sending 10100011
# Received: 10100011
# sending 01111110
# Received: 01111110
# sending 00000000
# Received: 00000000
# sending 11111111
# Received: 11111111
# sending 11000011
# Received: 11000011
# sending 00111100
# Received: 00111100
# sending 01011010
# Received: 01011010
# sending 10000001
# Received: 10000001
# sending 00011110
# Received: 00011110
```

(c) UART test bench console output of transmitting and receiving data

Figure 14: UART test bench simulation timing diagram waveforms

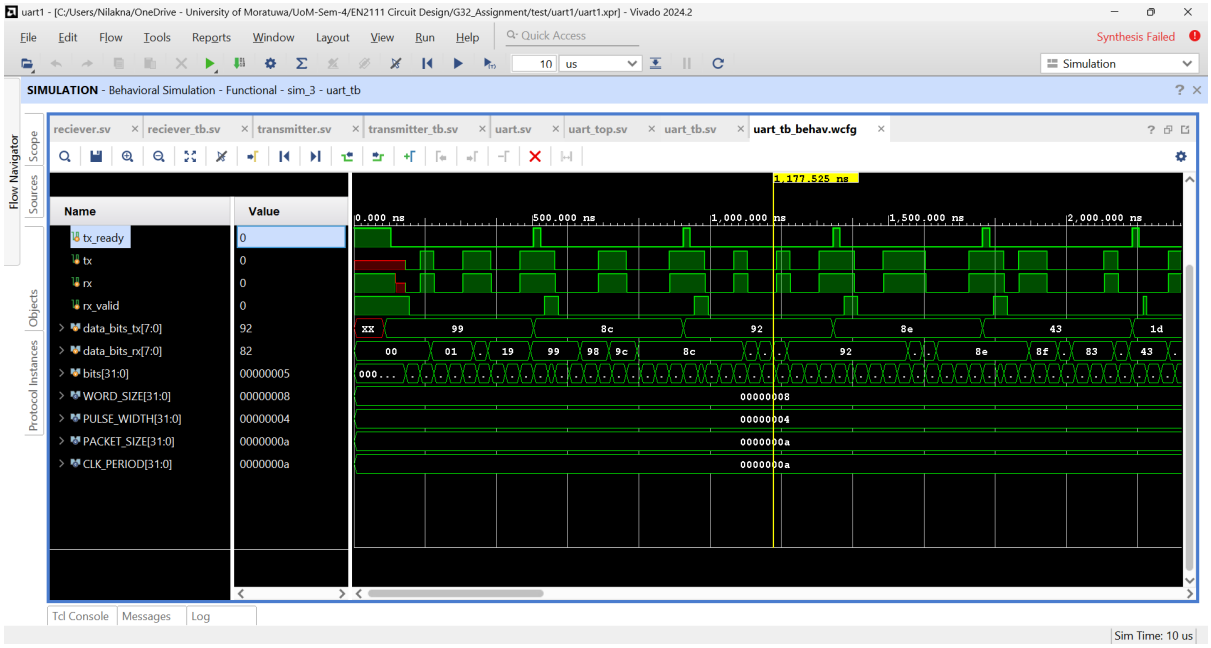


Figure 15: Vivado Simulation

5 Implementation on DEO Nano FPGA

5.1 Introduction

The UART top module represents the system-level implementation designed for FPGA deployment, incorporating practical considerations for real-world hardware interfaces. The module instantiates the complete UART transceiver with parameters configured for 115200 baud operation on a 50MHz system clock for DEO Nano, requiring a pulse width of 434 clock cycles per bit period. This configuration demonstrates proper timing calculation for reliable serial communication.

The implementation includes a continuous transmission system that automatically retransmits a fixed data pattern (`data_bits_rx`) whenever the transmitter becomes ready, creating a predictable test pattern for system verification and debugging. The `send_valid` signal ensures proper timing and prevents glitched false transmissions. Reset signal is held high for this testing implementation.

Data visualization and monitoring are implemented through LED output mapping, where received data is latched (to `data_latch`) and displayed on the FPGA's LED array. The latching mechanism ensures that the received data remains visible between transmission events, providing immediate visual feedback for verification. The clock domain considerations are addressed through synchronous design, ensuring reliable operation throughout the system. The pin assignment uses dedicated GPIO pins for UART signals. This top-level integration ensures proper I/O assignment, parameter calculation, and system-level verification approaches.

5.2 Pin plan

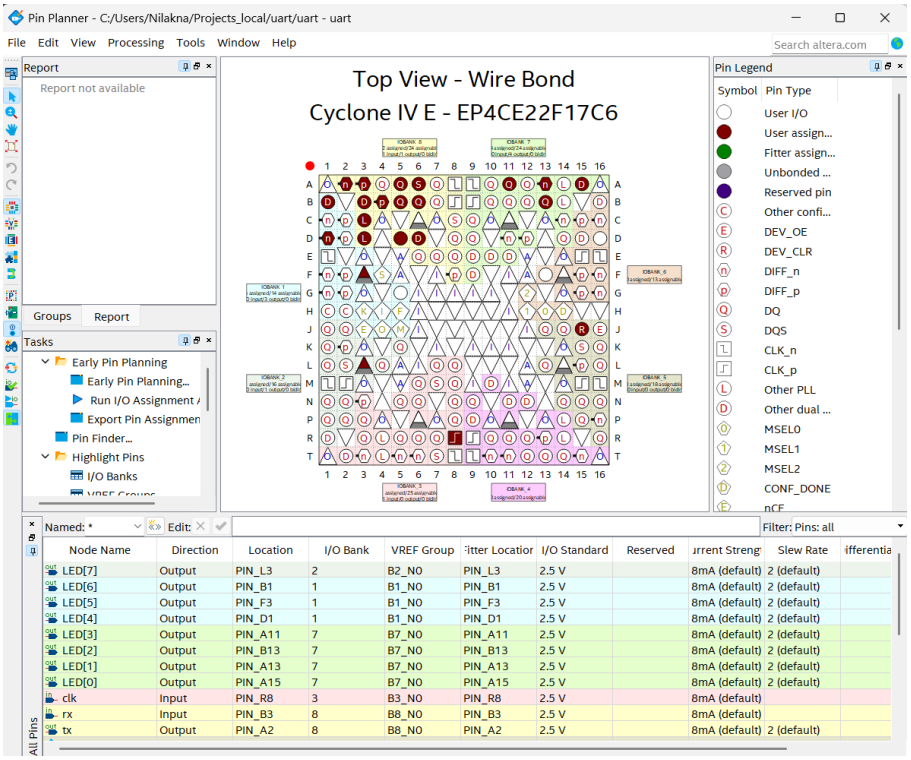


Figure 16: Pin planner for DEO Nano Board

Pins should be assigned to clk, tx, rx, and LED in the top module. Refering to the user manual [1] clock is assigned to 50 MHz oscillator via PIN_R8. The input rx is assigned to PIN_B3, pin 7 of GPIO-0 header named GPIO_02, while the output tx is assigned to PIN_A2, pin 5 of GPIO-0 header named GPIO_04 (refer to 3.5 expansion headers [1]). The LED signals are connected to the onboard LEDs as mentioned in Table 3-2 of the user manual [1].

5.3 RTL Code of the top module

