EN 2111 – Electronic Circuit Design
UART Implementation in FPGA

Department of Electronic and Telecommunication Engineering
University of Moratuwa, Sri Lanka

Group 33

Submitted by

Wickramasinghe S.D.	220701X
Weragoda W.A.A.P.	220692R
Wijenayaka M.B.T.I.	220711D

Submitted on May 22, 2025

Contents

1 Introduction

UART, or Universal Asynchronous Receiver–Transmitter, is one of the world's most-used hardware communication protocols. It is a serial, full-duplex protocol that can be configured to operate at various baud rates. "Asynchronous" means there is no shared clock signal on the transmission medium; instead, both ends must agree on the same baud rate to correctly sample the bit stream.

In a typical UART design, the Transmitter (Tx) and Receiver (Rx) are implemented as separate modules. The Tx converts parallel data from a host bus into a serial bit stream, inserting start, stop, and optional parity bits. The Rx detects framing boundaries in the incoming serial stream, samples each bit at the configured baud rate, and reconstructs the original parallel word.

- **Transmitter (Tx):** Encodes parallel data into a serial format with start/stop bits.
- **Receiver (Rx):** Detects start bit, samples data bits, checks stop bit, and outputs parallel data.

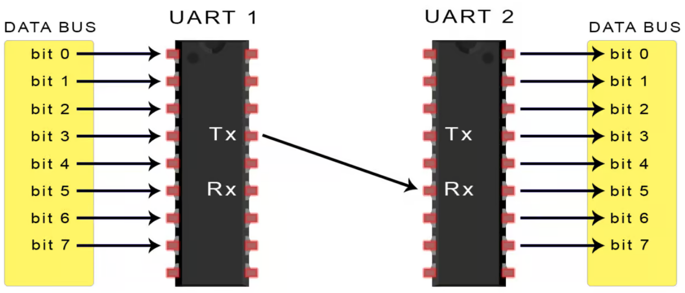

Figure 1: Conventional UART with separate Tx and Rx connected to a data bus

In this assignment, we go beyond the conventional split design by integrating both transmitter and receiver into a single *transceiver* module. This unified approach shares clock-division logic, control signals, and I/O buffers, thereby reducing resource utilization on the FPGA and simplifying timing coordination between Tx and Rx paths.

The objectives of this work are to:

1. Develop a Verilog RTL implementation of a fully integrated UART transceiver
2. Create a comprehensive testbench using Verilog, with waveform inspection in ModelSim, to verify protocol compliance, timing accuracy, and error detection.
3. Synthesize and deploy the transceiver design onto a DE0-Nano FPGA board, perform hardware-in-the-loop validation with peer devices, and capture real-time serial waveforms on an oscilloscope.

By combining Tx and Rx into a single RTL block, this design not only conserves FPGA logic resources but also ensures tight synchronization and simpler integration into larger serial-communication subsystems.

2 UART Transceiver Design

2.1 UART Configuration

This UART transceiver uses a common baud-rate generator and I/O buffering for both TX and RX. The configuration parameters are:

- System clock: `clk`
- Reset: active-low `rst_n`
- Baud Rate: defined by

$$\texttt{CLKS\_PER\_BIT} = \frac{\texttt{CLK\_FREQ}}{\texttt{BAUD\_RATE}} = \frac{50\,000\,000}{115\,200}$$

- Default parameters:
  - `CLK_FREQ` = 50 000 000 (50 MHz)
  - `BAUD_RATE` = 115 200 bps

2.2 UART Transmitter

The transmitter FSM has four states controlling the serial output `tx` and busy flag `tx_busy`:

- `IDLE`:
  - Drive `tx = 1` (line idle)
  - Clear `tx_clk_count` and `tx_bit_count`
  - Wait for `tx_start && !tx_busy` to latch `tx_data` into `tx_shift_reg` and assert `tx_busy`
- `START`:
  - Drive `tx = 0` for one bit period
  - Increment `tx_clk_count` until it reaches `CLKS_PER_BIT - 1`, then reset counter and move to `DATA`
- `DATA`:
  - Drive `tx = tx_shift_reg[0]` (LSB first)
  - On each full bit period, shift `tx_shift_reg` right by one and increment `tx_bit_count`
  - After 8 bits, transition to `STOP`
- `STOP`:
  - Drive `tx = 1` for one bit period
  - At end of period, deassert `tx_busy` and return to `IDLE`

