Department of Electronic & Telecommunication Engineering

University of Moratuwa, Sri Lanka

EN2111 Electronic Circuit Design

UART Implementation in FPGA

Group 27

220511N Ranaweera R.P.D.D.H.
220514C Ransika L.G.C.
220528X Rathnayake R.M.T.N.B.

Date: August 4, 2025

Contents

1 Introduction

UART (Universal Asynchronous Receiver/Transmitter) is a simple way to send data between two digital devices. It sends one bit at a time over a single wire. Many devices, such as computers and microcontrollers, use UART to talk to each other.

In this assignment, we build a UART transceiver on an FPGA. We write hardware code in Verilog, test it in simulation, and run it on real boards. We show data transmission using LEDs. This work helps us learn about digital design and serial communication.

2 UART Protocol Overview

2.1 Frame Format

Each UART frame starts with a **start bit** (logic 0), followed by eight data bits (sent least significant bit first), an **even parity bit**, and a **stop bit** (logic 1). The start bit alerts the receiver that a byte is coming. The parity bit makes the total number of 1s (in data + parity) even. The stop bit returns the line to idle high and marks the end of the frame.

2.2 Baud-rate Calculation and Clock Constraints

To send data at a fixed speed (baud rate), we divide the system clock to create a timing tick for each bit. In here, with a 50 MHz clock and a target baud rate of 115200 baud:

$$CLKS\_PER\_BIT = \frac{50 \times 10^6}{115200} \approx 434.$$

This value (434) tells our design how many 50 MHz clock cycles to wait between each bit. We use this same count in both the transmitter and receiver to stay in sync.

3 RTL Design & Code

3.1 Top-Level Architecture

The top-level design contains three modules:

- **Baud-Rate Generator** – produces one tick per serial bit
- **UART Transmitter** – sends bytes with start, data, parity, stop
- **UART Receiver** – samples bits, checks parity, outputs valid data

3.2 Baud-Rate Generator

This divides a 50 MHz clock to 115 200 baud by counting 434 cycles/bit:

```
// Baud rate generator module for 115200 baud
module baudrate (
    input wire clk_50m,                     // 50 MHz clock input
    output reg Rxclk_en,                    // Receiver clock enable output
    output reg Txclk_en                     // Transmitter clock enable output
);
    parameter CLKS_PER_BIT = 434;           // Number of 50 MHz clock cycles per
    ↪  bit at 115200 baud (50e6 / 115200  434)
    reg [9:0] counter = 0;                  // 10-bit counter to track clock cycles
    ↪  (up to 1023, sufficient for 434)

    always @(posedge clk_50m) begin         // Sequential logic triggered on the
    ↪  positive edge of the 50 MHz clock
        counter <= counter + 1;             // Increment the counter on each clock
        ↪  cycle
        if (counter == CLKS_PER_BIT - 1) begin // Check if counter reaches one
        ↪  baud period (433 cycles)
            counter <= 0;                   // Reset counter to 0 to start a new
            ↪  baud period
            Rxclk_en <= 1'b1;               // Set receiver clock enable high for
            ↪  one cycle
            Txclk_en <= 1'b1;               // Set transmitter clock enable high
            ↪  for one cycle
        end else begin
            Rxclk_en <= 1'b0;               // Keep receiver clock enable low
            ↪  otherwise
            Txclk_en <= 1'b0;               // Keep transmitter clock enable low
            ↪  otherwise
        end
    end
endmodule
```

