Project 105: UARTProtocol

A Comprehensive Study of Advanced Digital Circuits

By: Gati Goyal ,Abhishek Sharma, Nikunj Agrawal, Ayush Jain

Documentation Specialist: Dhruv Patel & Nandini Maheshwari

Contents

1 Introduction

The Universal Asynchronous Receiver-Transmitter (UART) is a widely used communication protocol for serial data transmission between devices. It enables asynchronous communication, meaning that data is sent one bit at a time, with the timing determined by the baud rate rather than a shared clock. UART is fundamental in many embedded systems and digital devices for transmitting and receiving data without the need for complex synchronization.

In UART, data transmission typically involves a start bit, followed by a sequence of data bits, an optional parity bit for error checking, and a stop bit to signify the end of the data frame. The protocol allows devices to communicate over a single wire, using standard baud rates to control the speed of transmission. Its simplicity, combined with low resource requirements, makes it ideal for embedded applications such as sensor data communication, microcontroller interfacing, and debugging.

While UART generally handles unsigned binary data, its fundamental principle of serial data transmission is central to many digital systems. The protocol's simplicity and reliability make it an essential tool in the development of systems where straightforward, low-overhead communication is needed.

2 Background

The Universal Asynchronous Receiver-Transmitter (UART) is a serial communication protocol widely used for transmitting data between devices. UART operates asynchronously, meaning that data is transmitted one bit at a time without the need for a shared clock signal. This simplifies communication in embedded systems, as devices do not need to be synchronized with each other beyond the agreement on a common baud rate.

The UART protocol involves a simple frame structure, typically consisting of a start bit, data bits, an optional parity bit for error checking, and stop bits to indicate the end of the transmission. The data bits can range from 5 to 9 bits, with the start bit signaling the beginning of the data transfer and the stop bit marking the end of the transmission.

In UART communication, the timing for sending and receiving bits is determined by the baud rate, which must be the same on both transmitting and receiving devices to ensure proper data reception. The simplicity and reliability of UART make it particularly well-suited for low-speed, short-distance communication, such as interfacing microcontrollers with sensors, external devices, and other embedded systems.

While UART is primarily used for unsigned binary data, it is commonly used for communication in various applications, including debugging, serial data transfer, and control systems. The protocol's straightforward design and widespread support in embedded systems make it a versatile choice for many communication needs.

3 Structure and Operation

The Universal Asynchronous Receiver-Transmitter (UART) is designed for serial communication, transmitting and receiving data one bit at a time. Its structure involves several key components that work together to enable reliable data transmission and reception between devices.

3.1 Key Components

- **Transmitter**: The component responsible for converting parallel data (from a microcontroller or system) into a serial stream, which is then sent bit-by-bit over a single communication line.
- **Receiver**: The counterpart of the transmitter, it receives the serial data, converts it back into parallel form, and passes it to the receiving system.
- **Shift Register**: Used in both the transmitter and receiver, this component shifts the data bits one by one into or out of the system at the correct baud rate.
- **Baud Rate Generator**: Controls the timing for data transmission and reception, ensuring that both devices communicate at the same speed (baud rate).

- **Start, Data, and Stop Bits**: The start bit indicates the beginning of the transmission, the data bits represent the transmitted data, and the stop bit(s) signal the end of the transmission.
- **Parity Bit (optional)**: Used for error detection, the parity bit can be even or odd, and it ensures that the number of ones in the transmitted data follows the specified parity scheme.

The operation of the UART protocol can be summarized as follows:

1. **Data Transmission Initialization**: In the transmitter, parallel data (often from a microcontroller register) is prepared for serial transmission. The start bit is added, followed by the data bits, optional parity bit, and stop bits.
2. **Serial Bit Transmission**: The transmitter shifts each bit from the data register one at a time, starting with the start bit, followed by the data bits, parity bit (if used), and ending with the stop bit. The data is transmitted at the specified baud rate.
3. **Data Reception**: On the receiving end, the receiver listens for a start bit, which indicates the beginning of incoming data. The receiver shifts each bit one by one, storing them in a shift register.
4. **Data Conversion**: After receiving all the data bits (and the optional parity bit), the receiver converts the serial data back into parallel format, making it accessible to the receiving device.
5. **Error Detection (optional)**: If a parity bit is used, the receiver checks it against the received data to detect any errors in transmission. If errors are found, the data may be discarded or flagged for retransmission.
6. **Output Data**: After the data is shifted and error checking is completed (if applicable), the receiver outputs the parallel data for use by the receiving device.

