University of Moratuwa Faculty of

Engineering Department of Electronic and

Telecommunication Engineering EN2111 -

Electronic Circuit Design

Report - UART Implementation in FPGA

Team 16

Kuruppuarachchi K.A.R.R.	220350T
Kumarage R.V.	220343B
Kulasinghe H.P.G.N.A.	220334A

May 9, 2025

Contents

1 Introduction

1.1 Overview of UART Protocol

UART (Universal Asynchronous Receiver/Transmitter) is a widely used serial communication protocol in embedded systems. Unlike synchronous protocols (such as SPI and I2C), UART does not require a clock signal, making it simpler to implement but requiring both transmitter and receiver to agree on timing parameters.

1.2 UART Communication Basics

UART transmits data serially, one bit at a time, through two separate lines:

- TX (Transmit) - For sending data
- RX (Receive) - For receiving data

Each UART frame consists of:

- Start bit (always 0) - Signals the beginning of data transmission
- Data bits (typically 8 bits) - The actual information being transmitted
- Optional parity bit - For basic error detection (not used in this implementation)
- Stop bit(s) (always 1) - Signals the end of data transmission

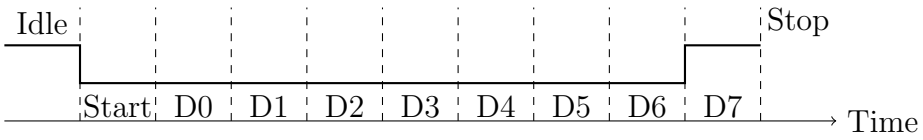


Figure 1: UART Frame Structure with 8 Data Bits

1.3 Baud Rate

Baud rate is the speed of data transmission measured in bits per second. Both transmitter and receiver must operate at the same baud rate (with minimal timing error) for successful communication. This project uses a baud rate of 115,200 bits per second, which is a standard rate for many applications.

2 Objective

The primary objectives of this project are:

1. To design and implement a complete UART transceiver on the DE0-Nano FPGA board
2. To verify the functionality through comprehensive simulation

3. To demonstrate practical hardware communication between two FPGA boards
4. To display received data on 7-segment displays and LEDs for visual verification
5. To gain practical experience with digital design and FPGA implementation of communication protocols

Additionally, this project aims to provide a reusable and well-documented UART implementation that can be incorporated into future FPGA designs for serial communication.

3 RTL Code for UART

3.1 UART Transmitter Module

```
`timescale 1ns / 1ps
module uart_tx (
input  wire clk,        // System clock
input  wire rst,        // Active high reset
input  wire start,      // Start transmission signal
input  wire [7:0] data, // Data byte to transmit
input  wire baud_tick, // Baud rate timing signal
output reg  tx          // Serial output line
);
reg [3:0] bit_index = 0;   // Track which bit is being transmitted (0 = idle)
reg [9:0] shift_reg = 10'b1111111111;  // Shift register with all bits set to 1 (idle)
always @(posedge clk or posedge rst) begin
    if (rst) begin
        tx <= 1'b1;        // Idle state of TX line is high
        bit_index <= 0;    // Reset to idle state
        shift_reg <= 10'b1111111111;  // Reset shift register to idle state
    end else begin
        // Start a new transmission only if idle (bit_index == 0)
        if (start && bit_index == 0) begin
            // Load shift register with frame: {stop bit, 8 data bits, start bit}
            shift_reg <= {1'b1, data, 1'b0};  // Stop bit, data, start bit
            bit_index <= 1;   // Start counting from 1 (transmitting)
        end else if (baud_tick && bit_index != 0) begin
            // Only shift and transmit on baud_tick and when actively transmitting
            tx <= shift_reg[0];                       // Send LSB to output
            shift_reg <= {1'b1, shift_reg[9:1]};      // Shift right, filling with '1'

            if (bit_index == 10)        // All 10 bits sent (1 start + 8 data + 1 stop)
                bit_index <= 0;         // Reset to idle state
            else
                bit_index <= bit_index + 1;  // Increment bit counter
        end
```

```
33      end
34  end
35  endmodule
```

Listing 1: UART Transmitter Implementation

3.2 UART Transmitter Description

The UART transmitter module handles the conversion of parallel data into a serial bit stream following the UART protocol. Here's a detailed explanation:

3.2.1 Module Parameters and Signals

- **clk**: System clock input
- **rst**: Active high reset signal
- **start**: Trigger signal to initiate a new transmission
- **data[7:0]**: 8-bit parallel data to be transmitted
- **baud_tick**: Timing signal that pulses at the baud rate
- **tx**: Serial output line where data is transmitted