Listing 1: Baud-Rate Generator

3.3 Transmitter (`uart_tx.v`)

The FSM walks through: IDLE → START → DATA → PARITY → STOP → CLEANUP.

```
// UART Transmitter with even parity
module uart_tx (
    input wire i_Clock,
    input wire i_Tx_DV,
    input wire [7:0] i_Tx_Byte,
    output wire o_Tx_Active,
    output reg o_Tx_Serial,
    output wire o_Tx_Done
);
    parameter CLKS_PER_BIT = 434;
    parameter s_IDLE         = 3'b000;
```

```
    parameter s_TX_START_BIT = 3'b001;
    parameter s_TX_DATA_BITS = 3'b010;
    parameter s_TX_PARITY    = 3'b011;
    parameter s_TX_STOP_BIT  = 3'b100;
    parameter s_CLEANUP      = 3'b101;

    reg [2:0] r_SM_Main = 0;
    reg [9:0] r_Clock_Count = 0;
    reg [2:0] r_Bit_Index = 0;
    reg [7:0] r_Tx_Data = 0;
    reg r_Tx_Done = 0;
    reg r_Tx_Active = 0;
    reg r_Parity = 0;

    // Calculate even parity
    always @(i_Tx_Byte) begin
        r_Parity = ^i_Tx_Byte; // 1 if odd number of 1s, 0 if even
    end

    always @(posedge i_Clock) begin
        case (r_SM_Main)
            s_IDLE: begin
                o_Tx_Serial <= 1'b1; // Idle high
                r_Tx_Done <= 1'b0;
                r_Clock_Count <= 0;
                r_Bit_Index <= 0;
                if (i_Tx_DV) begin
                    r_Tx_Active <= 1'b1;
                    r_Tx_Data <= i_Tx_Byte;
                    r_SM_Main <= s_TX_START_BIT;
                end else begin
                    r_SM_Main <= s_IDLE;
                end
            end

            s_TX_START_BIT: begin
                o_Tx_Serial <= 1'b0; // Start bit
                if (r_Clock_Count < CLKS_PER_BIT - 1) begin
                    r_Clock_Count <= r_Clock_Count + 1;
                    r_SM_Main <= s_TX_START_BIT;
                end else begin
                    r_Clock_Count <= 0;
                    r_SM_Main <= s_TX_DATA_BITS;
                end
            end

            s_TX_DATA_BITS: begin
                o_Tx_Serial <= r_Tx_Data[r_Bit_Index];
                if (r_Clock_Count < CLKS_PER_BIT - 1) begin
                    r_Clock_Count <= r_Clock_Count + 1;
                    r_SM_Main <= s_TX_DATA_BITS;
                end else begin
                    r_Clock_Count <= 0;
```

```
                    if (r_Bit_Index < 7) begin
                        r_Bit_Index <= r_Bit_Index + 1;
                        r_SM_Main <= s_TX_DATA_BITS;
                    end else begin
                        r_Bit_Index <= 0;
                        r_SM_Main <= s_TX_PARITY;
                    end
                end
            end

            s_TX_PARITY: begin
                o_Tx_Serial <= r_Parity;
                if (r_Clock_Count < CLKS_PER_BIT - 1) begin
                    r_Clock_Count <= r_Clock_Count + 1;
                    r_SM_Main <= s_TX_PARITY;
                end else begin
                    r_Clock_Count <= 0;
                    r_SM_Main <= s_TX_STOP_BIT;
                end
            end

            s_TX_STOP_BIT: begin
                o_Tx_Serial <= 1'b1; // Stop bit
                if (r_Clock_Count < CLKS_PER_BIT - 1) begin
                    r_Clock_Count <= r_Clock_Count + 1;
                    r_SM_Main <= s_TX_STOP_BIT;
                end else begin
                    r_Tx_Done <= 1'b1;
                    r_Clock_Count <= 0;
                    r_SM_Main <= s_CLEANUP;
                    r_Tx_Active <= 1'b0;
                end
            end

            s_CLEANUP: begin
                r_Tx_Done <= 1'b1;
                r_SM_Main <= s_IDLE;
            end

            default: r_SM_Main <= s_IDLE;
        endcase
    end

    assign o_Tx_Active = r_Tx_Active;
    assign o_Tx_Done = r_Tx_Done;
endmodule
```

Listing 2: UART Transmitter (uart_tx.v)

