Department of Electronic and Telecommunication Engineering

University of Moratuwa Faculty of Engineering

A logo of a university

Description automatically generated

Electronic Circuit Design EN2111

**UART implementation in FPGA**

**Group - 35**

|  |  |
| --- | --- |
| **Name** | **Index Number** |
|  |  |
|  |  |
|  |  |

This report is submitted as the fulfillment of FPGA Assignment of module EN2111

Table of Contents

[**Abstract:** 3](#_Toc165910441)

[**Introduction:** 3](#_Toc165910442)

[**Methodology and Observations:** 3](#_Toc165910443)

[1. Setup and Hardware Configuration: 3](#_Toc165910444)

[2. Data Packet Transmission: 3](#_Toc165910445)

[3. Data Conversion and Reception: 3](#_Toc165910446)

[**Appendices** 4](#_Toc165910447)

[ Transmitter 4](#_Toc165910448)

[ Receiver 6](#_Toc165910449)

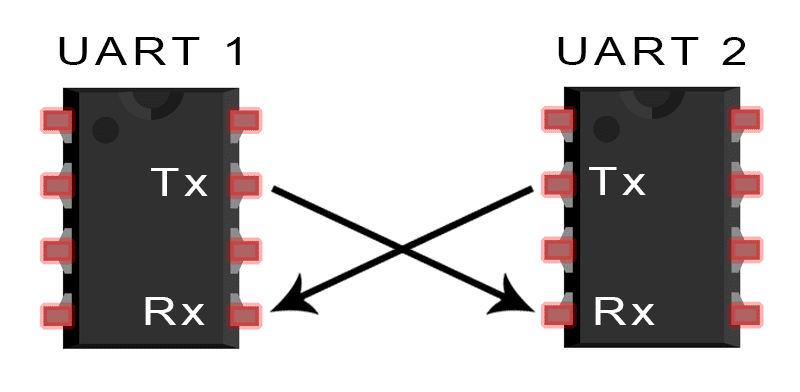
[ Binary to seven-segment display 7](#_Toc165910450)

[ Testbench 8](#_Toc165910451)

## **Abstract:**

This lab report explores UART, a hardware communication protocol for serial communication between Duo Nano boards, focusing on objectives, methodology, observations, and findings. It provides an in-depth understanding of the protocol's practical implementation.

## **Introduction:**



UART is a widely adopted hardware communication protocol for seamless device-to-device communication in embedded systems, microcontrollers, and computers, requiring only two wires for transmitting and receiving data. UART uses asynchronous serial communication without a clock signal, utilizing start, data frame, parity, and stop bits for accurate data transfer. Understanding UART principles is crucial for developing device-to-device communication products. The lab aimed to establish reliable serial communication between Duo Nano boards using UART, demonstrating proper configuration and frame protocol utilization for secure, error-free communication.

## **Methodology and Observations:**

### 1. Setup and Hardware Configuration:

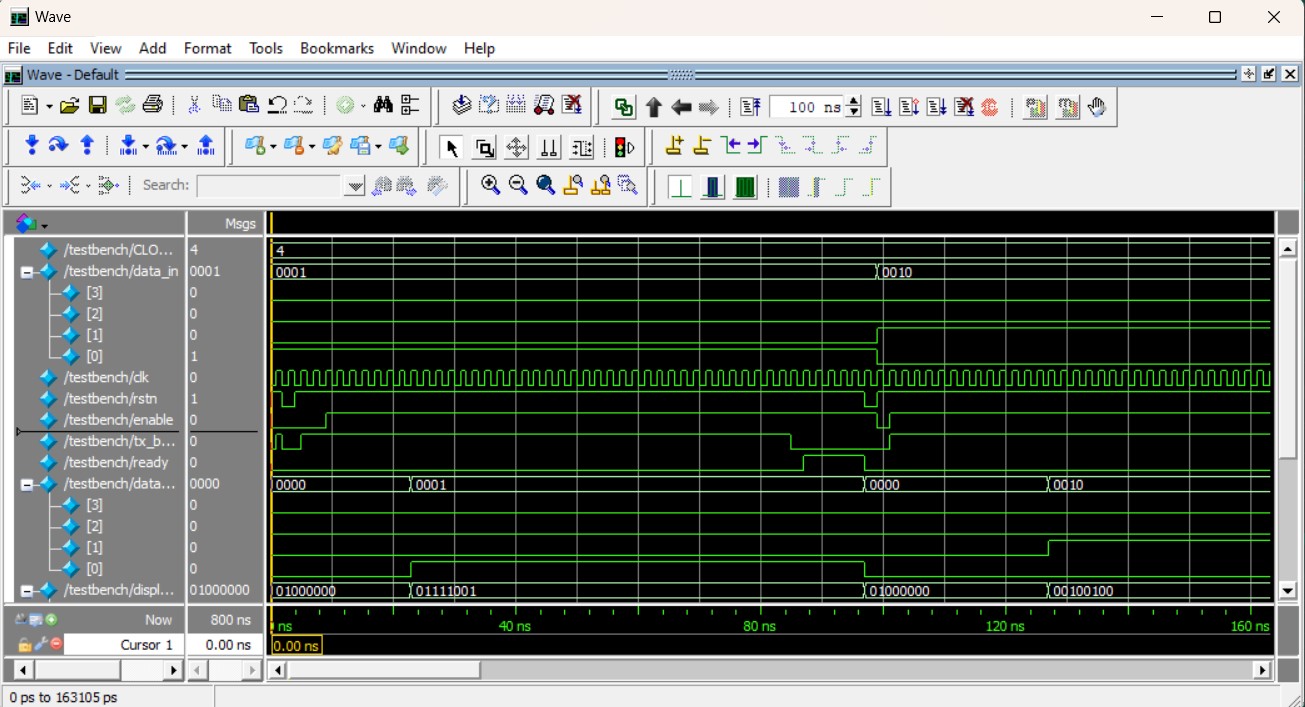
* Two Duo Nano boards were interconnected via UART, where one served as the transmitter (Tx) and the other as the receiver (Rx).
* UART transmitter of one board was linked to the UART receiver of the other, establishing a bidirectional communication link.
* A specific baud rate was configured on both boards to ensure synchronization.