2.3 UART Receiver

The receiver FSM samples the synchronized input `rx_d2` at mid-bit times and assembles bytes:

- **Input Synchronization:**
  - Two-flip-flop synchronizer (`rx_d1`, `rx_d2`) to avoid metastability
- `IDLE`:
  - Wait for `rx_d2 == 0` (start-bit falling edge)
  - Reset `rx_clk_count` and `rx_bit_count`
- `START`:
  - Count to midpoint: `CLKS_PER_BIT` – 12
  - Verify `rx_d2` still low; if valid, wait remainder of bit period then go to `DATA`
  - On false start, return to `IDLE`
- `DATA`:
  - At each mid-bit point, sample `rx_d2` into the MSB of `rx_shift_reg` and shift existing bits down
  - After 8 bits, transition to `STOP`
- `STOP`:
  - Wait one full bit period, then assert `rx_done` for one clock and transfer `rx_shift_reg` to `rx_data`
  - Return to `IDLE`

```verilog
module uart_transceiver(
  input  wire       clk,        // System clock
  input  wire       rst_n,      // Active low reset
  // UART TX Interface
  input  wire       tx_start,   // Start transmission
  input  wire [7:0] tx_data,    // Data to transmit
  output reg        tx_busy,    // Transmitter busy flag
  output reg        tx,         // Serial TX output
  // UART RX Interface
  input  wire       rx,         // Serial RX input
  output reg        rx_done,    // Reception complete flag
  output reg [7:0]  rx_data     // Received data byte
);

  // Parameters
  parameter CLK_FREQ    = 50_000_000;      // 50 MHz system clock
  parameter BAUD_RATE   = 115200;          // Desired baud rate
  localparam CLKS_PER_BIT = CLK_FREQ / BAUD_RATE;
```

```

// State definitions
localparam IDLE  = 2'b00;
localparam START = 2'b01;
localparam DATA  = 2'b10;
localparam STOP  = 2'b11;

// TX registers
reg [1:0]  tx_state;
reg [15:0] tx_clk_count;
reg [2:0]  tx_bit_count;
reg [7:0]  tx_shift_reg;

// RX registers
reg [1:0]  rx_state;
reg [15:0] rx_clk_count;
reg [2:0]  rx_bit_count;
reg [7:0]  rx_shift_reg;
reg        rx_d1, rx_d2;            // Double-flop synchronizer

// Synchronize RX
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    rx_d1 <= 1'b1;
    rx_d2 <= 1'b1;
  end else begin
    rx_d1 <= rx;
    rx_d2 <= rx_d1;
  end
end

// TX State Machine
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    tx_state     <= IDLE;
    tx_clk_count <= 0;
    tx_bit_count <= 0;
    tx_busy      <= 1'b0;
    tx           <= 1'b1;
    tx_shift_reg <= 8'h00;
  end else begin
    case (tx_state)
      IDLE: begin
        tx           <= 1'b1;
        tx_clk_count <= 0;
        tx_bit_count <= 0;
        if (tx_start && !tx_busy) begin
          tx_busy      <= 1'b1;
          tx_shift_reg <= tx_data;