The UART protocol is simple, yet effective, for reliable, low-speed serial communication. It is used in a variety of applications, such as microcontroller communication, sensor data acquisition, and serial debugging. The protocol's flexibility with respect to baud rate, data bits, parity, and stop bits makes it adaptable to a wide range of devices and systems.

4 Implementation in System Verilog

The following RTL code implements the Uart Protocol for the Transmitter Module in System Verilog:

Listing 1: Uart Protocol for Transmitter Module

```

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
);
    // State encoding
    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 the 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
```

The following RTL code implements the Uart protocol for the baud generator for 50Mhz Clock Frequency with 9600 Baud Rate in System Verilog:

Listing 2: baud generator for 50Mhz Clock Frequency with 9600 Baud Rate

```

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
```

The following RTL code implements the Uart protocol for the Receiver Module in System Verilog:

Listing 3: Receiver Module for Uart Protocol

```
    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
```

```
                            after last data bit
                        $display("Time: %0t | Last data bit received, moving to STOP state", $time);
                    end else begin
                        bit_index <= bit_index + 1; // Move to next bit
                        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 should be high
                        data_out <= shift_reg;   // Load received data
                        rx_done <= 1;            // Indicate reception is 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 state
                end else begin
                    next_state <= STOP; // Wait for baud tick
                end
            end

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

The following RTL code implements the Uart protocol in System Verilog:

Listing 4: Uart Protocol

```

module uart_protocol1 #(
    parameter BAUD_RATE = 9600,
    parameter 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 receiver
    output wire tx_serial,       // Serial output for transmitter
    output wire [7:0] rx_data,   // Received data output
    output wire tx_done,         // Transmission done flag
    output wire rx_done          // Reception 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 transmitter
    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 receiver
    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
```

5 Test Bench

The following test bench verifies the functionality of the Receiver Module Testbench :

Listing 5: Receiver Module Testbench

```

 //its rx testbench
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),
```

```verilog
        .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);
        $display("Time: %0t | Received byte: 0x%h | rx_done: %b",
            $time, data_out, rx_done);

        // Transmit second byte: 0xCC
        #100; // Wait before sending next byte
        $display("Time: %0t | Sending byte: 0xCC", $time);
        send_serial_data(8'hCC);
        wait(rx_done);
        $display("Time: %0t | Received byte: 0x%h | rx_done: %b",
           $time, data_out, rx_done);

        // End simulation
        #100;
        $finish;
    end
endmodule
```

The following test bench verifies the functionality of the Uart Protocol Testbench

Listing 6: Uart Protocol Testbench

```

    'timescale 1ns/1ps

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),
        .rx_serial(rx_serial),
        .tx_serial(tx_serial),
        .rx_data(rx_data),
        .tx_done(tx_done),
        .rx_done(rx_done)
    );

    // Task for applying reset
    task apply_reset();
        begin
            rst = 1;
            #100; // 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);

        // Test case 2: Send 0x3C
        send_data(8'h3C);