### 2. Data Packet Transmission:

* Data packets were transmitted from the transmitting board to the receiving board using UART.
* Each packet comprised a start bit, data frame, parity bit, and stop bit(s).
* The start bit initiated data transmission, followed by the data frame containing the actual data.
* Parity bits were integrated into the data frame to detect and validate any transmission errors or alterations.
* Stop bits marked the conclusion of each data packet, facilitating clear differentiation between successive packets.

### 3. Data Conversion and Reception:

* The receiving board processed the transmitted data packets upon reception.
* It disregarded the start bit, parity bit, and stop bit(s) to extract the actual data frame.
* Serial data received was converted back into a parallel format for subsequent processing or display.



## **Appendices**

### Transmitter

module transmitter #(

    parameter CLOCKS\_PER\_PULSE = 16

)

(

    input logic [7:0] data\_in,

    input logic data\_en,

    input logic clk,

    input logic rstn,

    output logic tx,

    output logic tx\_busy

);

    enum {TX\_IDLE, TX\_START, TX\_DATA, TX\_END} state;

    logic[7:0] data = 8'b0;

    logic[2:0] c\_bits = 3'b0;

    logic[$clog2(CLOCKS\_PER\_PULSE)-1:0] c\_clocks = 0;

    always\_ff @(posedge clk or negedge rstn) begin

        if (!rstn) begin

            c\_clocks <= 0;

            c\_bits <= 0;

            data <= 0;

            tx <= 1'b1;

            state <= TX\_IDLE;

        end else begin

            case (state)

            TX\_IDLE: begin

                if (~data\_en) begin

                    state <= TX\_START;

                    data <= data\_in;

                    c\_bits <= 3'b0;

                    c\_clocks <= 0;

                end else tx <= 1'b1;

            end

            TX\_START: begin

                if (c\_clocks == CLOCKS\_PER\_PULSE-1) begin

                    state <= TX\_DATA;

                    c\_clocks <= 0;

                end else begin

                    tx <= 1'b0;

                    c\_clocks <= c\_clocks + 1;

                end

            end

            TX\_DATA: begin

                if (c\_clocks == CLOCKS\_PER\_PULSE-1) begin

                    c\_clocks <= 0;

                    if (c\_bits == 3'd7) begin

                        state <= TX\_END;

                    end else begin

                        c\_bits <= c\_bits + 1;

                        tx <= data[c\_bits];

                    end

                end else begin

                    tx <= data[c\_bits];

                    c\_clocks <= c\_clocks + 1;

                end

            end

            TX\_END: begin

                if (c\_clocks == CLOCKS\_PER\_PULSE-1) begin

                    state <= TX\_IDLE;

                    c\_clocks <= 0;

                end else begin

                    tx <= 1'b1;

                    c\_clocks <= c\_clocks + 1;

                end

            end

            default: state <= TX\_IDLE;

            endcase

        end

    end

    assign tx\_busy = (state != TX\_IDLE);

endmodule

### Receiver

module receiver #(

    parameter CLOCKS\_PER\_PULSE = 16

)