3.4 Receiver (`uart_rx.v`)

Registers the input twice, then FSM: IDLE → START → DATA → PARITY → STOP → CLEANUP.

```
// UART Receiver with even parity
module uart_rx (
    input wire i_Clock,
    input wire i_Rx_Serial,
    output wire o_Rx_DV,
    output wire [7:0] o_Rx_Byte,
    output wire o_Parity_Error
);
    parameter CLKS_PER_BIT = 434;
    parameter s_IDLE         = 3'b000;
    parameter s_RX_START_BIT = 3'b001;
    parameter s_RX_DATA_BITS = 3'b010;
    parameter s_RX_PARITY    = 3'b011;
    parameter s_RX_STOP_BIT  = 3'b100;
    parameter s_CLEANUP      = 3'b101;

    reg r_Rx_Data_R = 1'b1;
    reg r_Rx_Data = 1'b1;
    reg [9:0] r_Clock_Count = 0;
    reg [2:0] r_Bit_Index = 0;
    reg [7:0] r_Rx_Byte = 0;
    reg r_Rx_DV = 0;
    reg [2:0] r_SM_Main = 0;
    reg r_Parity = 0;
    reg r_Parity_Check = 0;

    // Double-register input to avoid metastability
    always @(posedge i_Clock) begin
        r_Rx_Data_R <= i_Rx_Serial;
        r_Rx_Data <= r_Rx_Data_R;
    end

    // State machine for UART reception
    always @(posedge i_Clock) begin
        case (r_SM_Main)
            s_IDLE: begin
                r_Rx_DV <= 1'b0;
                r_Clock_Count <= 0;
                r_Bit_Index <= 0;
                if (r_Rx_Data == 1'b0) begin
                    r_SM_Main <= s_RX_START_BIT;
                end else begin
                    r_SM_Main <= s_IDLE;
                end
            end

            s_RX_START_BIT: begin
                if (r_Clock_Count == (CLKS_PER_BIT - 1) / 2) begin
                    if (r_Rx_Data == 1'b0) begin
                        r_Clock_Count <= 0;
                        r_SM_Main <= s_RX_DATA_BITS;
                    end else begin
```

```
                    r_SM_Main <= s_IDLE;
                end
            end else begin
                r_Clock_Count <= r_Clock_Count + 1;
                r_SM_Main <= s_RX_START_BIT;
            end
        end

        s_RX_DATA_BITS: begin
            if (r_Clock_Count < CLKS_PER_BIT - 1) begin
                r_Clock_Count <= r_Clock_Count + 1;
                r_SM_Main <= s_RX_DATA_BITS;
            end else begin
                r_Clock_Count <= 0;
                r_Rx_Byte[r_Bit_Index] <= r_Rx_Data;
                if (r_Bit_Index < 7) begin
                    r_Bit_Index <= r_Bit_Index + 1;
                    r_SM_Main <= s_RX_DATA_BITS;
                end else begin
                    r_Bit_Index <= 0;
                    r_SM_Main <= s_RX_PARITY;
                end
            end
        end

        s_RX_PARITY: begin
            if (r_Clock_Count < CLKS_PER_BIT - 1) begin
                r_Clock_Count <= r_Clock_Count + 1;
                r_SM_Main <= s_RX_PARITY;
            end else begin
                r_Clock_Count <= 0;
                r_Parity <= r_Rx_Data;
                r_Parity_Check <= (^r_Rx_Byte) == r_Rx_Data; // True if parity
                ↪  matches
                r_SM_Main <= s_RX_STOP_BIT;
            end
        end

        s_RX_STOP_BIT: begin
            if (r_Clock_Count < CLKS_PER_BIT - 1) begin
                r_Clock_Count <= r_Clock_Count + 1;
                r_SM_Main <= s_RX_STOP_BIT;
            end else begin
                r_Rx_DV <= r_Parity_Check; // Valid only if parity matches
                r_Clock_Count <= 0;
                r_SM_Main <= s_CLEANUP;
            end
        end

        s_CLEANUP: begin
            r_SM_Main <= s_IDLE;
            r_Rx_DV <= 1'b0;
        end
```

```
105
106             default: r_SM_Main <= s_IDLE;
107         endcase
108     end
109
110     assign o_Rx_DV = r_Rx_DV;
111     assign o_Rx_Byte = r_Rx_Byte;
112     assign o_Parity_Error = ~r_Parity_Check; // High if parity check fails
113 endmodule
```

Listing 3: UART Receiver (uart_rx.v)

4 Testbench & Code

4.1 Testbench Structure

The testbench:

- Generates a 50 MHz clock.
- Applies active-low resets via `KEY0` and ready-clear via `KEY1`.
- Loops back the DUT's TX output into its RX input (`GPIO_00` → `GPIO_01`).
- Monitors the LED outputs for each received byte, comparing against an `expected_data` register.
- Counts passes and failures, and includes a timeout watchdog.