3.2.2 Internal Registers

- **bit_index[3:0]**: Tracks the current bit position during transmission
  - 0: Idle state (no active transmission)
  - 1-10: Currently transmitting bits 1-10 of the frame
- **shift_reg[9:0]**: 10-bit shift register that holds the entire UART frame
  - Bit 0: Currently being transmitted (LSB first)
  - Initial value 10'b1111111111 represents the idle state

3.2.3 Operation Flow

1. **Reset Condition**: When reset is asserted, tx line is set high (idle), bitindex is reset to 0, and shift_reg is filled with 1's.
2. **Start Transmission**: When the start signal is asserted and transmitter is idle (bit_index = 0):
   - The shift register is loaded with the complete frame: {stop bit (1), 8 data bits, start bit (0)}
   - bit_index is set to 1 to begin transmission
3. **Bit Transmission**: At each baud_tick (when actively transmitting):
   - The LSB of shift_reg is output on the tx line

- The shift register is shifted right, inserting a '1' at the MSB
- bit_index is incremented

4. **End of Transmission**: When bit_index reaches 10:
   - All bits have been transmitted
   - bit_index is reset to 0, returning to idle state

3.3 UART Receiver Module

```
`timescale 1ns / 1ps
module uart_rx (
input  wire clk,         // System clock
input  wire rst,         // Active high reset
input  wire rx,          // Serial input line
input  wire baud_tick,   // Baud rate timing signal
output reg [7:0] data    // Received parallel data output
);
reg [3:0] bit_index = 0;    // Bit counter for reception
reg [7:0] shift_reg = 0;    // Shift register for received bits
reg [1:0] state = 0;        // FSM state
reg rx_sync = 1;            // Synchronized rx input

// FSM state definitions
localparam IDLE  = 0,     // Waiting for start bit
           START = 1,     // Verifying start bit
           DATA  = 2,     // Receiving data bits
           STOP  = 3;     // Receiving stop bit

always @(posedge clk or posedge rst) begin
    if (rst) begin
        state <= IDLE;      // Reset to idle state
        bit_index <= 0;     // Reset bit counter
        shift_reg <= 0;     // Clear shift register
        data <= 0;          // Clear output data
        rx_sync <= 1;       // Reset synchronized input
    end else begin
        rx_sync <= rx;      // Synchronize rx input with system clock

        case (state)
            IDLE: begin
                if (!rx_sync)  // Start bit detected (falling edge)
                    state <= START;
            end

            START: begin
                if (baud_tick) begin
                    // Sample in the middle of the presumed start bit
                    if (!rx_sync) begin
                        state <= DATA;    // Confirmed start bit, move
   to data state
                        bit_index <= 0;   // Reset bit counter for data
    bits
                    end else begin
                        state <= IDLE;    // False start bit, return to
    idle
```

```
                    end
                end
            end

            DATA: begin
                if (baud_tick) begin
                    // Sample in the middle of each data bit
                    shift_reg[bit_index] <= rx_sync;  // Store bit in
    shift register
                    if (bit_index == 7)             // All 8 data bits
    received
                        state <= STOP;              // Move to stop bit
     state
                    bit_index <= bit_index + 1;     // Increment bit
    counter
                end
            end

            STOP: begin
                if (baud_tick) begin
                    // Sample in the middle of the stop bit
                    if (rx_sync) begin               // Valid stop bit
    (high)
                        data <= shift_reg;           // Update output
    data
                    end
                    state <= IDLE;                   // Return to idle
    state
                end
            end
        endcase
    end
end
endmodule
```

Listing 2: UART Receiver Implementation

3.4 UART Receiver Description

The UART receiver module converts the serial bit stream back into parallel data. It employs a finite state machine (FSM) approach to detect and process incoming UART frames.

3.4.1 Module Parameters and Signals

- **clk**: System clock input
- **rst**: Active high reset signal
- **rx**: Serial input line where data is received
- **baud_tick**: Timing signal that pulses at the baud rate
- **data[7:0]**: 8-bit parallel data output

3.4.2 Internal Registers

- **bit_index[3:0]**: Tracks the current bit position during reception
- **shift_reg[7:0]**: 8-bit shift register that accumulates received data bits
- **state[1:0]**: Current state of the receiver FSM
- **rx_sync**: Synchronized version of the rx input (prevents metastability)

3.4.3 Finite State Machine States

- **IDLE (0)**: Waiting for a start bit (falling edge on rx line)
- **START (1)**: Verifying the start bit by sampling in its middle
- **DATA (2)**: Receiving and storing the 8 data bits
- **STOP (3)**: Verifying the stop bit and updating the output data

