Department of Electronic & Telecommunication Engineering,
University of Moratuwa, Sri Lanka.

Lab Assignment – Project on UART Implementation

Submitted By:

210705E	Wickramasinghe M.P.D.N.
210724K	Wijethilaka U.K.G.P.M.B.
210728C	Wijewickrama W.K.D.D.D.

Submitted in partial fulfillment of the requirements for the module

EN2111 Electronic Circuits Design

8th of May 2024

Contents

1 Introduction

Universal Asynchronous Receiver-Transmitter (UART) is a foundational protocol for serial communication, facilitating data exchange between electronic devices without requiring a shared clock signal. In this report, we explore the implementation of UART on the Altera Deo Nano Development board, leveraging the flexibility of Field-Programmable Gate Arrays (FPGAs) to configure serial communication protocols. With its Altera Cyclone IV FPGA, the Deo Nano board provides an ideal platform for experimenting with UART functionality, enabling us to delve into design considerations, implementation steps, and performance evaluation.

2 Overview - UART

UART (Universal Asynchronous Receiver-Transmitter) is a fundamental communication protocol for serial data exchange between electronic devices. In UART communication, data is transmitted asynchronously, meaning there is no shared clock signal between the transmitter and receiver for synchronization. Instead, UART relies on predefined baud rates to regulate the timing of data transmission and reception.

At the transmitter end, parallel data is converted into serial format and transmitted sequentially, one bit at a time. This serial data stream is framed with a start bit to indicate the beginning of a transmission and one or more stop bits to denote the end. These bits sandwich the data bits, ensuring synchronization and proper framing of the transmitted data.

On the receiving end, the UART receiver continuously monitors the incoming data stream. Upon detecting the start bit, it begins clocking in the data bits at the specified baud rate. Once all the data bits, along with any associated parity bits, are received, the stop bit(s) signify the end of the transmission. The receiver then reconstructs the original parallel data from the received serial stream for further processing.

Overall, UART's simplicity, versatility, and asynchronous operation make it a widely used protocol in various embedded systems and communication interfaces, enabling efficient data exchange between devices.