(

    input logic clk,

    input logic rstn,

    input logic ready\_clr,

    input logic rx,

    output logic ready,

    output logic [7:0] data\_out

);

    enum {RX\_IDLE, RX\_START, RX\_DATA, RX\_END} state;

    logic[2:0] c\_bits;

    logic[$clog2(CLOCKS\_PER\_PULSE)-1:0] c\_clocks;

    logic[7:0] temp\_data;

    logic rx\_sync;

    always\_ff @(posedge clk or negedge rstn) begin

        if (!rstn) begin

            c\_clocks <= 0;

            c\_bits <= 0;

            temp\_data <= 8'b0;

            //data\_out <= 8'b0;

            ready <= 0;

            state <= RX\_IDLE;

        end else begin

            rx\_sync <= rx;  // Synchronize the input signal using a flip-flop

            case (state)

            RX\_IDLE : begin

                if (rx\_sync == 0) begin

                    state <= RX\_START;

                    c\_clocks <= 0;

                end

            end

            RX\_START: begin

                if (c\_clocks == CLOCKS\_PER\_PULSE/2-1) begin

                    state <= RX\_DATA;

                    c\_clocks <= 0;

                end else

                    c\_clocks <= c\_clocks + 1;

            end

            RX\_DATA : begin

                if (c\_clocks == CLOCKS\_PER\_PULSE-1) begin

                    c\_clocks <= 0;

                    temp\_data[c\_bits] <= rx\_sync;

                    if (c\_bits == 3'd7) begin

                        state <= RX\_END;

                        c\_bits <= 0;

                    end else c\_bits <= c\_bits + 1;

                end else c\_clocks <= c\_clocks + 1;

            end

            RX\_END : begin

                if (c\_clocks == CLOCKS\_PER\_PULSE-1) begin

                    //data\_out <= temp\_data;

                    ready <= 1'b1;

                    state <= RX\_IDLE;

                    c\_clocks <= 0;

                end else c\_clocks <= c\_clocks + 1;

            end

            default: state <= RX\_IDLE;

            endcase

        end

    end

    assign data\_out = temp\_data;

endmodule

### 

### Binary to seven-segment display

module binary\_to\_7seg (

    input logic [3:0] data\_in,

    output logic [6:0] data\_out

);

    // Make a LUT to convert digits to 7 segment output

    // Input - 4 bits, output - 7 bits

    logic [15:0][6:0] lut\_7seg;

    // Output is gfedcba

    assign lut\_7seg[0] = 7'b0111111;

    assign lut\_7seg[1] = 7'b0000110;

    assign lut\_7seg[2] = 7'b1011011;

    assign lut\_7seg[3] = 7'b1001111;

    assign lut\_7seg[4] = 7'b1100110;

    assign lut\_7seg[5] = 7'b1101101;

    assign lut\_7seg[6] = 7'b1111101;

    assign lut\_7seg[7] = 7'b0000111;

    assign lut\_7seg[8] = 7'b1111111;

    assign lut\_7seg[9] = 7'b1101111;

    assign lut\_7seg[15:10] = 7'b0;    // unused

     assign data\_out = ~lut\_7seg[data\_in];

endmodule

/\*

Seven segment display

    a

f       b

    g

e       c

    d

\*/

### Testbench

`timescale 1ns/1ps

module testbench();

    localparam CLOCKS\_PER\_PULSE = 4;

  logic [3:0] data\_in = 4'b0001;

    logic clk = 0;

    logic rstn = 0;

    logic enable = 1;

    logic tx\_busy;

    logic ready;

    logic [7:0] data\_out;

    logic [7:0] display\_out;

    logic loopback;

    logic ready\_clr = 1;

    uart #(.CLOCKS\_PER\_PULSE(CLOCKS\_PER\_PULSE))

            test\_uart(.data\_in(data\_in),

                        .data\_en(enable),

                        .clk(clk),

                        .tx(loopback),

                        .tx\_busy(tx\_busy),

                        .rx(loopback),

                        .ready(ready),

                        .ready\_clr(ready\_clr),

                        .led\_out(data\_out),

                        .display\_out(display\_out),

                        .rstn(rstn)

                        );

    always begin

        #1 clk = ~clk;

    end

    initial begin

        $dumpfile("testbench.vcd");

        $dumpvars(0, testbench);

        rstn <= 1;

        enable <= 1'b0;

        #2 rstn <= 0;

        #2 rstn <= 1;

        #5 enable <= 1'b1;

    end

    always @(posedge ready) begin

        if (data\_out != data\_in) begin

            $display("FAIL: rx data %x does not match tx %x", data\_out, data\_in);

         $finish();

        end else begin

          if (data\_out == 4'b1111) begin //Check if received data is 11111111

                $display("SUCCESS: all bytes verified");

                $finish();

            end

            #10 rstn <= 0;

            #2 rstn <= 1;

            data\_in <= data\_in + 1'b1;

            enable <= 1'b0;

            #2 enable <= 1'b1;

        end

    end

endmodule