3.4.4 Operation Flow

1. **Reset Condition**: All registers are initialized to their default values.
2. **IDLE State**: The receiver waits for a falling edge on the rx line, indicating a potential start bit.
3. **START State**: After detecting a falling edge, the receiver waits for one baud interval to sample in the middle of the start bit:
    - If the sampled bit is low (0), it confirms a valid start bit and moves to DATA state
    - If the sampled bit is high (1), it was a false trigger and returns to IDLE state
4. **DATA State**: The receiver samples each data bit at the baud rate:
    - Each sampled bit is stored in the shift register at the appropriate position
    - After receiving all 8 data bits, it transitions to STOP state
5. **STOP State**: The receiver samples the stop bit:
    - If the stop bit is valid (high), the received data is transferred to the output
    - Regardless of stop bit validity, the receiver returns to IDLE state to await the next frame

3.5 UART Transceiver

```
`timescale 1ns / 1ps
module invert_uart_transceiver_test #(
parameter CLK_FREQ  = 50000000,     // System clock frequency in Hz
parameter BAUD_RATE = 115200        // UART baud rate in bits/second
)(
input  wire        clk,             // System clock
input  wire        rst_n,           // Active LOW reset (KEY0)
input  wire        key1_n,          // Active LOW transmit trigger (KEY1)
output wire        txd,             // UART transmit line
input  wire        rxd,             // UART receive line
output wire [7:0]  rx_data,         // Received data output
output wire [7:0]  leds,            // LED display of received data
output wire [6:0]  seg              // 7-segment display output
);
wire rst = ~rst_n;                  // Convert active-low to active-high reset
wire key1 = ~key1_n;                // Convert active-low to active-high key

wire baud_tick;                     // Baud rate timing signal
reg [15:0] baud_cnt = 0;            // Baud rate counter
reg tx_start;                       // Transmission trigger
reg [7:0] tx_data = 8'h05;          // Data to transmit (0x05 = 5)

// Key debounce and edge detection
reg [2:0] key1_sync;                // Synchronizer registers
wire key1_pressed;                  // Edge detection output

// Synchronize key input to prevent metastability
always @(posedge clk) begin
    key1_sync <= {key1_sync[1:0], key1};
end

// Detect falling edge (button press)
assign key1_pressed = (key1_sync[2:1] == 2'b10);

// Baud rate generator
// Divides the system clock to generate timing for UART operations
always @(posedge clk or posedge rst) begin
    if (rst)
        baud_cnt <= 0;              // Reset counter
    else if (baud_cnt == (CLK_FREQ / BAUD_RATE - 1))
        baud_cnt <= 0;              // Reset at terminal count
    else
        baud_cnt <= baud_cnt + 1;   // Increment counter
end

// Generate tick at baud rate
assign baud_tick = (baud_cnt == 0);

// Trigger transmission on key press (falling edge)
always @(posedge clk or posedge rst) begin
    if (rst)
        tx_start <= 0;              // Reset transmission trigger
    else
```

```
        tx_start <= key1_pressed;   // Set trigger on key press
end

// Instantiate UART transmitter
uart_tx transmitter (
    .clk(clk),
    .rst(rst),
    .start(tx_start),
    .data(tx_data),
    .baud_tick(baud_tick),
    .tx(txd)
);

// Instantiate UART receiver
uart_rx receiver (
    .clk(clk),
    .rst(rst),
    .rx(rxd),
    .baud_tick(baud_tick),
    .data(rx_data)
);

// Connect received data to LEDs for visual feedback
assign leds = rx_data;

// Instantiate 7-segment decoder to display lower 4 bits of received
   data
binary_to_7seg seg_decoder (
    .data_in(rx_data[3:0]),
    .data_out(seg)
);
endmodule
```

Listing 3: UART Transceiver Integration

3.6 UART Transceiver Description

The UART transceiver module integrates the transmitter and receiver modules into a complete communication system with the necessary support circuitry.

3.6.1 Parameterization

The module is parameterized to allow for different clock frequencies and baud rates:

- **CLK_FREQ**: System clock frequency in Hz (default: 50 MHz)
- **BAUD_RATE**: UART baud rate in bits per second (default: 115,200 bps)

3.6.2 Key Components

1. **Baud Rate Generator**:
    - Divides the system clock to generate precise timing for UART operations
    - Uses a counter that cycles at the baud rate: CLK_FREQ / BAUD_RATE
    - Generates a single-cycle pulse (baud_tick) at the baud rate