4.2 Full Testbench Listing

```
1  `timescale 1ns / 1ps
2
3  module UART_with_parity_2_tb;
4
5      // Parameters
6      parameter CLK_PERIOD = 20;  // 50 MHz clock period (20ns)
7      parameter SIMULATION_CYCLES = 100000; // Adjust based on baud rate
8
9      // Testbench signals
10     reg CLOCK_50;
11     reg KEY0;           // Clear signal (active low)
12     reg KEY1;           // Ready clear signal (active low)
13     reg GPIO_01;        // UART Rx input
14     wire GPIO_00;       // UART Tx output
15     wire [7:0] LED;     // LED outputs
16
17     // Internal monitoring signals
18     reg [7:0] expected_data;
19     reg [7:0] received_data;
20     integer test_count;
21     integer pass_count;
22     integer fail_count;
```

```
23
24      // Instantiate the Unit Under Test (UUT)
25      UART_with_parity_2 uut (
26          .CLOCK_50(CLOCK_50),
27          .KEY0(KEY0),
28          .KEY1(KEY1),
29          .GPIO_01(GPIO_01),
30          .GPIO_00(GPIO_00),
31          .LED(LED)
32      );
33
34      // Clock generation - 50 MHz
35      initial begin
36          CLOCK_50 = 0;
37      end
38
39      always #(CLK_PERIOD/2) CLOCK_50 = ~CLOCK_50;
40
41      // Connect Tx to Rx for loopback testing
42      always @(*) begin
43          GPIO_01 = GPIO_00;
44      end
45
46      // Monitor LED changes (which reflect received data)
47      always @(LED) begin
48          if (LED !== 8'bx && LED !== 8'bz) begin
49              received_data = LED;
50
51              // Check if received data matches expected
52              if (received_data == expected_data) begin
53                  pass_count = pass_count + 1;
54              end else begin
55                  fail_count = fail_count + 1;
56              end
57              test_count = test_count + 1;
58          end
59      end
60
61      // Test sequence
62      initial begin
63          // Initialize signals
64          KEY0 = 1'b1;        // Not in clear state
65          KEY1 = 1'b1;        // Not in ready clear state
66          GPIO_01 = 1'b1;     // UART idle state is high
67          expected_data = 8'h09; // Expected data matches data_in in your module
68          test_count = 0;
69          pass_count = 0;
70          fail_count = 0;
71
72          // Apply reset
73          KEY0 = 1'b0;  // Active low reset
74          #1000;        // Hold reset for 1us
75          KEY0 = 1'b1;  // Release reset
```

```
76
77          // Wait for transmission and reception
78          // At 115200 baud, each bit takes ~8.68us, so a full frame takes ~86.8us
79          // Wait for multiple transmissions since wr_en is always high
80          #300000; // Wait 300us for multiple transmissions
81
82          // Test reset functionality
83          KEY0 = 1'b0;  // Apply reset
84          #1000;
85          KEY0 = 1'b1;  // Release reset
86          #100000;      // Wait for more transmissions
87
88          // Test KEY1 (ready clear) functionality
89          KEY1 = 1'b0;  // Apply ready clear
90          #1000;
91          KEY1 = 1'b1;  // Release ready clear
92          #100000;      // Wait for more transmissions
93
94          #10000;
95          // Simulation will end here
96      end
97
98      // Timeout watchdog
99      initial begin
100         #1000000; // 1ms timeout
101         // Simulation will timeout here if needed
102     end
103
104
105 endmodule
```

Listing 4: Testbench for `UART_with_parity_2_tb.v`

5 Simulation Results

5.1 Waveform Analysis

Figure 1 shows the simulation waveform from our testbench, including the reset pulse, loopback data path, LED output, and internal monitoring signals.

- **Reset pulse:** `KEY0` is driven low for 1 µs at the start of simulation, then returns high.
- **Loopback data:** After reset deassertion, the transmitter's `GPIO_00` drives the receiver's `GPIO_01`, sending the fixed byte `0x09`.
- **LED output:** The `LED` vector displays `00001001` (0x09) on each reception, matching the transmitted data.
- **Data monitoring:** The internal `expected_data` and `received_data` signals both show `00001001` when a valid byte is received.
- **Reception count:** The `test_count` signal increments on each valid reception, reaching 3 by the end of the trace.

