Design and Verification of Universal Asynchronous Reciever Transmitter (UART) Protocol

(by Ch.Anirudh)

1.Introduction:

Universal Asynchronous Receiver-Transmitter (**UART**) is a widely used communication protocol that enables serial data transfer between devices. Unlike synchronous communication protocols, UART does not require a shared clock between the transmitter and receiver, making it simple and efficient for data exchange over short distances. It is commonly used in **embedded systems, microcontrollers**, and computer peripherals for serial communication.

UART operates by transmitting data one bit at a time over a single data line. It consists of two main components: **UART Transmitter (TX)** and **UART Receiver (RX)**. The transmitter converts parallel data into a serial stream, while the receiver reconstructs the original parallel data. A **baud rate generator** controls the transmission speed, ensuring **proper synchronization** between the sender and receiver.

This project implements a complete **UART protocol**, including the transmitter, receiver, baud rate generator, and a protocol module. The design is verified through simulation and synthesis, ensuring correct data transmission and reception. The implementation adheres to standard UART specifications and can be used in various real-world applications such as embedded systems, IoT devices, and industrial automation.

2.Overview:

This project focuses on designing and implementing a **UART protocol** using Verilog/SystemVerilog. The system consists of the following major components:

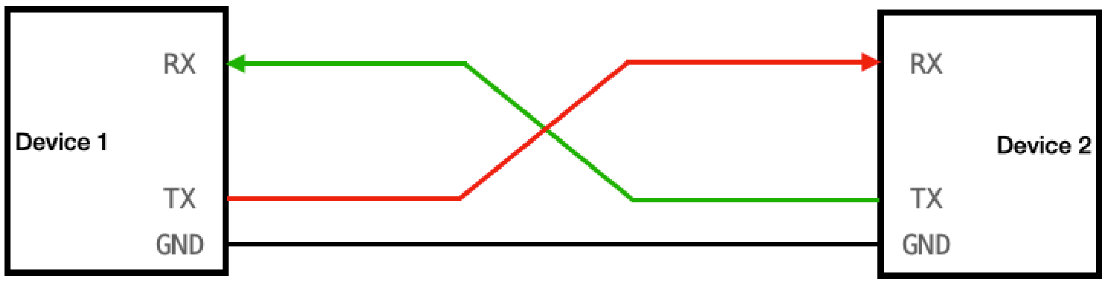
- **UART Transmitter (TX):** Converts parallel data into a serial stream.
- **UART Receiver (RX):** Converts the received serial data back into parallel format.
- **UART Protocol Module:** Manages the communication process between TX and RX.
- **Baud Rate Generator:** Ensures data transmission at a predefined speed.
- **Testbenches:** Verifies the functionality of individual modules and the complete UART system.

The design is simulated and synthesized to verify its correctness and efficiency.

3.Design and Implementation:

The UART protocol is designed using **Verilog/SystemVerilog** with a modular approach, ensuring flexibility and ease of debugging. The system consists of a **UART Transmitter (TX)**, which converts parallel data into a serial stream by adding start and stop bits, and a **UART Receiver (RX)**, which detects the start bit, samples the incoming data, and reconstructs it into parallel format. A **Baud Rate Generator** ensures synchronized communication by generating clock signals at the desired transmission speed. The **UART Protocol Module** manages data flow and ensures proper handshaking between the transmitter and receiver. The design is verified using **SystemVerilog testbenches**.

- **Transmitter (TX):**
    - Accepts an 8-bit parallel data input.
    - Adds start and stop bits.
    - Sends data bit-by-bit through tx_serial.
- **Receiver (RX):**
    - Detects the start bit.
    - Samples the incoming bits based on the baud rate.
    - Reconstructs the parallel data output.
- **Baud Rate Generator:**
    - Generates the clock signal for data sampling.
    - Configurable to support different baud rates.