2. **Key Input Processing**:
   - Synchronizes the active-low key input to the system clock domain
   - Detects the falling edge of the key input (button press)
   - Generates a single-cycle tx_start pulse to trigger data transmission
3. **UART Transmitter Integration**:
   - Connects to the top-level txd output
   - Triggered by key1 button press
   - Transmits a predefined data value (0x05)
4. **UART Receiver Integration**:
   - Connects to the top-level rxd input
   - Outputs received data to the rx_data port
5. **Output Display**:
   - Routes received data to LEDs for visual monitoring
   - Connects the lower 4 bits to a 7-segment display via decoder

3.7 Binary to 7-Segment LED Display

```
module binary_to_7seg (
input [3:0] data_in,      // 4-bit binary input
output [6:0] data_out     // 7-segment display output
);
reg [6:0] lut_7seg [0:15];  // Look-up table for segment patterns
reg [6:0] seg_val;          // Selected segment pattern

// Assign output from internal register
assign data_out = seg_val;

always @(*) begin
    // Look-up table initialization for decimal digits 0-9
    // Segment order: abcdefg (1 = segment ON)
    lut_7seg[0]  = 7'b0111111;  // Digit 0
    lut_7seg[1]  = 7'b0000110;  // Digit 1
    lut_7seg[2]  = 7'b1011011;  // Digit 2
    lut_7seg[3]  = 7'b1001111;  // Digit 3
    lut_7seg[4]  = 7'b1100110;  // Digit 4
    lut_7seg[5]  = 7'b1101101;  // Digit 5
    lut_7seg[6]  = 7'b1111101;  // Digit 6
    lut_7seg[7]  = 7'b0000111;  // Digit 7
    lut_7seg[8]  = 7'b1111111;  // Digit 8
    lut_7seg[9]  = 7'b1101111;  // Digit 9

    // Blank display for values 10-15 (A-F not implemented)
    lut_7seg[10] = 7'b0000000;
    lut_7seg[11] = 7'b0000000;
    lut_7seg[12] = 7'b0000000;
    lut_7seg[13] = 7'b0000000;
```

```
    lut_7seg[14] = 7'b0000000;
    lut_7seg[15] = 7'b0000000;

    // Select pattern based on input value
    seg_val = lut_7seg[data_in];
end
endmodule
```

Listing 4: 7-Segment Display Decoder

3.8 7-Segment Display Decoder Description

The binary to 7-segment decoder converts a 4-bit binary value into the corresponding pattern for a 7-segment display.

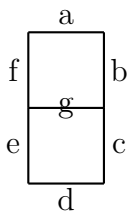


Figure 2: 7-Segment Display Segment Labeling

3.8.1 Implementation Details

- Uses a look-up table approach for efficient implementation
- Each 7-bit pattern represents the state of segments a through g (1 = segment ON)
- Patterns are defined for decimal digits 0-9
- Values 10-15 (A-F) are not implemented and display blank
- The module is purely combinational with no clock or reset inputs