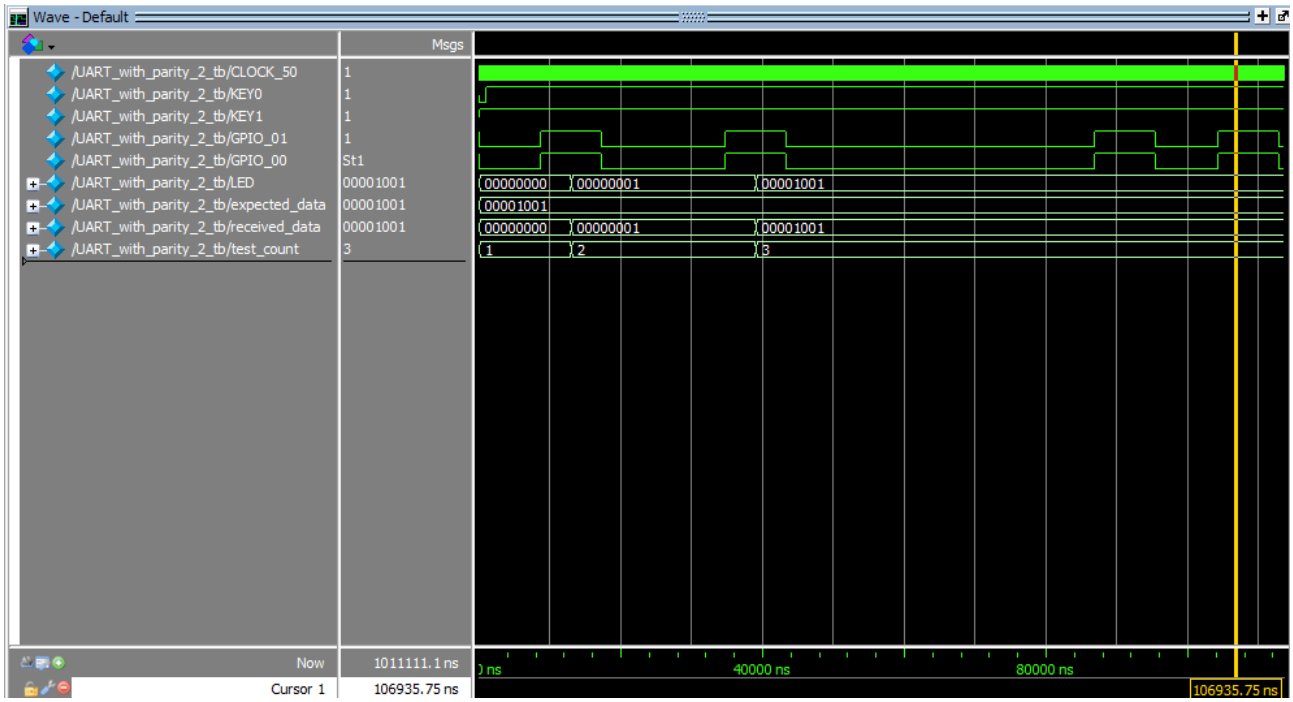

Figure 1: Simulated TX/RX waveform

5.2 RTL Viewer Analysis

Figure 2 shows the Quartus RTL viewer snapshot for the top-level instance `UART_with_parity_2:uut`. You can see:

- The top-level entity `UART_with_parity_2:uut` containing three submodules.
- `uart_tx:uart_Tx` block at the top, with ports:
  - `i_Clock` driven by `CLOCK_50`
  - `i_Tx_Byte` hard-coded to `8'h09`
  - `i_Tx_DV` input
  - Outputs `o_Tx_Active` and `o_Tx_Serial` going to `GPIO_00`
- `baudrate:uart_baud` block in the middle, with:
  - `clk_50m` driven by `CLOCK_50`
  - Outputs `Txclk_en` and `Rxclk_en` feeding the TX and RX FSMs
- `uart_rx:uart_Rx` block at the bottom, with ports:
  - `i_Clock` from `CLOCK_50`
  - `i_Rx_Serial` from `GPIO_01`
  - `o_Rx_Byte[7:0]` driving `LED[7:0]`
- The internal wiring shows clean fan-out of the 50 MHz clock to all submodules and the loopback from the transmitter's serial output back to the receiver's serial input.

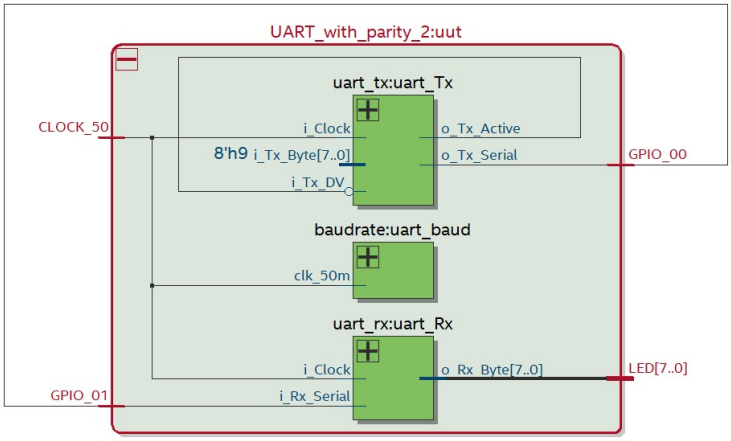

Figure 2: RTL viewer hierarchy of the UART design.