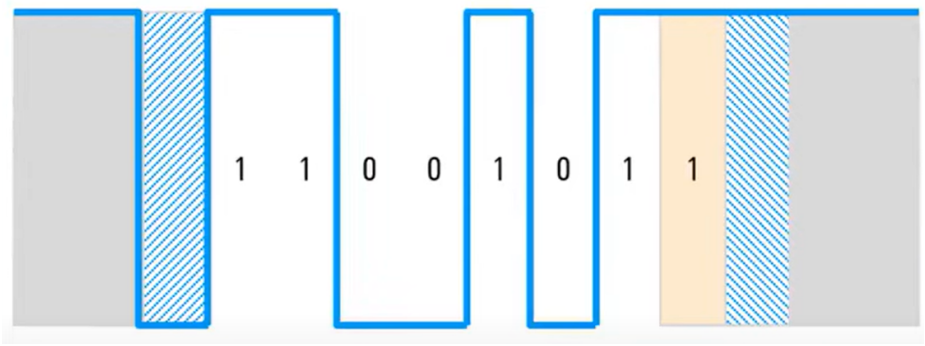

Figure 1: UART

3 Implementation and Results

3.1 Codes

3.1.1 TX

Listing 1: Transmitter

```
module transmitter(      input wire [7:0] data_in, //input data as an 8-bit regsiter/ve
                                                   //input wire wr_en, //enable u
                                                   input wire clk_50m,
                                                   input wire clken, //clock sign
                                                   output reg Tx, //a single 1-bi
                                                   output wire Tx_busy //transmit
                                                   );

initial begin
        Tx = 1'b1; //initialize Tx = 1 to begin the transmission
end
//Define the 4 states using 00,01,10,11 signals
parameter TX_STATE_IDLE = 2'b00;
parameter TX_STATE_START        = 2'b01;
parameter TX_STATE_DATA = 2'b10;
parameter TX_STATE_STOP = 2'b11;

reg [7:0] data = 8'h00; //set an 8-bit register/vector as data,initially equal to 0000
reg [2:0] bit_pos = 3'h0; //bit position is a 3-bit register/vector, initially equal t
reg [1:0] state = TX_STATE_IDLE; //state is a 2 bit register/vector,initially equal to
reg wr_en = 1'b0;

always @(posedge clk_50m) begin
        case (state) //Let us consider the 4 states of the transmitter
        TX_STATE_IDLE: begin //We define the conditions for idle
or NOT-BUSY state
                if (~wr_en) begin
                        state <= TX_STATE_START; //assign the start signal to state
                        data <= data_in; //we assign input data vector to the current
                        bit_pos <= 3'h0; //we assign the bit position to zero
                end
        end
        TX_STATE_START: begin //We define the conditions for the transmission start st
                if (clken) begin
                        Tx <= 1'b0; //set Tx = 0 indicating transmission has started
                        state <= TX_STATE_DATA;
                end
        end
        TX_STATE_DATA: begin
                if (clken) begin
                        if (bit_pos == 3'h7) //we keep assigning Tx with the data unti
                                state <= TX_STATE_STOP; // when bit position has final
                        else
                                bit_pos <= bit_pos + 3'h1; //increment the bit positio
                        Tx <= data[bit_pos]; //Set Tx to the data value of the bit pos
                end
        end
```

```
        TX_STATE_STOP: begin
                if (clken) begin
                        Tx <= 1'b1; //set Tx = 1 after transmission has ended
                        state <= TX_STATE_IDLE; //Move to IDLE state once a transmissi
                end
        end
        default: begin
                Tx <= 1'b1; // always begin with Tx = 1 and state assigned to IDLE
                state <= TX_STATE_IDLE;
        end
        endcase
end

assign Tx_busy = (state != TX_STATE_IDLE); //We assign the BUSY signal when the transn

endmodule
```

3.1.2 RX

Listing 2: Reciever Verilog

```
module receiver  (input wire Rx,
                                                output reg ready,
                                                //input wire ready_clr,
                                                input wire clk_50m,
                                                input wire clken,
                                                output reg [7:0] data   // 8 bit regis
                                                );
initial begin
        ready = 1'b0; // initialize ready = 0
        data = 8'b11111111; // initialize data as 00000000
end
// Define the 4 states using 00,01,10 signals
parameter RX_STATE_START        = 2'b00;
parameter RX_STATE_DATA         = 2'b01;
parameter RX_STATE_STOP         = 2'b10;

reg [1:0] state = RX_STATE_START; // state is a 2-bit register/vector, initially equal
reg [3:0] sample = 0; // This is a 4-bit register
reg [3:0] bit_pos = 0; // bit position is a 4-bit register/vector, initially equal to
reg [7:0] scratch = 8'b0; // An 8-bit register assigned to 00000000
reg ready_clr = 1'b0;

always @(posedge clk_50m) begin
        if (ready_clr)
                ready <= 1'b0; // This resets ready to 0

        if (clken) begin
                case (state) // Let us consider the 3 states of the receiver
                RX_STATE_START: begin // We define condtions for starting the receiver
                        if (!Rx || sample != 0) // start counting from the first low s
                                sample <= sample + 4'b1; // increment by 0001
                        if (sample == 15) begin // once a full bit has been sampled
                                state <= RX_STATE_DATA; //      start collecting data
```

```
                                bit_pos <= 0;
                                sample <= 0;
                                scratch <= 0;
                        end
                end
                RX_STATE_DATA: begin // We define conditions for starting the data col
                        sample <= sample + 4'b1; // increment by 0001
                        if (sample == 4'h8) begin // we keep assigning Rx data until c
                                scratch[bit_pos[2:0]] <= Rx;
                                bit_pos <= bit_pos + 4'b1; // increment by 0001
                        end
                        if (bit_pos == 8 && sample == 15) // when a full bit has been
                                state <= RX_STATE_STOP; // bit position has finally re
                end
                RX_STATE_STOP: begin
                        /*
                         * Our baud clock may not be running at exactly the
                         * same rate as the transmitter.
If we thing that
                         * we're at least half way into the stop bit, allow
                         * transition into handling the next start bit.
                         */
                        if (sample == 15 || (sample >= 8 && !Rx)) begin
                                state <= RX_STATE_START;
                                data <= ~scratch;
                                ready <= 1'b1;
                                sample <= 0;
                        end
                        else begin
                                sample <= sample + 4'b1;
                        end
                end
                default: begin
                        state <= RX_STATE_START; // always begin with state assigned t
                end
                endcase
        end
end

endmodule
```