```
// testing code for the uart module

module uart_top (
    input logic clk,              // 50MHz clock from DE0-Nano
    output logic tx,              // pin 5 of JP1
    input logic rx,               // pin 7 of JP1
    output logic [7:0] LED        // LED7 to LED0
);

    logic [7:0] data_bits_tx = 8'b00000110;  // tx data
    logic [7:0] data_bits_rx;
    logic rx_valid;
    logic tx_ready;
    logic send_valid;
    logic rstn;

    assign rstn = 1'b1;

    uart #(
        .WORD_SIZE(8),
        .PULSE_WIDTH(434),
        .PACKET_SIZE(10)
    ) uart_inst (
        .clk(clk),
        .rstn(rstn),
        .send_valid(send_valid),
        .data_bits_tx(data_bits_tx),
        .data_bits_rx(data_bits_rx),
        .rx_valid(rx_valid),
        .tx_ready(tx_ready),
        .tx(tx),
        .rx(rx)
    );

    logic [7:0] data_latch;

    // send_valid signal
    always_ff @(posedge clk or negedge rstn) begin
    if (!rstn)
        send_valid <= 0;
    else if (tx_ready)
        send_valid <= 1;
    else
        send_valid <= 0;
    end

    // Latch data when rx_valid goes high
    always_ff @(posedge clk) begin
        if (!rstn)
            data_latch <= 8'b0;
        else if (rx_valid)
            data_latch <= data_bits_rx;
    end


    assign LED = data_latch;