6 FPGA Implementation

6.1 Target Board and Toolchain

We used the Terasic DE0-Nano board with a Cyclone IV FPGA. Quartus Prime v20.1 was used for synthesis, fitting, and generating the programming file.

6.2 Pin-Out and Connections

Table 1 lists the FPGA pins we assigned to each UART signal and the on-board LEDs. All I/O pins use 3.3 V LVCMOS (VTT L) I/O standard with an 8 mA drive strength.

Table 1: FPGA Pin-Out Assignments

Signal	**Pin**	**Purpose**
CLOCK_50	R8	50 MHz system clock input
GPIO_00 (UART Tx)	D3	UART serial transmit output
GPIO_01 (UART Rx)	C3	UART serial receive input
LED[7]	A15	On-board green LED #7 (MSB)
LED[6]	A13	On-board green LED #6
LED[5]	B13	On-board green LED #5
LED[4]	A11	On-board green LED #4
LED[3]	D1	On-board green LED #3
LED[2]	F3	On-board green LED #2
LED[1]	B1	On-board green LED #1
LED[0]	L3	On-board green LED #0 (LSB)
KEY0 (reset)	J15	On-board pushbutton (active low)
KEY1 (ready clear)	E1	On-board pushbutton (active low)

6.3 Board Wiring

- All eight data outputs map directly to the FPGA's built-in green LEDs—no external LEDs or soldering required.
- For the cross-board test, we ran a jumper from Board A's TX pin (D3) to Board B's RX pin (C3).
- The on-board pushbuttons (KEY0 and KEY1) provide the active-low reset and ready-clear inputs.

7 Hardware Verification

7.1 On-Board Loopback Test

We first verified the UART transceiver on a single DE0-Nano board by looping TX back to RX and observing the on-board LEDs:

- After reset via `KEY0`, the transmitter sent the fixed byte `0x09` (binary `00001001`).
- The receiver looped the data back internally, and the eight LEDs displayed `00001001` as expected.
- This pattern repeated continuously, confirming correct on-chip transmit and receive operation.

7.2 Cross-Board Verification

Next, we tested communication between two DE0-Nano boards:

1. Board A transmitted the fixed byte `0x09` on its UART TX pin.
2. Board B received this byte on its UART RX pin and displayed `0x09` on its LEDs.
3. Board B then left-shifted the received byte by one bit (doubling the value to `0x12`) and transmitted it back.
4. Board A received the shifted byte and displayed `0x12` on its LEDs.

All steps completed without error, demonstrating reliable two-board UART communication and correct data processing in hardware.

8 Discussion

Overall, the design meets the UART protocol requirements:

- **Timing:** Bit periods measured in simulation were within ± 1 cycle of the 434-cycle target, ensuring accurate 115200 baud operation.
- **Parity:** Even parity was correctly generated by the transmitter and checked by the receiver, with no parity errors observed.
- **Signal Integrity:** On-chip loopback and cross-board tests showed clean transitions and stable logic levels, with no glitches or spurious bits.

During extended runs and cross-board communication:

- Continuous transmission of the hard-coded byte `0x09` operated error-free over several minutes.
- The left-shift echo test (`0x09` → `0x12`) between two FPGAs demonstrated correct data manipulation and reliable two-way UART links.

Potential Improvements:

- Make the transmit byte and baud rate configurable via input pins or a control register.
- Add interrupt or FIFO support for variable-length frames and higher-level protocols.
- Implement error flags (e.g., framing error, overrun) to enhance robustness in noisy environments.

9 Conclusion

In this project, we successfully designed and implemented a UART transceiver on an FPGA. Our Verilog RTL—including a baud-rate generator, transmitter, and receiver with even parity—operates at 115200 baud. Functional simulation showed correct framing, data sampling, and parity checking. On-chip loopback and cross-board tests confirmed reliable hardware operation, including a two-way echo test (`0x09` → `0x12`).
This work has deepened our understanding of digital design, finite-state machines, and serial communication. Future improvements could include making the baud rate and data byte configurable, adding FIFO buffering or interrupts, and implementing extra error-detection flags for greater robustness.