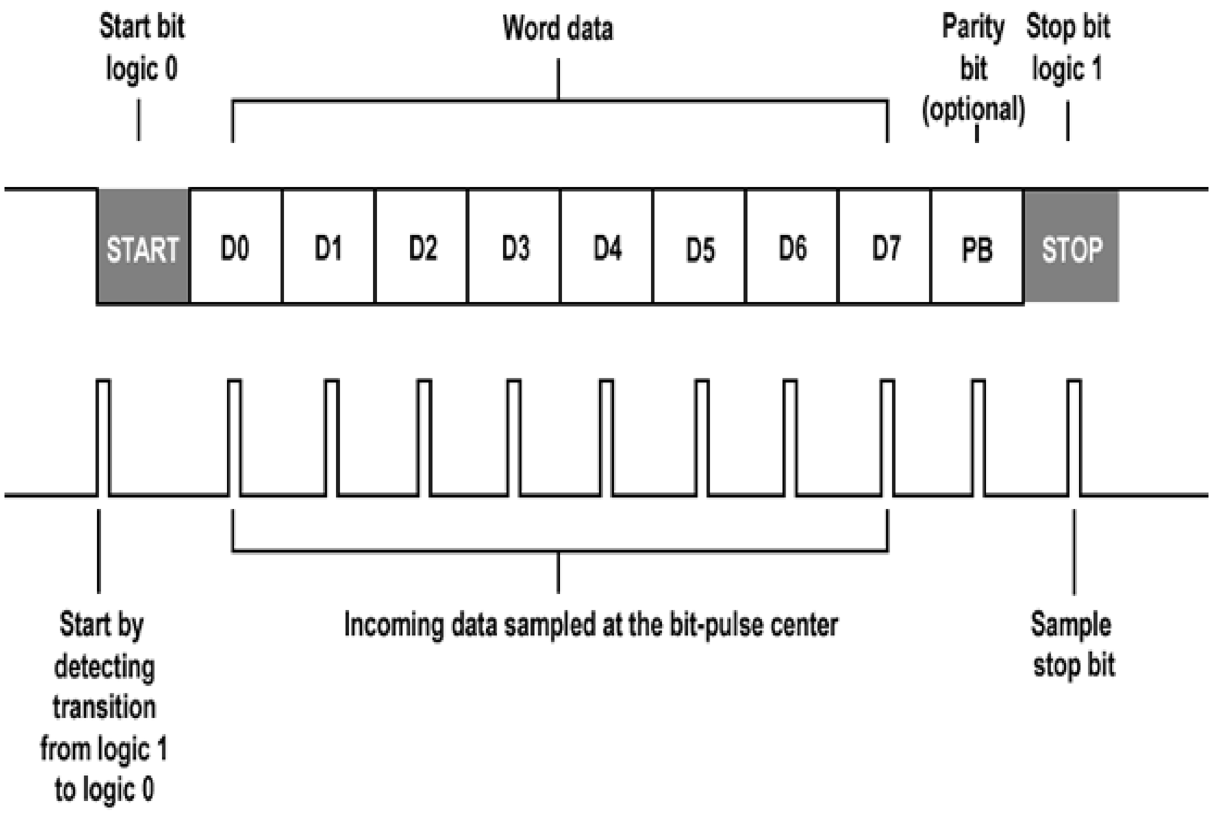


Fig(a): UART Communication Wiring

4. Operations

- **Data Transmission:** The TX module converts the parallel input into a serial bit stream with start and stop bits.
- **Data Reception:** The RX module samples the serial data and reconstructs it into parallel format.
- **Baud Rate Synchronization:** The baud rate generator ensures both TX and RX operate at the same frequency.
- **Handshake Mechanism:** Ensures proper communication between sender and receiver.

The system is tested with different data values to confirm accurate transmission and reception.