        // Simulation complete
        $display("Simulation Complete: All tests passed.");
        #100 $finish;
    end
endmodule
```

6 Advantages and Disadvantages

6.1 Advantages

- **Simplicity**: UART communication is straightforward to implement, requiring only a minimal number of control signals (start, data, stop bits) and a shared baud rate for communication.
- **Low Cost**: UART uses a simple design with fewer components compared to more complex communication protocols, making it cost-effective for many embedded systems.
- **No Need for Synchronization**: Since UART is asynchronous, devices don't need to share a clock signal, simplifying the hardware and making it easier to implement across different systems.
- **Wide Compatibility**: UART is widely supported by most microcontrollers, computers, and peripheral devices, making it a versatile communication protocol in embedded systems.
- **Error Detection (optional)**: UART supports optional parity bits for error detection, allowing for simple error checking and ensuring data integrity during transmission.

6.2 Disadvantages

- **Limited Speed**: UART is typically slower compared to other communication protocols like SPI or I2C, making it less suitable for high-speed data transmission.
- **Short Range**: The distance over which UART can reliably transmit data is limited due to the absence of a clock signal, especially at higher baud rates.
- **Limited Scalability**: UART is designed for point-to-point communication, which means it is not ideal for connecting multiple devices in a network without additional multiplexing or bus management techniques.
- **No Built-In Synchronization**: Since it operates asynchronously, the baud rate must be carefully matched between transmitting and receiving devices. Mismatched baud rates can result in data loss or corruption.