4 Testbench

```
`timescale 1ns / 1ps
module tb_invert_uart_transceiver_test();
// Parameters
parameter CLK_FREQ  = 50000000;     // 50 MHz system clock
parameter BAUD_RATE = 115200;       // 115,200 bps baud rate
parameter BAUD_TICK_PERIOD = 1_000_000_000 / BAUD_RATE; // Period in ns

// Testbench signals
reg clk = 0;                        // System clock
reg rst_n = 0;                      // Active-low reset
reg key1_n = 1;                     // Active-low transmit trigger
wire txd;                           // UART transmit line
wire rxd;                           // UART receive line
wire [7:0] rx_data;                 // Received data
```

```
wire [7:0] leds;                    // LED outputs

// Clock generation - 50 MHz (20 ns period)
always #10 clk = ~clk;

// Instantiate Device Under Test (DUT)
invert_uart_transceiver_test #(
    .CLK_FREQ(CLK_FREQ),
    .BAUD_RATE(BAUD_RATE)
) dut (
    .clk(clk),
    .rst_n(rst_n),
    .key1_n(key1_n),
    .txd(txd),
    .rxd(rxd),          // Loopback connection
    .rx_data(rx_data),
    .leds(leds)
);

// Create loopback by connecting txd to rxd
assign rxd = txd;

// Test sequence
initial begin
    // Initialize and apply reset
    rst_n = 0;          // Assert reset
    key1_n = 1;         // Release key
    #100;               // Wait 100 ns
    rst_n = 1;          // Deassert reset
    #1000;              // Wait 1000 ns for stabilization

    // Press KEY1 to start transmission
    key1_n = 0;         // Press button (active low)
    #40;                // Hold for 40 ns
    key1_n = 1;         // Release button

    // Wait for full transmission and reception cycle
    // ~12 bit times at baud rate (including start and stop bits)
    #(BAUD_TICK_PERIOD * 12 * 1000);

    // Display results
    $display("Received data: %h", rx_data);
    $display("LEDs: %b", leds);

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

Listing 5: UART Transceiver Testbench

4.1 Testbench Description

The testbench provides a controlled environment to verify the functionality of the UART transceiver implementation. It simulates both the transmitter and receiver operations in a loopback configuration.

4.1.1 Testbench Structure

- **Parameter Definition**: Sets the clock frequency and baud rate, and calculates the baud period in nanoseconds for timing calculations
- **Signal Declaration**: Defines all necessary signals to interface with the DUT
- **Clock Generation**: Creates a 50 MHz clock signal with 20 ns period
- **DUT Instantiation**: Creates an instance of the UART transceiver with proper parameter passing
- **Loopback Connection**: Connects the txd output directly to the rxd input to enable self-testing

4.1.2 Test Sequence

1. **Initialization**: Applies a 100 ns reset pulse to initialize all internal registers
2. **Stabilization**: Waits an additional 1000 ns for the system to stabilize
3. **Transmission Trigger**: Simulates a button press by asserting key1_n for 40 ns
4. **Observation Period**: Waits long enough for a complete UART frame transmission and reception
5. **Result Reporting**: Displays the received data in both hexadecimal and binary formats

The testbench uses a loopback configuration where the transmitter output is directly connected to the receiver input. This allows testing the entire communication path without requiring a second UART device. The test verifies that data sent by the transmitter (0x05) is correctly received by the receiver and displayed on the outputs.

5 Simulation Results

The simulation waveform shows the complete UART transmission and reception process in the loopback configuration.

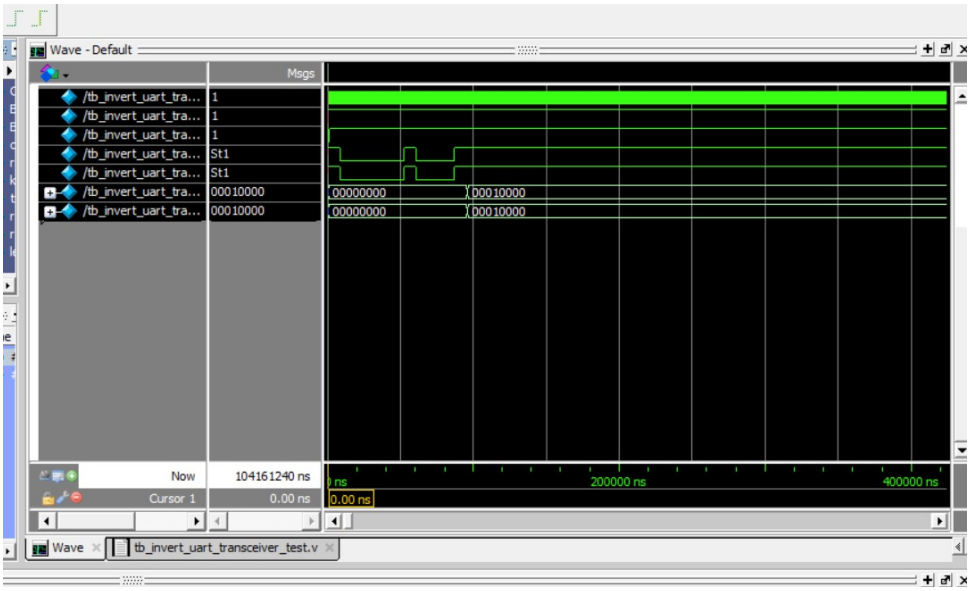


Figure 3: UART simulation waveform