Fig(b): UART Frame

5.RTL Code UART TX:

```
module uart_tx1 (
    input wire clk,
    input wire rst,
    input wire baud_tick,
    input wire [7:0] data_in,
    input wire send,
    output reg tx_serial,
    output reg tx_done
);

typedef enum logic [2:0] {IDLE, START, DATA, STOP} state_t;
state_t current_state, next_state;

reg [3:0] bit_index; // To track the current bit being sent
reg [7:0] shift_reg; // Shift register for data transmission

// State transition logic
always_ff @(posedge clk or posedge rst) begin
    if (rst) begin
        current_state <= IDLE; // Initialize to IDLE on reset
        tx_serial <= 1; // Idle state is high
        tx_done <= 0; // Reset tx_done
        bit_index <= 0; // Reset bit index
    end else begin
        current_state <= next_state; // Update current state on clock edge
    end
end

// State machine logic
always_comb begin
    // Default assignments to avoid latches
    next_state = current_state;

    case (current_state)
        IDLE: begin
          if (send) begin
            next_state = START; // Transition to START on send
          end
        end

        START: begin
          next_state = DATA; // Transition to DATA after start bit
        end

        DATA: begin
          if (baud_tick) begin
            if (bit_index == 7) begin
              next_state = STOP; // Move to STOP after the last data bit
            end
          end
        end

        STOP: begin
          if (baud_tick) begin
            next_state = IDLE; // Return to IDLE after stop bit
          end
        end

        default: next_state = IDLE;
    endcase
end

// Output and data handling
always_ff @(posedge clk) begin
```

```
    case (current_state)
        IDLE: begin
            tx_done <= 0;
            if (send) begin
                shift_reg <= data_in; // Load data
                bit_index <= 0;
                tx_serial <= 1; // Idle state for serial line
            end
        end

        START: begin
            tx_serial <= 0; // Start bit
        end

        DATA: begin
            if (baud_tick) begin
                tx_serial <= shift_reg[bit_index]; // Transmit current bit
                bit_index <= bit_index + 1; // Move to next bit
            end
        end

        STOP: begin
            if (baud_tick) begin
                tx_serial <= 1; // Stop bit
                tx_done <= 1; // Indicate completion
            end
        end
    endcase
end

endmodule
```

UART RX:

```
module uart_rx1 (
input wire clk,
input wire rst,
input wire rx_serial,
input wire baud_tick,
output reg [7:0] data_out,
output reg rx_done
);
typedef enum logic [2:0] {IDLE, START, DATA, STOP} state_t;
state_t current_state, next_state;

reg [3:0] bit_index; // To track the current bit being received
reg [7:0] shift_reg; // Shift register for received data

// State transition logic
always_ff @(posedge clk or posedge rst) begin
  if (rst) begin
    current_state <= IDLE; // Initialize to IDLE on reset
    rx_done <= 0; // Reset rx_done
    shift_reg <= 0; // Reset shift register
    bit_index <= 0; // Reset bit index
    $display("Time: %0t | Reset: Moving to IDLE state", $time);
  end else begin
    current_state <= next_state; // Update current state on clock edge
  end
end

// State machine logic
always_ff @(posedge clk) begin
  case (current_state)
    IDLE: begin
```

```
        rx_done <= 0; // Reset rx_done in IDLE
        if (~rx_serial) begin // Start bit is low
            next_state <= START;
            $display("Time: %0t | Start bit detected, moving to START state", $time);
        end else begin
            next_state <= IDLE;
        end
    end

    START: begin
        if (baud_tick) begin
            next_state <= DATA; // Move to data state after start bit
            bit_index <= 0; // Reset bit index
            $display("Time: %0t | Start bit received", $time);
        end else begin
            next_state <= START; // Wait for baud tick
        end
    end

    DATA: begin
        if (baud_tick) begin
            shift_reg <= {rx_serial, shift_reg[7:1]}; // Shift in the received bit
            $display("Time: %0t | Received Data Bit %b: %b", $time, bit_index, rx_serial);
            if (bit_index == 7) begin
                next_state <= STOP; // Move to STOP state
                $display("Time: %0t | Last data bit recieved, moving to stop state", $time);
            end else begin
                bit_index <= bit_index + 1;
                next_state <= DATA;
            end
        end else begin
            next_state <= DATA; // Wait for baud tick
        end
        end

        STOP: begin
          if (baud_tick) begin
            if (rx_serial == 1) begin // Stop bit high
              data_out <= shift_reg; // Load received data
              rx_done <= 1; // Indicate reception done
              $display("Time: %0t | Stop bit received, data_out: %h", $time, data_out);
            end else begin
              $display("Time: %0t | Error: Stop bit not high", $time);
            end
            next_state <= IDLE; // Move back to IDLE
          end else begin
            next_state <= STOP; // Wait for baud tick
          end
        end

        default: next_state <= IDLE;
    endcase
  end
endmodule
```