3.1.3 Baud Rate

Listing 3: Baud Rate Verilog

```
//This is a baud rate generator to divide a 50MHz clock into a 115200 baud Tx/Rx pair.
//The Rx clock oversamples by 16x.

module baudrate  (input wire clk_50m,
                                                output wire Rxclk_en,
                                                output wire Txclk_en
                                                );
//Our Testbench uses a 50 MHz clock.
//Want to interface to 115200 baud UART for Tx/Rx pair
```

```
//Hence, 50000000 / 115200 = 435 Clocks Per Bit.
parameter RX_ACC_MAX = 50000000 / (115200 * 16);
parameter TX_ACC_MAX = 50000000 / 115200;
parameter RX_ACC_WIDTH = $clog2(RX_ACC_MAX);
parameter TX_ACC_WIDTH = $clog2(TX_ACC_MAX);
reg [RX_ACC_WIDTH - 1:0] rx_acc = 0;
reg [TX_ACC_WIDTH - 1:0] tx_acc = 0;

assign Rxclk_en = (rx_acc == 5'd0);
assign Txclk_en = (tx_acc == 9'd0);

always @(posedge clk_50m) begin
        if (rx_acc == RX_ACC_MAX[RX_ACC_WIDTH - 1:0])
                rx_acc <= 0;
        else
                rx_acc <= rx_acc + 5'b1; //increment by 00001
end

always @(posedge clk_50m) begin
        if (tx_acc == TX_ACC_MAX[TX_ACC_WIDTH - 1:0])
                tx_acc <= 0;
        else
                tx_acc <= tx_acc + 9'b1; //increment by 000000001
end

endmodule
```

3.1.4 Main

Listing 4: UART.V

```
module uart(input wire [7:0] data_in, //input data
                             input wire clear,
                             input wire clk_50m,
                             output wire Tx,
                             output wire Tx_busy,
                             input wire Rx,
                             output wire ready,
                             output wire [7:0] data_out,
                             output [7:0] LEDR,
                             output wire Tx2//output data
                             );
assign LEDR = data_in;
assign Tx2 = Tx;
//assign wr_en = 1'b1;
//assign ready_clr = 1'b1;
wire Txclk_en, Rxclk_en;
baudrate uart_baud(      .clk_50m(clk_50m),
                                                .Rxclk_en(Rxclk_en),
                                                .Txclk_en(Txclk_en)
                                                );
transmitter uart_Tx(     .data_in(data_in),
                                                //.wr_en(wr_en),
```

```
                                                    .clk_50m(clk_50m),
                                                    .clken(Txclk_en), //We assign
                                                    .Tx(Tx),
                                                    .Tx_busy(Tx_busy)
                                                    );
receiver uart_Rx(       .Rx(Rx),
                                                .ready(ready),
                                                //.ready_clr(ready_clr),
                                                .clk_50m(clk_50m),
                                                .clken(Rxclk_en), //We assign Tx clock
                                                .data(data_out)
                                                );
```