5.1 Waveform Analysis

The key signals in the simulation waveform can be interpreted as follows:

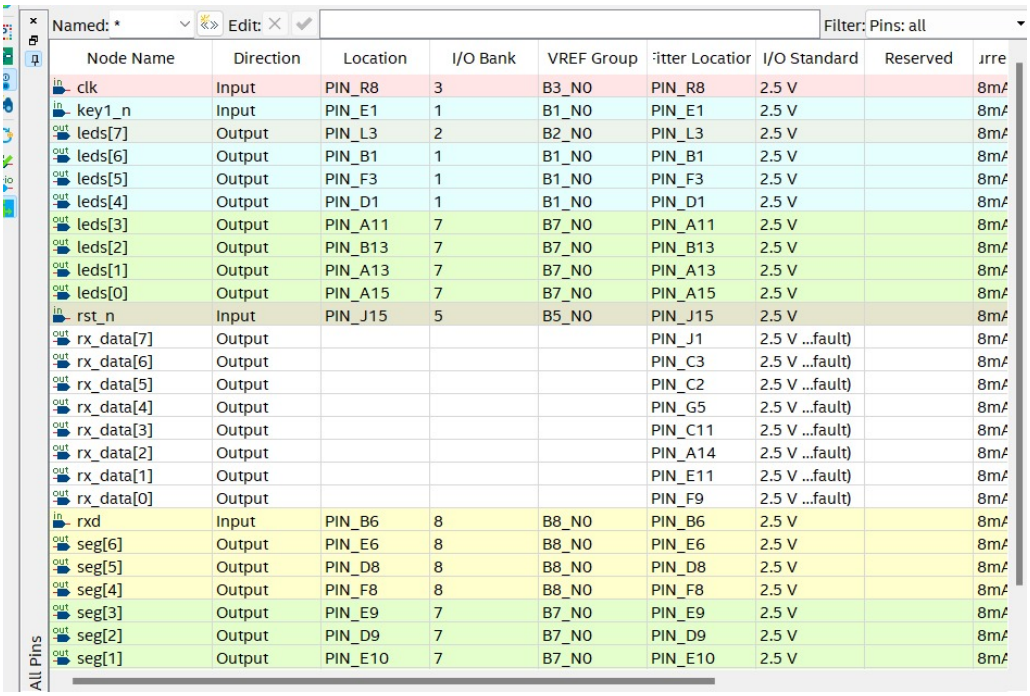
1. **txd Signal**: The transmitter output shows the complete UART frame:
   - Idle state (high) before transmission
   - Start bit (low) to initiate the frame
   - 8 data bits (in this case, representing 0x05 = 00000101, sent LSB first)
   - Stop bit (high) to complete the frame
   - Return to idle state (high)
2. **rxd Signal**: Mirrors the txd signal due to the loopback connection
3. **rx_data Signal**: Shows the received data value (0x05) after the complete frame has been processed
4. **key1_n and tx_start Signals**: Show the button press and resulting transmission trigger
5. **baud_tick Signal**: Shows the periodic timing pulses used to clock the UART operations

The simulation confirms that the data is correctly transmitted and received, verifying the proper operation of both the transmitter and receiver modules and their integration in the transceiver.

6 FPGA Implementation

6.1 Pin Assignments

The following pin assignments were used to connect the UART transceiver to the physical pins on the DE0-Nano FPGA board:



Named: * Edit: Filter: Pins: all

Node Name	Direction	Location	I/O Bank	VREF Group	itter Locatior	I/O Standard	Reserved	ırre
clk	Input	PIN_R8	3	B3_N0	PIN_R8	2.5 V		8mA
key1_n	Input	PIN_E1	1	B1_N0	PIN_E1	2.5 V		8mA
leds[7]	Output	PIN_L3	2	B2_N0	PIN_L3	2.5 V		8mA
leds[6]	Output	PIN_B1	1	B1_N0	PIN_B1	2.5 V		8mA
leds[5]	Output	PIN_F3	1	B1_N0	PIN_F3	2.5 V		8mA
leds[4]	Output	PIN_D1	1	B1_N0	PIN_D1	2.5 V		8mA
leds[3]	Output	PIN_A11	7	B7_N0	PIN_A11	2.5 V		8mA
leds[2]	Output	PIN_B13	7	B7_N0	PIN_B13	2.5 V		8mA
leds[1]	Output	PIN_A13	7	B7_N0	PIN_A13	2.5 V		8mA
leds[0]	Output	PIN_A15	7	B7_N0	PIN_A15	2.5 V		8mA
rst_n	Input	PIN_J15	5	B5_N0	PIN_J15	2.5 V		8mA
rx_data[7]	Output				PIN_J1	2.5 V ...fault)		8mA
rx_data[6]	Output				PIN_C3	2.5 V ...fault)		8mA
rx_data[5]	Output				PIN_C2	2.5 V ...fault)		8mA
rx_data[4]	Output				PIN_G5	2.5 V ...fault)		8mA
rx_data[3]	Output				PIN_C11	2.5 V ...fault)		8mA
rx_data[2]	Output				PIN_A14	2.5 V ...fault)		8mA
rx_data[1]	Output				PIN_E11	2.5 V ...fault)		8mA
rx_data[0]	Output				PIN_F9	2.5 V ...fault)		8mA
rxd	Input	PIN_B6	8	B8_N0	PIN_B6	2.5 V		8mA
seg[6]	Output	PIN_E6	8	B8_N0	PIN_E6	2.5 V		8mA
seg[5]	Output	PIN_D8	8	B8_N0	PIN_D8	2.5 V		8mA
seg[4]	Output	PIN_F8	8	B8_N0	PIN_F8	2.5 V		8mA
seg[3]	Output	PIN_E9	7	B7_N0	PIN_E9	2.5 V		8mA
seg[2]	Output	PIN_D9	7	B7_N0	PIN_D9	2.5 V		8mA
seg[1]	Output	PIN_E10	7	B7_N0	PIN_E10	2.5 V		8mA

All Pins

Figure 4: Pin Planner Configuration

6.2 Pin Assignment Details

6.3 Hardware Setup

For the hardware implementation, two DE0-Nano FPGA boards were connected as follows:

1. **Board-to-Board Connection**:
   - TXD pin of Board 1 connected to RXD pin of Board 2
   - TXD pin of Board 2 connected to RXD pin of Board 1
   - GND pins of both boards connected together
2. **Power Supply**: Both boards powered via USB connection
3. **Control Interface**: Push buttons (KEY0 and KEY1) used for reset and transmission trigger
4. **Output Display**: On-board LEDs and external 7-segment display used to visualize received data

Signal Name	FPGA Pin	Description
clk	PIN_R8	50 MHz system clock
rst_n	PIN_J15	KEY0 (Reset button)
key1_n	PIN_E1	KEY1 (Transmit trigger)
txd	PIN_A3	UART transmit line (GPIO_0)
rxd	PIN_B3	UART receive line (GPIO_0)
leds[0]	PIN_A15	LED0 (LSB of received data)
leds[1]	PIN_A13	LED1
leds[2]	PIN_B13	LED2
leds[3]	PIN_A11	LED3
leds[4]	PIN_D1	LED4
leds[5]	PIN_F3	LED5
leds[6]	PIN_B1	LED6
leds[7]	PIN_L3	LED7 (MSB of received data)
seg[0]	PIN_C14	7-segment display segment a
seg[1]	PIN_E15	7-segment display segment b