BAUD RATE Generator:

```
module baud_generator #(parameter BAUD_RATE = 9600, CLOCK_FREQ = 50000000)
(
    input wire clk,
    input wire rst,
    output reg baud_tick
    );
    localparam TICKS = CLOCK_FREQ / BAUD_RATE;
    reg [15:0] tick_counter;

    always @(posedge clk or posedge rst) begin
```

```
     if (rst) begin
       tick_counter <= 0;
       baud_tick <= 0;
     end else begin
       if (tick_counter == TICKS - 1) begin
        tick_counter <= 0;
        baud_tick <= 1;
       end else begin
         tick_counter <= tick_counter + 1;
         baud_tick <= 0;
       end
     end
    end

endmodule
```

UART Protocol:

```
module uart_protocol1#(parameter BAUD_RATE = 9600, CLOCK_FREQ = 50000000)(
    input wire clk,
    input wire rst,
    input wire [7:0] tx_data, //data to transmit
    input wire tx_start,     // signal to start transmission
    input wire rx_serial,   // serial input for rx
    output wire tx_serial,     // serial output for tx
    output wire [7:0] rx_data,   // recieved data for output
    output wire tx_done,       // tx done flag
    output wire rx_done      // rx done flag
    );
      //internal signals
      wire baud_tick;
      //instantiate baud generator
      baud_generator #(
        .BAUD_RATE (BAUD_RATE),
          .CLOCK_FREQ(CLOCK_FREQ)
        )baud_gen_inst(
         .clk (clk),
         .rst(rst),
         .baud_tick(baud_tick)
         );
       //instantiate uart tx
       uart_tx1 uart_tx_inst(.clk(clk),
                     .rst(rst),
                     .baud_tick(baud_tick),
                     .data_in(tx_data),
                     .send(tx_start),
                     .tx_serial(tx_serial),
                     .tx_done(tx_done)
                     );
       // instantiate uart rx
       uart_rx1 uart_rx_inst (.clk(clk),
                      .rst(rst),
                      .rx_serial(rx_serial),
                      .baud_tick(baud_tick),
                      .data_out(rx_data),
                      .rx_done(rx_done)
                      );
endmodule
```

6.Test Benches:

UART Rx TB:

```
module tb1_uart;


  // Parameters for baud rate and clock frequency
  parameter BAUD_RATE = 9600;
  parameter CLOCK_FREQ = 50000000;
```

```
// Signals
reg clk;
reg rst;
reg rx_serial;
reg baud_tick;
wire [7:0] data_out;
wire rx_done;

// Instantiate the receiver module
uart_rx1 uut (
   .clk(clk),
   .rst(rst),
   .rx_serial(rx_serial),
   .baud_tick(baud_tick),
   .data_out(data_out),
   .rx_done(rx_done)
);

// Instantiate the baud generator
baud_generator #(
   .BAUD_RATE(BAUD_RATE),
   .CLOCK_FREQ(CLOCK_FREQ)
) baud_gen (
   .clk(clk),
   .rst(rst),
   .baud_tick(baud_tick)
);

// Clock generation
always #10 clk = ~clk; // 50 MHz clock

// Serial transmission task
task send_serial_data(input [7:0] data);
```

```
    integer i;
    begin
        // Send start bit (0)
        rx_serial = 0;
        @(posedge baud_tick);

        // Send 8 data bits (LSB first)
        for (i = 0; i < 8; i = i + 1) begin
            rx_serial = data[i];
            @(posedge baud_tick);
        end

        // Send stop bit (1)
        rx_serial = 1;
        @(posedge baud_tick);
    end
endtask

// Testbench procedure
initial begin
    // Initialize signals
    clk = 0;
    rst = 1;
    rx_serial = 1; // Idle state for serial line
    #50 rst = 0; // Release reset

    // Wait for some time
    #100;

    // Transmit first byte: 0xAA
    $display("Time: %0t | Sending byte: 0xAA", $time);
    send_serial_data(8'hAA);
    wait(rx_done);
```