          tx_state     <= START;
        end
      end
      START: begin
        tx <= 1'b0;
        if (tx_clk_count < CLKS_PER_BIT-1) tx_clk_count <= tx_clk_count + 1;
        else begin
          tx_clk_count <= 0;
          tx_state     <= DATA;
        end
      end
      DATA: begin
        tx <= tx_shift_reg[0];
        if (tx_clk_count < CLKS_PER_BIT-1) tx_clk_count <= tx_clk_count + 1;
        else begin
          tx_clk_count  <= 0;
          tx_shift_reg  <= {1'b0, tx_shift_reg[7:1]};
          if (tx_bit_count < 7) tx_bit_count <= tx_bit_count + 1;
          else begin
            tx_bit_count <= 0;
            tx_state     <= STOP;
          end
        end
      end
      STOP: begin
        tx <= 1'b1;
        if (tx_clk_count < CLKS_PER_BIT-1) tx_clk_count <= tx_clk_count + 1;
        else begin
          tx_clk_count <= 0;
          tx_busy      <= 1'b0;
          tx_state     <= IDLE;
        end
      end
    endcase
  end
end

// RX State Machine
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    rx_state      <= IDLE;
    rx_clk_count  <= 0;
    rx_bit_count  <= 0;
    rx_done       <= 1'b0;
    rx_data       <= 8'h00;
    rx_shift_reg  <= 8'h00;
  end else begin
    if (rx_done) rx_done <= 1'b0;
    case (rx_state)
      IDLE: begin
        rx_clk_count <= 0;
```

```
          rx_bit_count <= 0;
          if (rx_d2 == 1'b0) rx_state <= START;
        end
        START: begin
          if (rx_clk_count == (CLKS_PER_BIT-1)/2) begin
            if (rx_d2 == 1'b0) rx_clk_count <= rx_clk_count + 1;
            else rx_state <= IDLE;
          end else if (rx_clk_count < CLKS_PER_BIT-1) rx_clk_count <= rx_clk_count + 1;
          else begin
            rx_clk_count <= 0;
            rx_state     <= DATA;
          end
        end
        DATA: begin
          if (rx_clk_count == (CLKS_PER_BIT-1)/2) begin
            rx_shift_reg <= {rx_d2, rx_shift_reg[7:1]};
            rx_clk_count <= rx_clk_count + 1;
          end else if (rx_clk_count < CLKS_PER_BIT-1) rx_clk_count <= rx_clk_count + 1;
          else begin
            rx_clk_count <= 0;
            if (rx_bit_count < 7) rx_bit_count <= rx_bit_count + 1;
            else begin
              rx_bit_count <= 0;
              rx_state     <= STOP;
            end
          end
        end
        STOP: begin
          if (rx_clk_count < CLKS_PER_BIT-1) rx_clk_count <= rx_clk_count + 1;
          else begin
            rx_done  <= 1'b1;
            rx_data  <= rx_shift_reg;
            rx_state <= IDLE;
          end
        end
      endcase
    end
  end
endmodule
```