endmodule

3.1.5 Testbench

Listing 5: UART TB.V

```
//This is a simple testbench for UART Tx and Rx.
//The Tx and Rx pins have been connected together creating a serial loopback.
//We check if we receive what we have transmitted by sending incremeting data bytes.

//It sends out byte 0xAB over the transmitter
//It then exercises the receive by receiving byte 0x3F
//`include "uart.v"

//`timescale 1ns/1ps

module uart_TB();

reg [7:0] data = 0;
reg clk = 0;
reg enable = 0;

wire Tx_busy;
wire ready;
wire [7:0] Rx_data;

wire loopback;
reg ready_clr = 0;

reg [7:0] sent_tx = 0;

reg init = 0;

uart test_uart(.data_in(data),
                                    .wr_en(enable),
                                    .clk_50m(clk),
                                    .Tx(loopback),
                                    .Tx_busy(Tx_busy),
                                    .Rx(loopback),
                                    .ready(ready),
                                    .ready_clr(ready_clr),
```

```
                                        .data_out(Rx_data)
                                        );
initial begin
        $dumpfile("uart.vcd");
        $dumpvars(0, uart_TB);
        enable <= 1'b1;
        #2 enable <= 1'b0;
        init <= 1;

end
always begin
        #1 clk = ~clk;
end
always @(posedge ready or init) begin
        #2 ready_clr <= 1;
        #2 ready_clr <= 0;
        if (Rx_data != sent_tx) begin
                $display("FAIL: rx data %x does not match tx %x", Rx_data, sent_tx);
                //$finish;
        end
        else begin
                if (Rx_data == 8'h2) begin //Check if received data is 11111111
                        $display("SUCCESS: all bytes verified");
                        //$finish;
                end
                sent_tx <= data;
                data <= data + 1'b1;
                enable <= 1'b1;
                #2 enable <= 1'b0;
        end
end
endmodule
```

3.1.6 Implementation on RPi

Listing 6: Python code for GPIO control on Raspberry Pi

```
import RPi.GPIO as GPIO
import time

# Use Broadcom SOC Pin numbers
GPIO.setmode(GPIO.BCM)

# Setup the output GPIO pins
pins = [14, 15, 18, 23, 24, 25, 8, 7]  # Example GPIO pins
for pin in pins:
    GPIO.setup(pin, GPIO.OUT)

def output_binary(number):
    binary_format = '{:08b}'.format(number)  # Format number as binary
    for i, bit in enumerate(binary_format):
        GPIO.output(pins[i], int(bit))  # Set each pin to high or low depending on bit

try:
```

```
    while True:
        user_input = input("Enter a number (0-255): ")  # Get user input
        try:
            num = int(user_input)  # Convert input to integer
            if 0 <= num <= 255:
                output_binary(255-num)  # Output the number in binary on GPIO pins
                time.sleep(0.5)  # Wait half a second between updates
            else:
                print("Number out of range. Please enter a number between 0 and 255.")
        except ValueError:
            print("Invalid input. Please enter a valid integer.")

except KeyboardInterrupt:
    print("Program stopped by user.")

finally:
    GPIO.cleanup()  # Clean up GPIO on exit
```