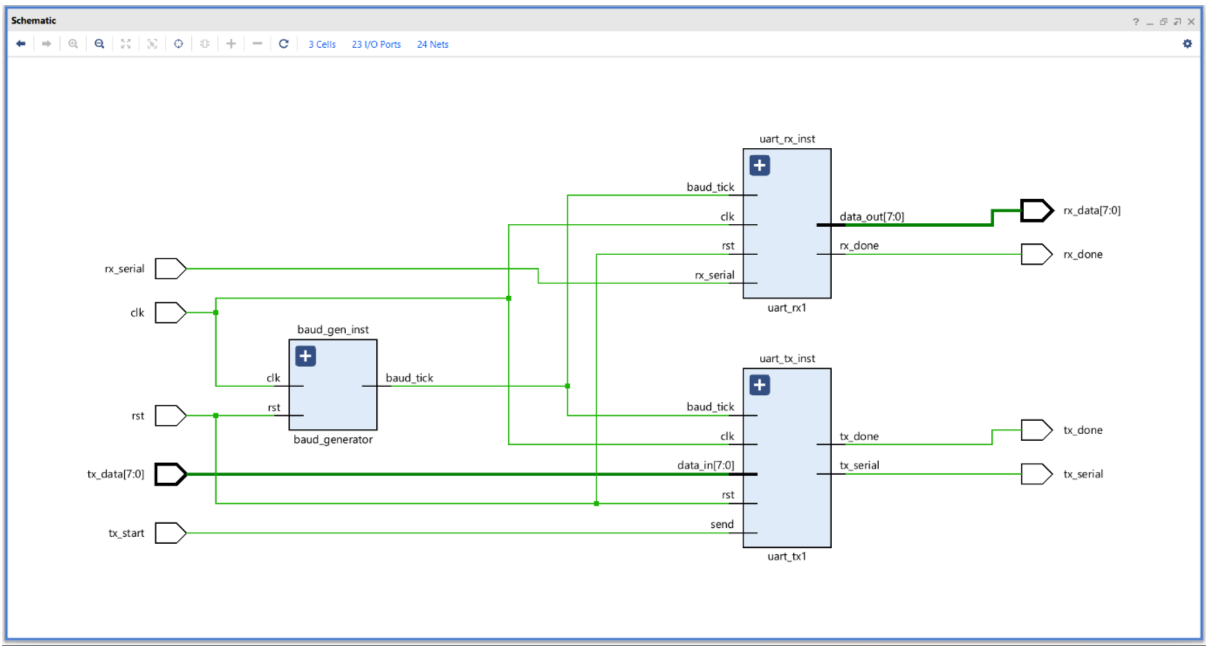
```
        #100;
        $display("Time: %0t | recieved byte: 0x%h | rx_done: %b", $time, data_out, rx_done);
        //end simulation
        #100;
        $finish;
    end

endmodule
```

UART Protocol TB:

```
module tb_uart_protocol1;
    // Parameters
    parameter BAUD_RATE = 9600;
    parameter CLOCK_FREQ = 50000000;

    // Signals
    reg clk;
    reg rst;
    reg [7:0] tx_data;
    reg tx_start;
    wire tx_serial;
    wire [7:0] rx_data;
    wire tx_done;
    wire rx_done;
    reg rx_serial;

    // Clock generation
    initial begin
        clk = 0;
        forever #10 clk = ~clk; // 50 MHz clock
    end
```