- **Requires Manual Error Handling**: Although the optional parity bit allows for error detection, UART does not have built-in mechanisms for error correction, meaning retransmission or other corrective actions must be implemented manually.

7 Simulation Results

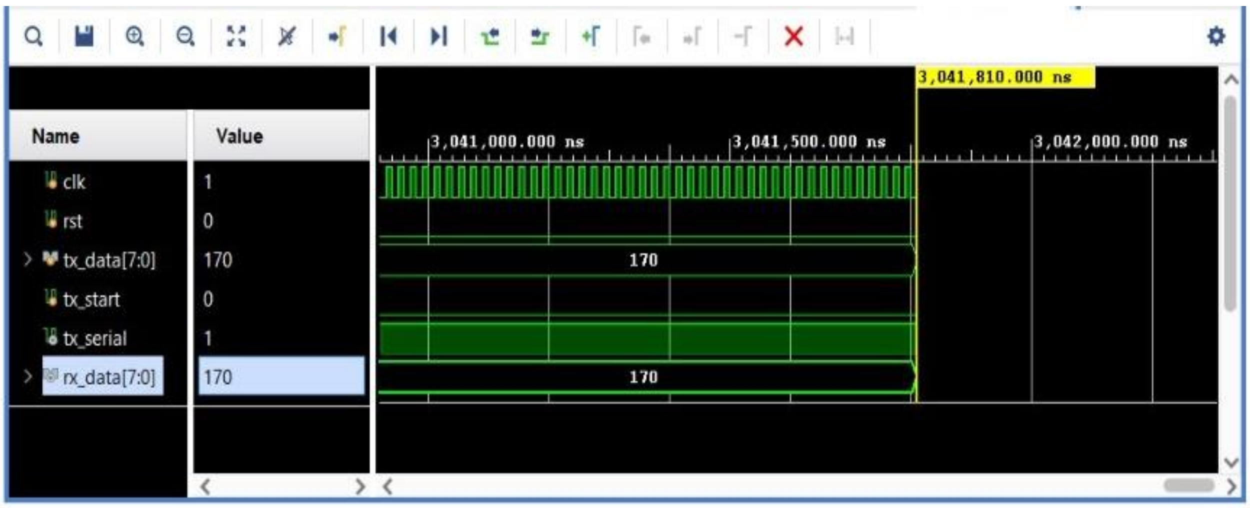

Figure 1: Simulation results ofUART Result TCL Console-1

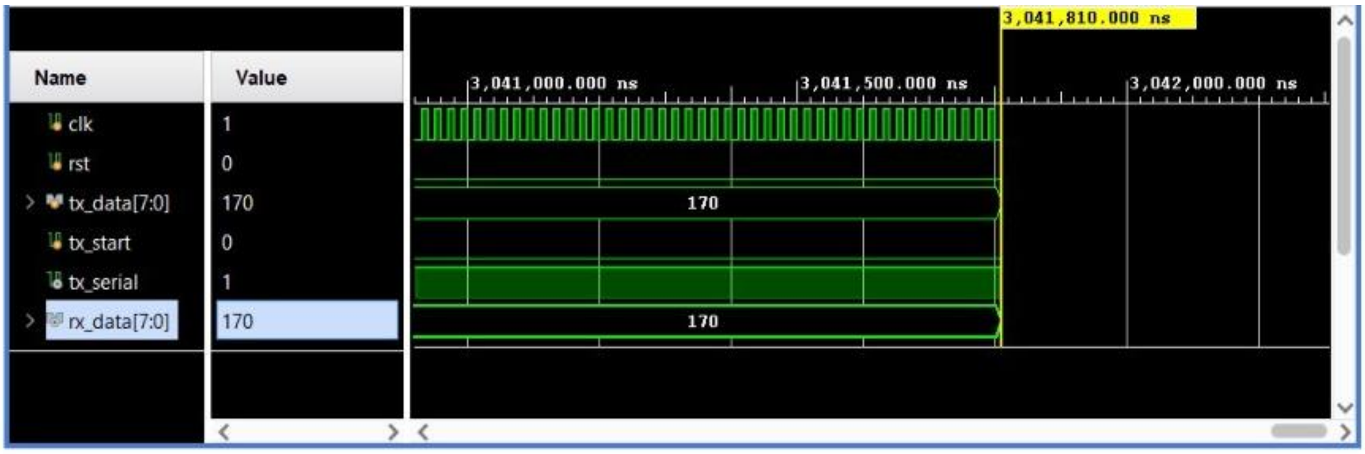

Figure 2: Simulation results ofUART Result TCL Console-2

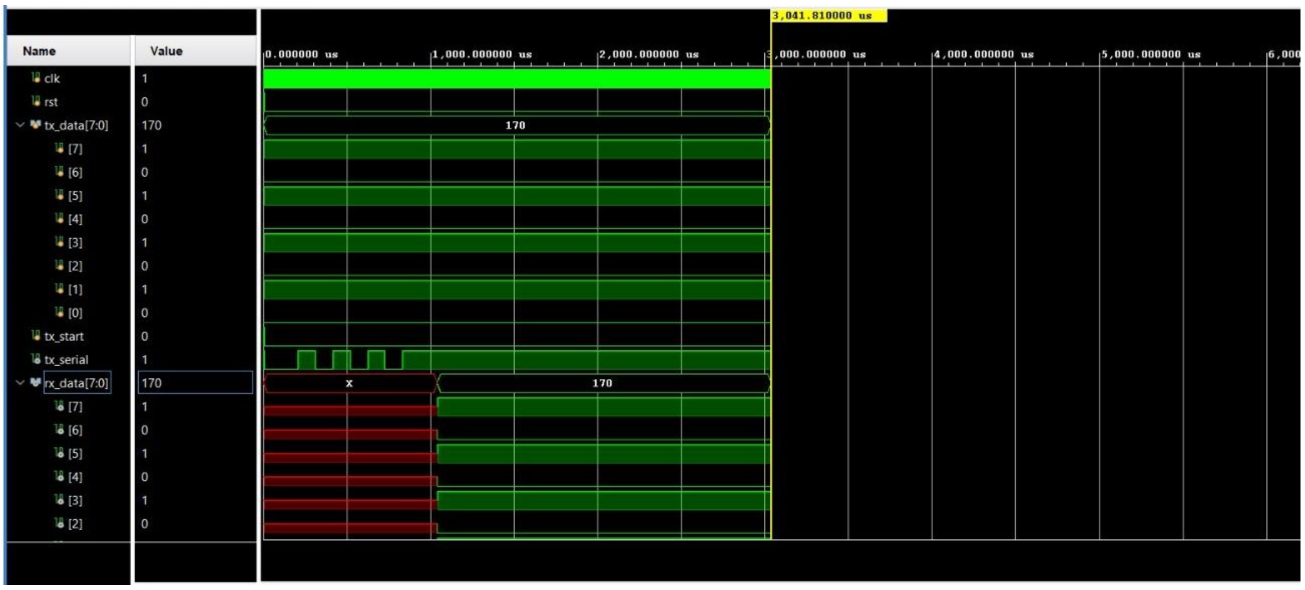

Figure 3: Simulation results of Output for Uart Protocol

8 Schematic Design

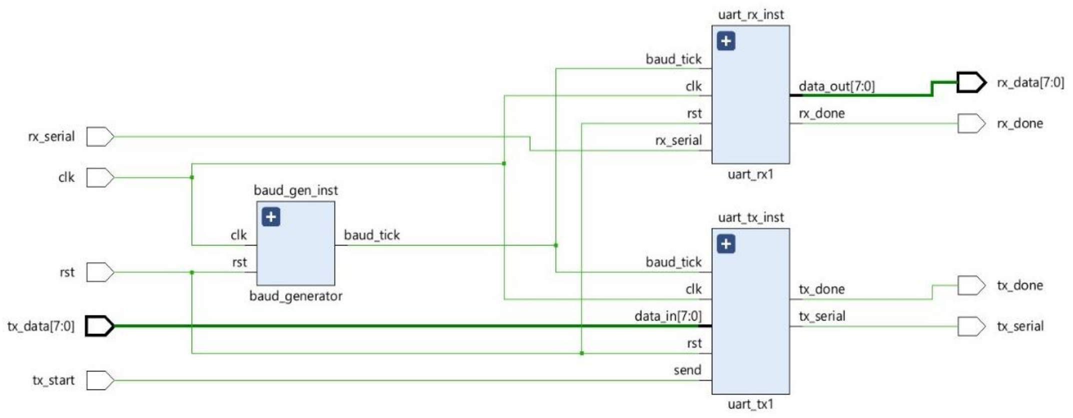

Figure 4: Schematic of Uart Protocol

9 Synthesis Design

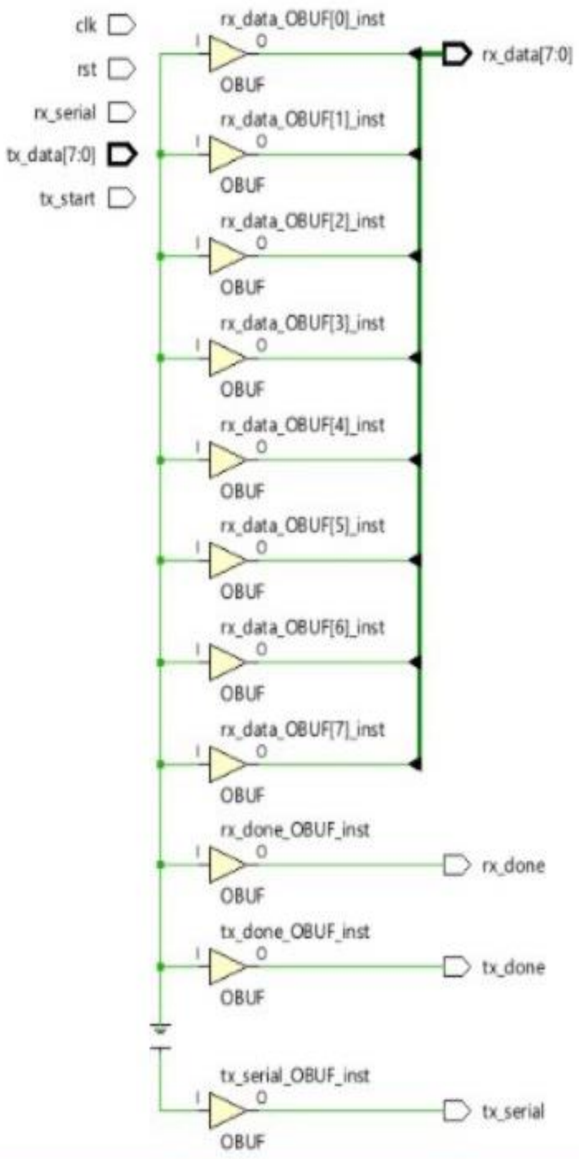

Figure 5: Synthesis of Uart Protocol

10 Conclusion

The UART protocol is a widely used and simple communication standard that enables reliable serial data transmission between devices. It operates asynchronously, transmitting data bit-by-bit, which allows devices to communicate without needing a shared clock signal. Its simplicity, ease of implementation, and low resource requirements make UART a popular choice in many embedded systems and applications that require low-speed, point-to-point communication.

While UART offers significant advantages such as low cost, wide compatibility, and simplicity, it also has certain limitations, including slower data transmission speeds, limited range, and scalability concerns in multi-device communication. Despite these drawbacks, UART remains an essential protocol for many applications, including microcontroller interfacing, sensor data acquisition, debugging, and more. Its optional error detection feature, using parity bits, further enhances its reliability in data transmission.

In conclusion, UART continues to be a versatile and widely adopted communication protocol in embedded systems, offering a straightforward solution for serial communication with minimal overhead, making it indispensable in numerous digital systems.