endmodule
```

Figure 17: Top module RTL code for FPGA

5.4 RTL design view

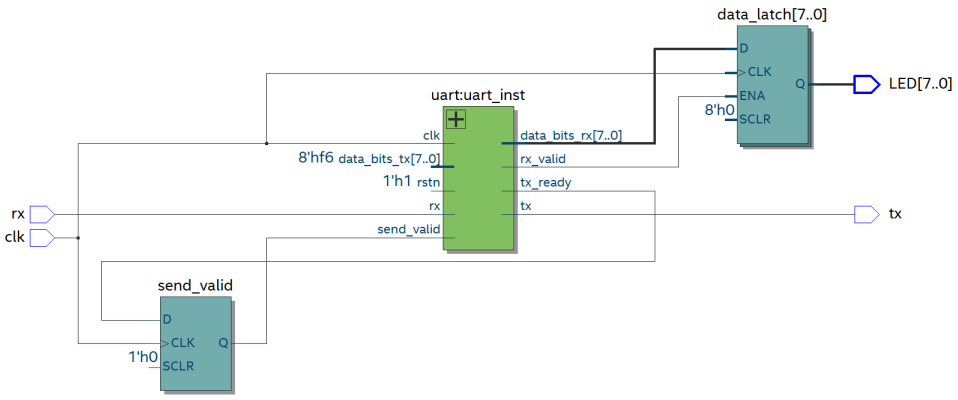
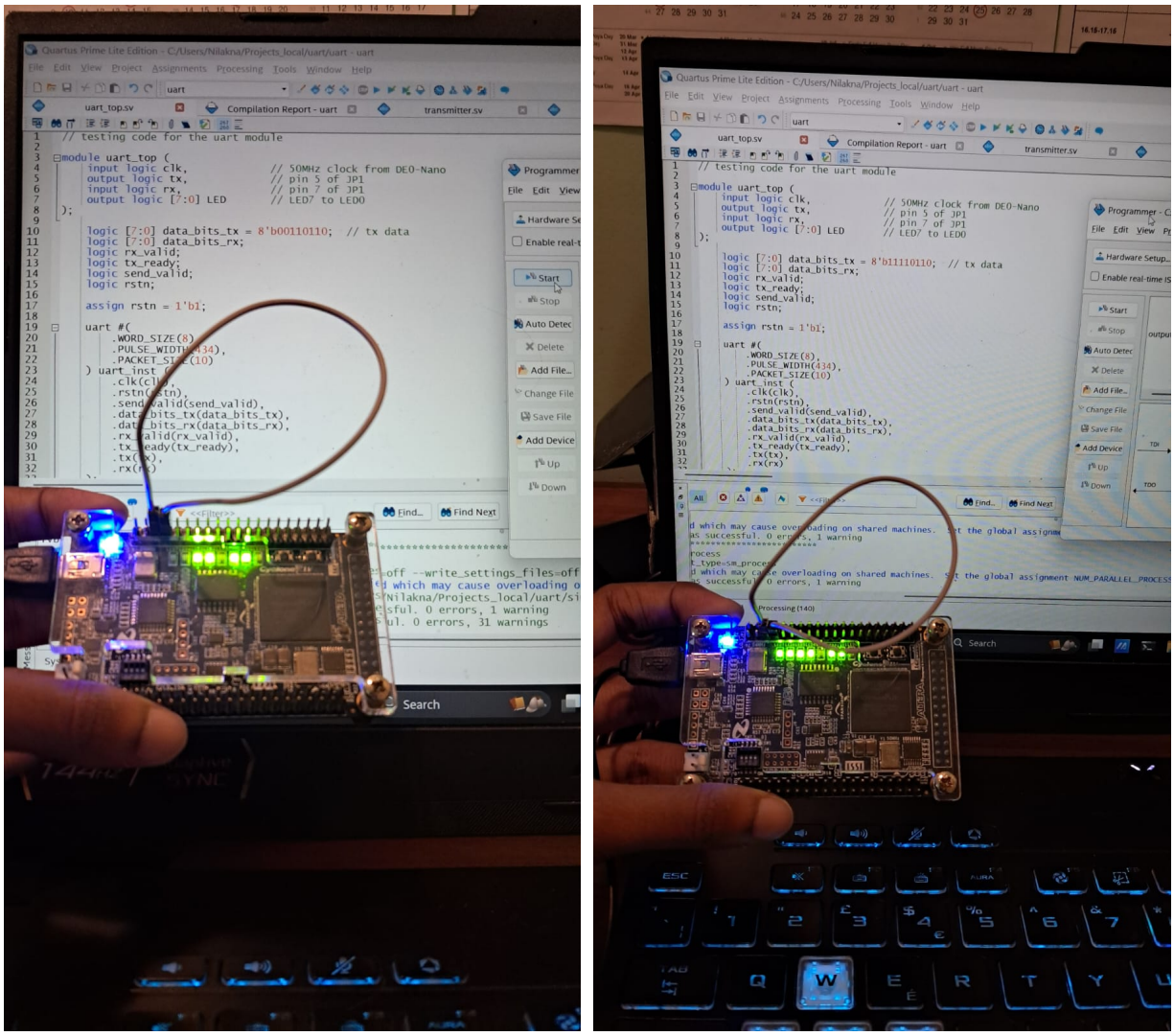


Figure 18: UART top module block diagram

5.5 Results on FPGA



(a) Payload 8'b00110110 (b) Payload 8'b11110110