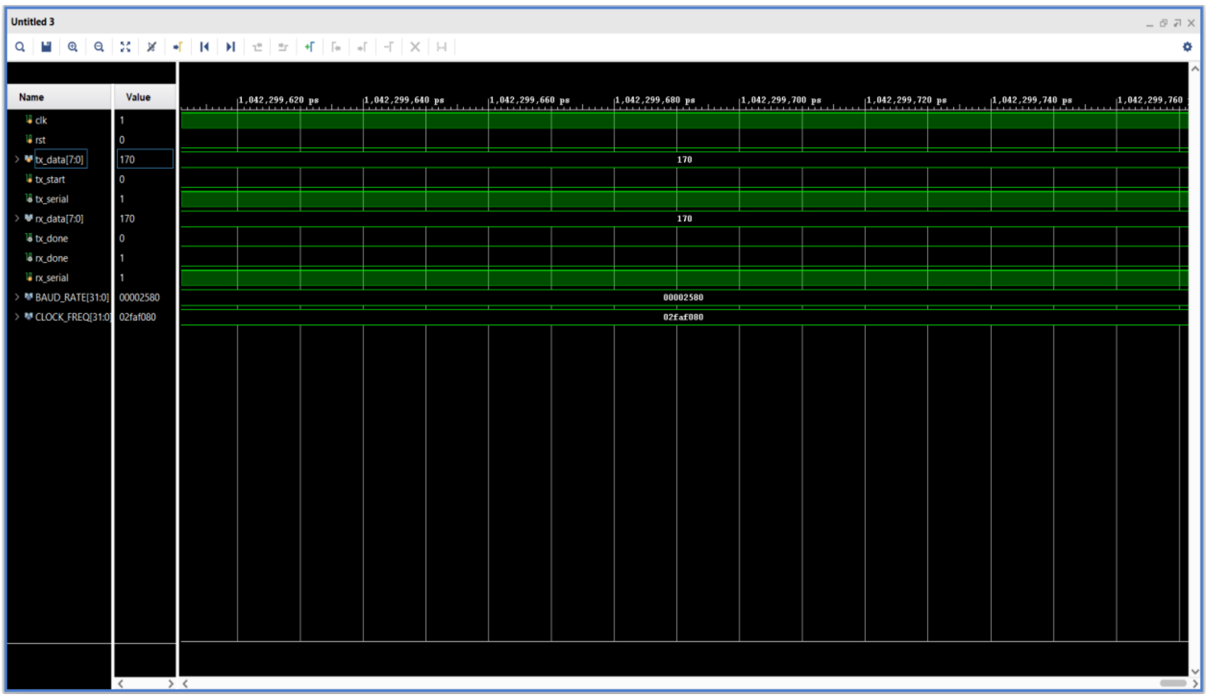
```
// Instantiate the UART protocol module
uart_protocol1 #(
    .BAUD_RATE(BAUD_RATE),
    .CLOCK_FREQ(CLOCK_FREQ)
) uut (
    .clk(clk),
    .rst(rst),
    .tx_data(tx_data),
    .tx_start(tx_start),
    .tx_serial(tx_serial),
    .rx_serial(rx_serial),
    .rx_data(rx_data),
    .tx_done(tx_done),
    .rx_done(rx_done)
);

// Task for applying reset
task apply_reset();
    begin
        rst = 1;
        #200; // Hold reset for 100 ns
        rst = 0;
    end
endtask

// Task to send and verify UART data
task send_data(input [7:0] data_to_send);
    begin
        tx_data = data_to_send;
        tx_start = 1;
        #20 tx_start = 0; // Clear start signal

        wait(tx_done); // Wait until transmission is done
        $display("Time: %0t | Sent Data: 0x%02h", $time, data_to_send);

        wait(rx_done); // Wait until reception is done
        if (rx_data == data_to_send) begin
          $display("Time: %0t | Received Data: 0x%02h | Test Passed", $time, rx_data);
        end else begin
          $fatal("Time: %0t | Received Data: 0x%02h |Test Failed", $time, rx_data);
        end
      end
    endtask

    // Loopback: Drive RX serial input with TX serial output
    always @(posedge clk) begin
      rx_serial <= tx_serial;
    end

    // Testbench logic
    initial begin
      $display("Starting UART Protocol Testbench...");

      // Apply reset
      apply_reset();

      // Test case 1: Send 0xAA
      send_data(8'hAA);



      // Simulation complete
      $display("Simulation Complete: All tests passed.");
      #500 $finish;
    end
  endmodule
```

7.Elaborated Design:



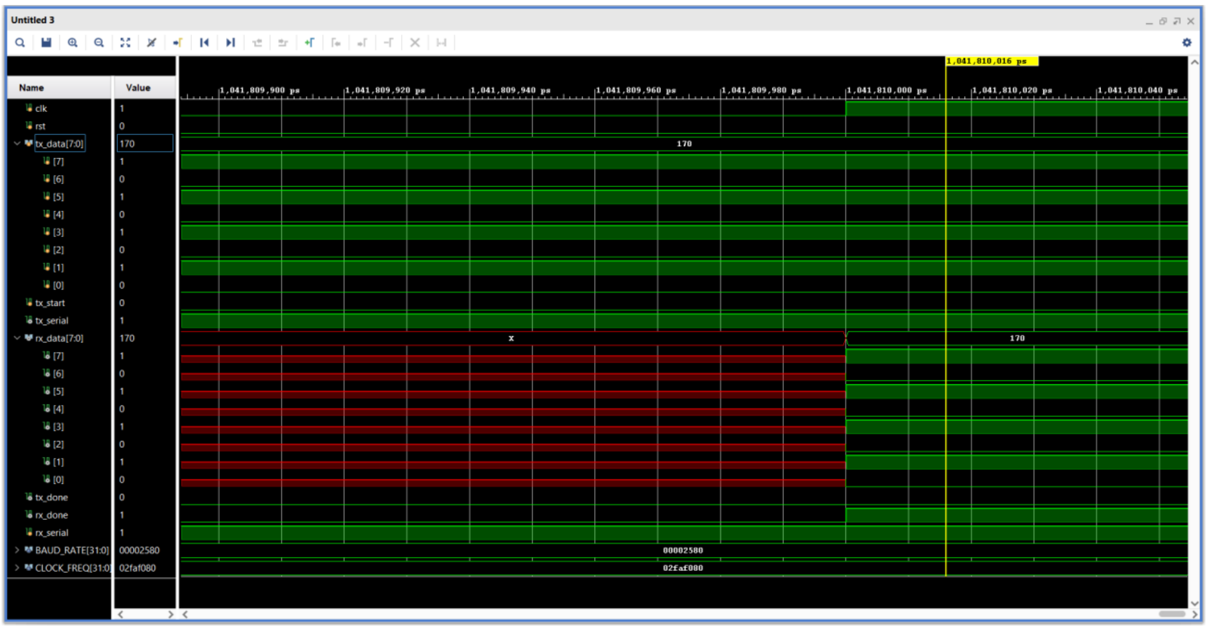
Fig(c): Elaborated design of UART

8.Simulations:



Name	Value
clk	1
rst	0
tx_data[7:0]	170
tx_start	0
tx_serial	1
rx_data[7:0]	170
tx_done	0
rx_done	1
rx_serial	1
BAUD_RATE[31:0]	00002580
CLOCK_FREQ[31:0]	02faf080