Listing 1: UART Transceiver Snippet

3 Testbench Development

To verify the UART implementation, a Verilog testbench (`uart_tb`) was written to simulate end-to-end loopback transmission and reception of ten predefined byte patterns.

3.1 Testbench Configuration

The testbench instantiates the DUT with:

- `.CLK_FREQ = 50_000_000` and `.BAUD_RATE = 115200`.
- `tx_out` connected to `rx` via a loopback wire.
- A 50 MHz clock generated by toggling `clk` every 10 ns.
- `rst_n` pulsed at time zero to reset the DUT.
- An array of ten test vectors (`test_data[0:9]`).

During simulation, each byte is sent when `tx_busy` is low and `tx_start` is pulsed; reception is signaled by `rx_done`, and `rx_data` is compared to `tx_data`. Any mismatch reports a failure.

```
`timescale 1ns/1ps
module uart_tb();

  // Parameters
  parameter CLK_PERIOD = 20;           // 50 MHz clock
  parameter BAUD_RATE  = 115200;
  parameter BIT_PERIOD = 1000000000 / BAUD_RATE;

  // Clock and Reset
  reg clk   = 0;
  reg rst_n = 0;

  // Transmitter Signals
  reg         tx_start = 0;
  reg [7:0]   tx_data  = 8'd0;
  wire        tx_busy;
  wire        tx_out;

  // Receiver Signals
  wire        rx_done;
  wire [7:0] rx_data;

  // Loopback connection
  wire rx_in = tx_out;

  // Test Data
  reg [7:0] test_data [0:9];
  integer   i;

  // Device Under Test
  uart_transceiver #(
    .CLK_FREQ (50_000_000),
    .BAUD_RATE(BAUD_RATE)
  ) dut (
```

```verilog
    .clk      (clk),
    .rst_n    (rst_n),
    .tx_start (tx_start),
    .tx_data  (tx_data),
    .tx_busy  (tx_busy),
    .tx       (tx_out),
    .rx       (rx_in),
    .rx_done  (rx_done),
    .rx_data  (rx_data)
  );

  // Clock Generator
  always #(CLK_PERIOD/2) clk = ~clk;

  // Waveform Dump
  initial begin
    $dumpfile("uart_transceiver_tb.vcd");
    $dumpvars(0, uart_tb);
  end

  // Monitor Received Data
  always @(posedge rx_done) begin
    $display("Time %0t: Received 0x%h", $time, rx_data);
    if (rx_data == tx_data)
      $display("Status: PASS");
    else
      $display("Status: FAIL - Expected: 0x%h, Got: 0x%h", tx_data, rx_data);
  end

  // Test Procedure
  initial begin
    // Initialize test data
    test_data[0] = 8'h55; test_data[1] = 8'hAA;
    test_data[2] = 8'h00; test_data[3] = 8'hFF;
    test_data[4] = 8'h01; test_data[5] = 8'h80;
    test_data[6] = 8'h33; test_data[7] = 8'hCC;
    test_data[8] = 8'hA5; test_data[9] = 8'h5A;

    // Apply reset
    #100; rst_n = 1; #100;

    // Loop through test vectors
    for (i = 0; i < 10; i = i + 1) begin
      wait (!tx_busy);
      @(posedge clk);

      tx_data  = test_data[i];
      tx_start = 1;
      $display("\nTime %0t: Sending 0x%h", $time, tx_data);

```

```
   @(posedge clk);
   tx_start = 0;

   wait (rx_done);
   #100;
  end

  // Done
  #5000;
  $display("\nUART Transceiver Test Complete");
  $finish;
 end

endmodule
```

Listing 2: Testbench Snippet

4 ModelSim Simulation

The UART transceiver was validated in ModelSim using a 50 MHz clock and a 115 200 baud loopback test. Figure 2 shows the captured waveform for a sequence of transmitted bytes. Each 10-bit frame (1 start bit, 8 data bits LSB-first, 1 stop bit) is sent and immediately received, confirming correct timing, bit ordering, and data integrity across all test vectors.

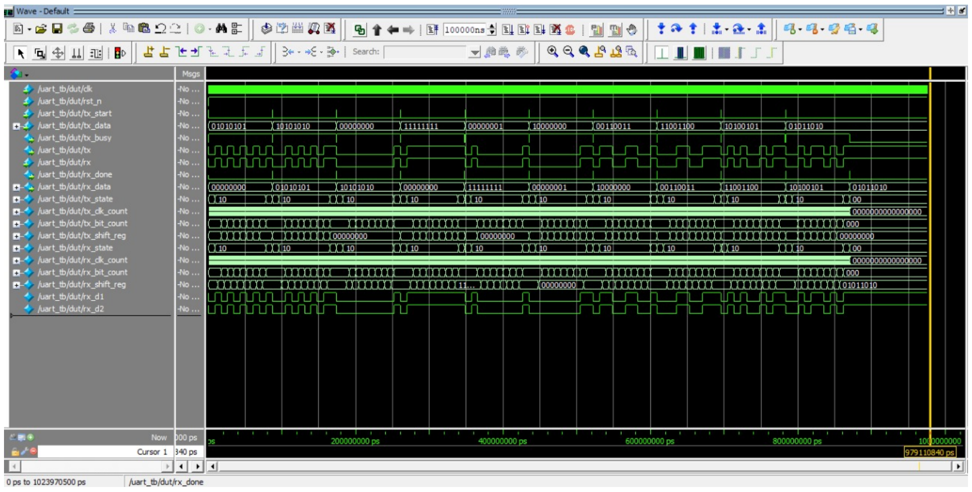

Figure 2: ModelSim waveform of UART loopback at 115 200 baud

Thank you!