Figure 19: Sample data sending from UART TX to RX

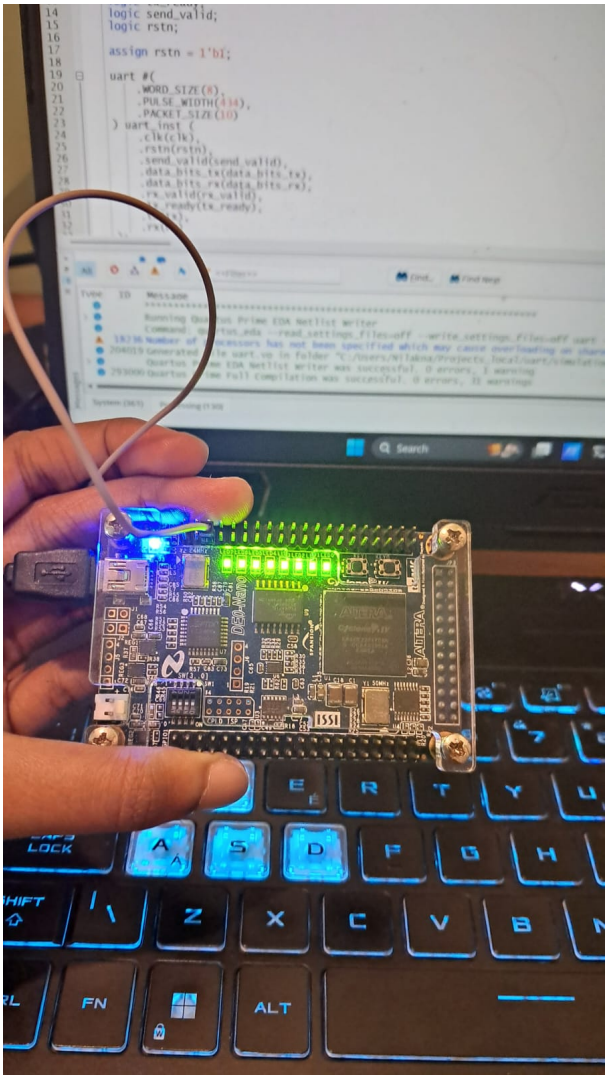


Figure 20: RX is '1 when transmitter sends no data

References

[1] Texas Instruments, *DEO Nano User's Guide*, Texas Instruments, May 2015, rev. A. [Online]. Available: https://www.ti.com/lit/ug/tidu737/tidu737.pdf