11 Frequently Asked Questions (FAQs)

11.1 What is UART?

The Universal Asynchronous Receiver-Transmitter (UART) is a communication protocol that en-ables asynchronous serial data transmission between devices, sending data one bit at a time without requiring a shared clock signal.

11.2 How does UART communication work?

In UART communication, data is transmitted starting with a start bit, followed by data bits, an optional parity bit for error detection, and stop bits. The receiver then reconstructs the data based on the predefined baud rate.

11.3 What are the advantages of using UART?

Advantages include its **simplicity**, **low cost**, **reliable data transmission**, **no need for synchronization**, **wide compatibility**, and optional **error detection** using parity bits.

11.4 What are the challenges associated with UART?

Challenges include **limited transmission speed**, **short communication range**, **limited scalability** (point-to-point communication), and the need for careful matching of baud rates between devices to ensure reliable data transmission.

11.5 How does UART differ from other communication protocols like SPI or I2C?

UART operates asynchronously with a single data line for transmission, whereas SPI and I2C are synchronous protocols that require additional lines and can support communication with multiple devices. UART is simpler but slower compared to these protocols.

11.6 Can UART support multiple devices?

UART is designed for **point-to-point communication**, meaning it is typically used to connect only two devices. However, it can support multiple devices with additional multiplexing or bus management schemes.

11.7 How does UART handle error detection?

UART can use an optional **parity bit** to detect errors during transmission. It supports **even** or **odd** parity, and the receiver can check if the number of ones in the data matches the expected parity.