seg[2]	PIN_C15	7-segment display segment c
seg[3]	PIN_C16	7-segment display segment d
seg[4]	PIN_E16	7-segment display segment e
seg[5]	PIN_D17	7-segment display segment f
seg[6]	PIN_C17	7-segment display segment g

Table 1: FPGA Pin Assignments

Resource Type	Used	Available	Utilization
Logic Elements	98	22,320	1%(less)
Registers	47	22,320	1%(less)
Memory Bits	0	608,256	0%
PLLs	0	4	0%

Table 2: FPGA Resource Utilization

6.4 Resource Utilization

The UART transceiver implementation requires minimal FPGA resources: The implementation is very resource-efficient, using less than 1

7 Practical Demonstration

7.1 Verification Methodology

The UART implementation was verified through a series of tests:

1. **Simulation Testing**: Initial verification through ModelSim/Questa simulation with loopback configuration

2. **Single-Board Testing**: Implementation on a single FPGA board with loopback connection (TXD to RXD)

3. **Two-Board Communication**: Connection of two FPGA boards to demonstrate bidirectional communication
4. **Visual Verification**: Observation of received data on LEDs and 7-segment displays

7.2 Test Scenarios

Several test scenarios were implemented to verify different aspects of the UART communication:

1. **Fixed Data Transmission**: Sending the predefined value (0x05) to verify basic operation
2. **Bit Pattern Testing**: Sending various bit patterns (0x55, 0xAA, 0xFF, 0x00) to check all possible bit transitions
3. **Continuous Transmission**: Repeatedly sending data to verify sustained operation
4. **Reset Recovery**: Testing the system's ability to recover from mid-frame resets

7.3 Results and Observations

All test scenarios were successfully completed with the following observations:

- **Transmission Reliability**: The UART transceiver reliably transmitted and received data in all test scenarios.
- **Visual Verification**: The received data was correctly displayed on the LEDs and 7-segment display, confirming proper operation.
- **Timing Stability**: No timing issues or data corruption was observed during extended operation.
- **Reset Recovery**: The system correctly returned to idle state after reset and was ready for new transmissions.

8 Sustainability and Future Enhancements

8.1 Design Sustainability

The current UART implementation is designed with sustainability in mind:

1. **Modularity**: The design is divided into distinct modules (transmitter, receiver, 7-segment decoder) that can be reused independently.
2. **Parameterization**: Key parameters like clock frequency and baud rate are configurable, allowing adaptation to different requirements.
3. **Documentation**: Comprehensive documentation of the design, implementation, and testing process ensures maintainability.

4. **Verification Infrastructure**: The testbench provides a reusable framework for testing modifications or enhancements.

8.2 Potential Enhancements

The current implementation can be extended in several ways:

1. **Parameterizable Data Width**: Modify the design to support different data widths (7, 8, or 9 bits).
2. **Parity Support**: Add optional parity bit generation and checking for error detection.
3. **Multiple Stop Bits**: Support configurable number of stop bits (1, 1.5, or 2).
4. **FIFO Buffers**: Add transmit and receive FIFOs to handle bursts of data.
5. **Baud Rate Detection**: Implement automatic baud rate detection for the receiver.
6. **Error Detection**: Add framing error and noise detection capability.
7. **Flow Control**: Implement hardware flow control (RTS/CTS) for reliable data transfer.

8.3 Implementation of Enhancements

To implement these enhancements, the following modifications would be required:

1. **Parameterizable Data Width**:

```
// Add data width parameter
parameter DATA_WIDTH = 8,
// Modify shift register and bit counter width
reg [DATA_WIDTH-1:0] shift_reg;
reg [log2(DATA_WIDTH+2)-1:0] bit_index;
```

2. **Parity Support**:

```
// Add parity parameters
parameter USE_PARITY = 0,
parameter PARITY_EVEN = 1,

// Calculate parity bit
wire parity = PARITY_EVEN ? ^data : ~^data;

// Include parity in frame
shift_reg <= USE_PARITY ?
             {1'b1, parity, data, 1'b0} :
             {1'b1, data, 1'b0};
```

3. **FIFO Buffers**:

```
// Define FIFO module
module uart_fifo #(
    parameter WIDTH = 8,
    parameter DEPTH = 16
)(
    input wire clk,
    input wire rst,
    input wire [WIDTH-1:0] data_in,
    input wire write,
    output wire [WIDTH-1:0] data_out,
    input wire read,
    output wire empty,
    output wire full
);
    // FIFO implementation
    // ...
endmodule
```

9 Learning Outcomes

Through this UART implementation project, several key learning outcomes were achieved:

1. **Digital Design Skills**: Practical application of synchronous digital design principles, including state machines and timing considerations.
2. **Protocol Implementation**: Understanding and implementation of a standard communication protocol (UART).
3. **FPGA Development**: Experience with the complete FPGA development flow, from design and coding to simulation, synthesis, and hardware verification.
4. **System Integration**: Integration of multiple digital components into a cohesive system.
5. **Hardware Debugging**: Practical experience in debugging hardware designs using simulation and physical testing.
6. **Documentation**: Development of technical documentation skills for complex digital systems.

10 Conclusion

The UART transceiver was successfully implemented on the DE0-Nano FPGA board, meeting all the project objectives:

1. A complete UART transceiver was designed and implemented with both transmitter and receiver functionality.
2. The implementation was verified through simulation, showing correct protocol behavior and timing.

3. Hardware communication between two FPGA boards was demonstrated, proving the practical applicability of the design.
4. Visual verification through LEDs and 7-segment displays confirmed the correct operation of the system.
5. The modular and parameterized design provides a sustainable foundation for future enhancements.

This project provided valuable hands-on experience with digital design, FPGA implementation, and communication protocols, reinforcing theoretical concepts with practical application. The resulting UART implementation is a reusable component that can serve as a building block for more complex digital systems requiring serial communication capabilities.

11 References

1. Altera DE0-Nano Development and Education Board, Terasic Technologies Inc.
2. "UART Protocol and Interface," Electronic Industries Association (EIA) Standard RS-232.
3. Quartus Prime Design Software, Intel FPGA.
4. ModelSim-Altera Simulation Tool, Intel FPGA.
5. "Digital Design with Hardware Description Languages," Frank Vahid, Roman Lysecky.
6. "FPGA Prototyping By Verilog Examples," Pong P. Chu.

12 Appendix

12.1 Timing Calculations

For a system clock of 50 MHz and a baud rate of 115,200 bps:

$$\text{Baud Rate Divisor} = \frac{\text{System Clock}}{\text{Baud Rate}} = \frac{50,000,000}{115,200} = 434.03 \quad (1)$$

Using a divisor of 434:

$$\text{Actual Baud Rate} = \frac{\text{System Clock}}{\text{Divisor}} = \frac{50,000,000}{434} = 115,207.37 \text{ bps} \quad (2)$$

This gives a baud rate error of:

$$\text{Error} = \frac{115,207.37 - 115,200}{115,200} \times 100 = 0.0064 \quad (3)$$

This error is well within the acceptable range for UART communication (typically ±5

12.2 Complete Project Files

The complete project includes the following files:

- uart_tx.v - UART transmitter module
- uart_rx.v - UART receiver module
- invert_uart_transceiver_test.v - Top-level transceiver module
- binary_to_7seg.v -segment display decoder
- tb_invert_uart_transceiver_test.v - Testbench
- uart_project.qpf - Quartus project file
- uart_project.qsf - Quartus settings file (including pin assignments)
- uart_transceiver.sdc - Timing constraints file