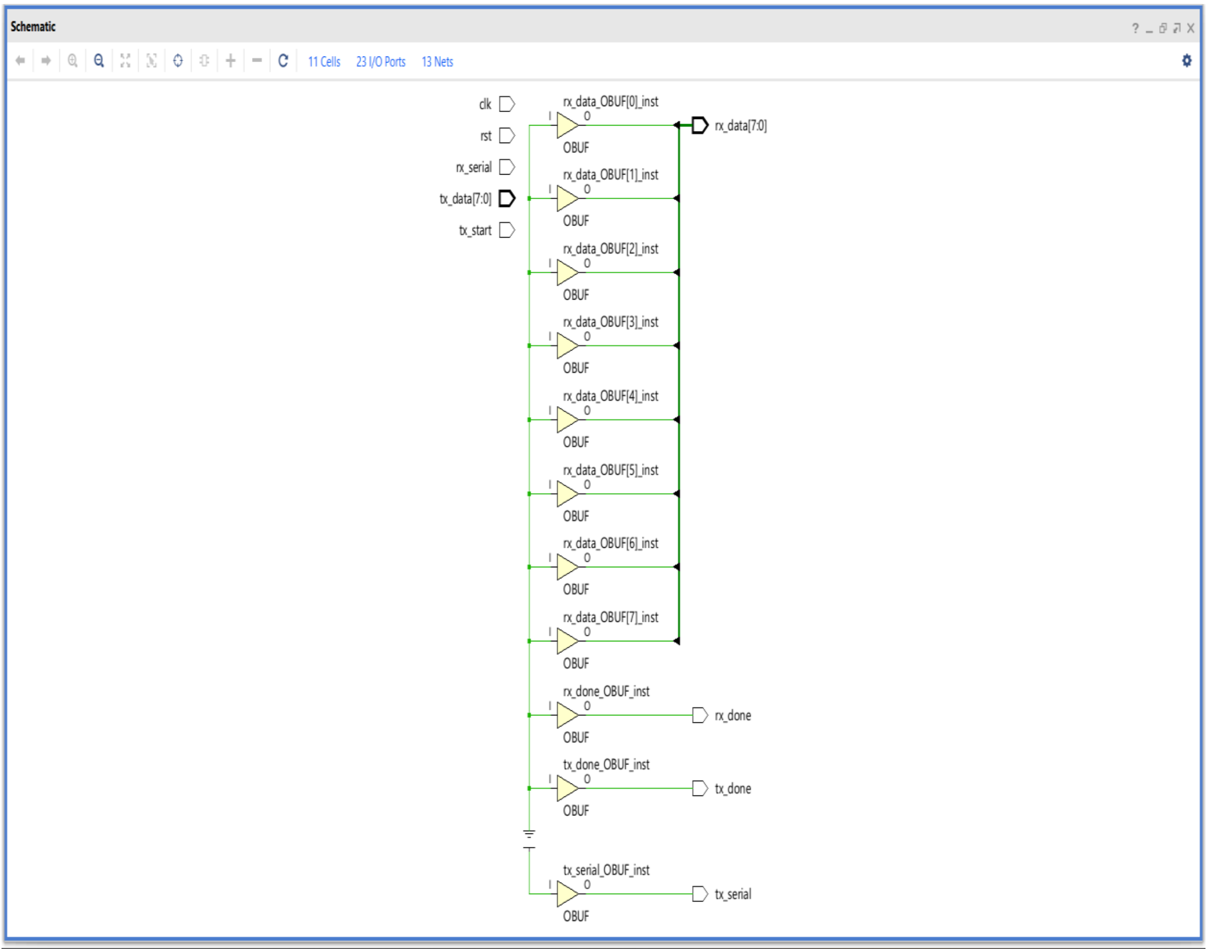
Fig(d): UART Simulation 1



Name	Value
clk	1
rst	0
tx_data[7:0]	170
[7]	1
[6]	0
[5]	1
[4]	0
[3]	1
[2]	0
[1]	1
[0]	0
tx_start	0
tx_serial	1
rx_data[7:0]	170
[7]	1
[6]	0
[5]	1
[4]	0
[3]	1
[2]	0
[1]	1
[0]	0
tx_done	0
rx_done	1
rx_serial	1
BAUD_RATE[31:0]	00002580
CLOCK_FREQ[31:0]	02faf080

Fig(e): UART Simulation 2

9.Schematic



Fig(f): UART Schematic

10.Advantages:

- Simple and widely used protocol.
- No need for a shared clock between TX and RX.
- Low hardware complexity compared to SPI and I2C.
- Supports multiple baud rates.

Disadvantages:

- Slower compared to SPI and I2C.
- Limited to point-to-point communication.
- No built-in error detection (except optional parity bit).

11.Applications:

- Microcontrollers and Embedded Systems
- Serial Communication between Computers and Peripherals
- Industrial Automation and Control Systems
- Wireless Communication Modules (Bluetooth, GSM, GPS)
- Debugging and Firmware Uploading

12.Conclusion:

This project successfully implements a fully functional UART protocol, including TX, RX, baud rate generator, and testbenches. The design is validated through simulation and synthesis, confirming accurate data transmission and reception. UART remains a fundamental communication standard due to its simplicity and efficiency. This implementation can be further enhanced by adding error detection mechanisms and flow control for more robust communication.

---- The End--