3.1.7 Pin Planner

Rx	Input	PIN_A7	8	B8_N0	PIN_A7	3.3-...CMOS	2mA (default)		
Tx	Output	PIN_B7	8	B8_N0	PIN_B7	3.3-...CMOS	2mA (default)	2 (default)	
Tx2	Output				PIN_M8	2.5 V ...fault)	8mA (default)	2 (default)	
Tx_busy	Output				PIN_M10	2.5 V ...fault)	8mA (default)	2 (default)	
clear	Input				PIN_J16	2.5 V ...fault)	8mA (default)		
clk_50m	Input	PIN_R8	3	B3_N0	PIN_R8	3.3-...CMOS	2mA (default)		
data_in[7]	Input	PIN_F13	6	B6_N0	PIN_F13	3.3-...CMOS	2mA (default)		
data_in[6]	Input	PIN_T15	4	B4_N0	PIN_T15	3.3-V LVTTL	8mA (default)		
data_in[5]	Input	PIN_T13	4	B4_N0	PIN_T13	3.3-V LVTTL	8mA (default)		
data_in[4]	Input	PIN_T12	4	B4_N0	PIN_T12	3.3-V LVTTL	8mA (default)		
data_in[3]	Input	PIN_T11	4	B4_N0	PIN_T11	3.0-V LVTTL	8mA (default)		
data_in[2]	Input	PIN_R11	4	B4_N0	PIN_R11	3.3-V LVTTL	8mA (default)		
data_in[1]	Input	PIN_R10	4	B4_N0	PIN_R10	3.3-V LVTTL	8mA (default)		
data_in[0]	Input	PIN_P9	4	B4_N0	PIN_P9	3.0-V LVTTL	8mA (default)		
data_out[7]	Output	PIN_A15	7	B7_N0	PIN_A15	3.3-...CMOS	2mA (default)	2 (default)	
data_out[6]	Output	PIN_A13	7	B7_N0	PIN_A13	3.3-...CMOS	2mA (default)	2 (default)	
data_out[5]	Output	PIN_B13	7	B7_N0	PIN_B13	3.3-...CMOS	2mA (default)	2 (default)	
data_out[4]	Output	PIN_A11	7	B7_N0	PIN_A11	3.3-...CMOS	2mA (default)	2 (default)	
data_out[3]	Output	PIN_D1	1	B1_N0	PIN_D1	3.3-...CMOS	2mA (default)	2 (default)	
data_out[2]	Output	PIN_F3	1	B1_N0	PIN_F3	3.3-...CMOS	2mA (default)	2 (default)	
data_out[1]	Output	PIN_B1	1	B1_N0	PIN_B1	3.3-...CMOS	2mA (default)	2 (default)	
data_out[0]	Output	PIN_L3	2	B2_N0	PIN_L3	3.3-...CMOS	2mA (default)	2 (default)	
ready	Output				PIN_R5	2.5 V ...fault)	8mA (default)	2 (default)	
wr_en	Unknown	PIN_M1	2	B2_N0		3.3-...CMOS	2mA (default)		
ready_clr	Unknown	PIN_T8	3	B3_N0		3.3-...CMOS	2mA (default)		

Figure 2: Pin Planner

3.2 Results

3.2.1 Simulation Results

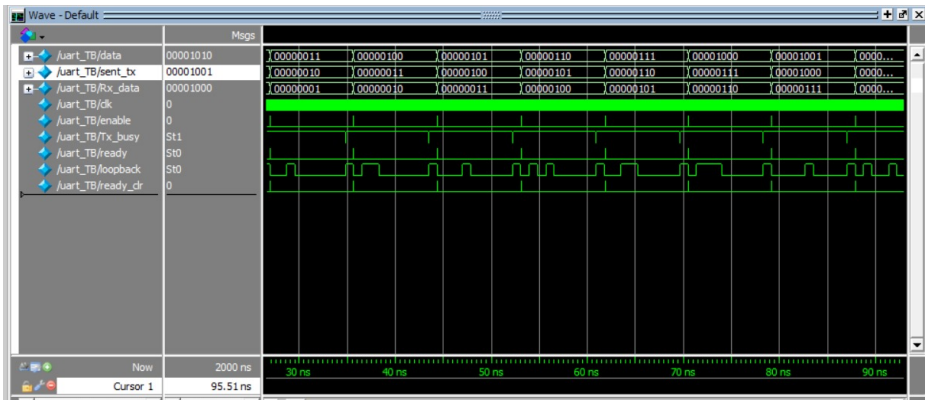

Figure 3: Simulation Results

3.2.2 Implementation on FPGA

Our methodology entails utilizing a Raspberry Pi as the input source, generating a numerical value within the range of 0 to 255. This numerical value is then converted into its binary representation, which is subsequently conveyed to the Field-Programmable Gate Array (FPGA) via the General Purpose Input/Output (GPIO) pins of the Raspberry Pi. Leveraging the UART communication protocol, the FPGA transmits this binary data to another FPGA-equipped board.

This approach ensures a systematic flow of data from the Raspberry Pi to the FPGA, where the FPGA serves as an intermediary for the transmission of binary data between the two boards. By integrating the capabilities of both the Raspberry Pi and the FPGA, we establish a robust framework for serial data exchange, facilitating seamless communication between the interconnected devices.

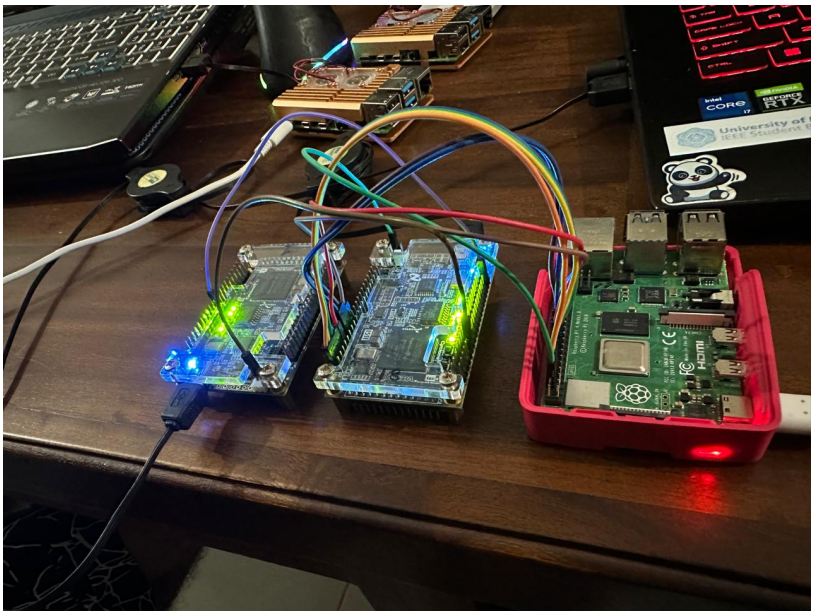

Figure 4: Communication with RPi

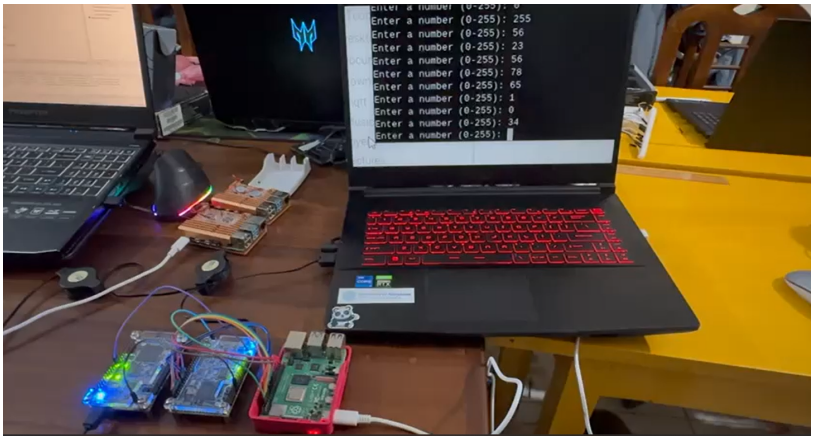

Figure 5: With Serial Monitor

Conclusion

In conclusion, UART remains a cornerstone in digital communication due to its simplicity, versatility, and asynchronous operation. Through the framing of data with start and stop bits and synchronization via baud rates, UART enables seamless serial data exchange between electronic devices. The implementation of UART on FPGA platforms like the DE2-115 board underscores its relevance in modern embedded systems. As technology continues to evolve, UART's enduring utility ensures its continued prominence in facilitating efficient and reliable data communication across diverse applications.