Department of Electronic and Telecommunication Engineering

University of Moratuwa

UART Implementation on FPGA

EN2111: Electronic Circuit Design

Group No: 31

Index Number	**Name**
210517E	Ranasinghe I.D.S.S.
210542B	Ratnayake R.M.L.H.
210549D	Rupasinghe N.P.S.S.

Contents

Chapter 1

RTL Design - Verilog Code Files

1.1 UART Transceiver - uart.v

```
    module uart(input wire [7:0] data_in, //input data
                input wire wr_en,
                input wire clear,
                input wire clk_50m,
                output wire Tx,
                output wire Tx_busy,
                input wire Rx,
                output wire ready,
                input wire ready_clr,
                output wire [7:0] data_out,
                output [7:0] LEDR,
                output wire Tx2           //output data
                );

assign LEDR = data_in;
assign Tx2 = Tx;
wire Txclk_en, Rxclk_en;

wire [7:0] dataFromROM;


ROM rom(.load_en(wr_en),
          .data_out(dataFromROM)
       );

baudrate uart_baud( .clk_50m(clk_50m),
                            .Rxclk_en(Rxclk_en),
                            .Txclk_en(Txclk_en)
                            );

transmitter uart_Tx(    .data_in(dataFromROM),
                            .wr_en(wr_en),
                            .clk_50m(clk_50m),
                            .clken(Txclk_en), //We assign Tx clock to enable clock
                            .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 to enable clock
                        .data(data_out)
                        );

endmodule
```

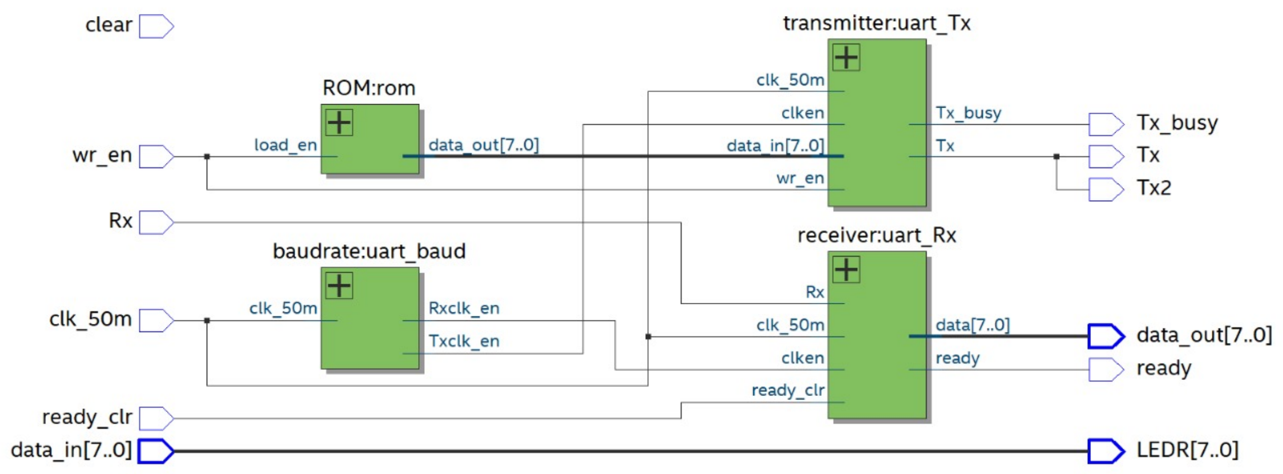


Figure 1.1: RTL View

1.2 Generating Baudrate for the Rx and Tx Clocks - baudrate.v

The baud rate is used to generate enabling pulses for the TX and RX, They are pulses that drive TX and RX when the Transmitter and Receiver are not in the IDLE state. It generates pulses with a given baudrate, in this case 115200.

The receiver is 16 times more sensitive than the transmitter.

```
    //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
                  );


//Want to interface to 115200 baud UART for Tx/Rx pair
//Hence, 50000000 / 115200 = 434 Clocks Per Bit.

parameter RX_ACC_MAX = 50000000 / (115200 * 16);      // = (1/115200) / (1/50000000)  / 16
parameter TX_ACC_MAX = 50000000 / 115200;
//parameter RX_ACC_MAX = 31;
//parameter TX_ACC_MAX = 511;
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;


always @(posedge clk_50m)
begin
    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

        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
end

```

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

endmodule
```

1.3 ROM - ROM.v

The architecture of the ROM is 32, 8-bit registers. We initialized it with predefined values.
The address is generated inside the ROM from 0 to 31, It runs iteratively then loads the data when enabled negedge of *load_en*. For given 5-bit data, with negative edges triggered in *load_en* (pressing the switch), 8-bit data is loaded to the *data_out*.

```verilog
    module ROM (
    input wire load_en,
    output reg [7:0] data_out
);

    reg [7:0] mem [31:0]; // 256 registers, each 8 bits wide
    reg [4:0] addr = 5'b0; // Address width changed to 8 bits


     initial begin
          mem[0]  = 8'b00000000;
          mem[1]  = 8'b00000001;
          mem[2]  = 8'b00000010;
          mem[3]  = 8'b00000100;
          mem[4]  = 8'b00001000;
          mem[5]  = 8'b00010000;
          mem[6]  = 8'b00100000;
          mem[7]  = 8'b01000000;
          mem[8]  = 8'b10000000;

          mem[9]  = 8'b10000000;
          mem[10] = 8'b01000000;
          mem[11] = 8'b00100000;
          mem[12] = 8'b00010000;
          mem[13] = 8'b00001000;
          mem[14] = 8'b00000100;
          mem[15] = 8'b00000010;
          mem[16] = 8'b00000001;
          mem[17] = 8'b00000000;

          mem[18] = 8'b00010000;
          mem[19] = 8'b00111000;
          mem[20] = 8'b01111100;
          mem[21] = 8'b11111110;
          mem[22] = 8'b11111111;
          mem[23] = 8'b00000000;


          mem[24] = 8'b11111111;
          mem[25] = 8'b11111110;
          mem[26] = 8'b01111100;
          mem[27] = 8'b00010000;
          mem[28] = 8'b11111111;
          mem[29] = 8'b00000000;
          mem[30] = 8'b10101010;
          mem[31] = 8'b01010101;
    end




     // Read operation
    always @(negedge load_en) begin
        addr <= (addr == 5'b11111) ? 8'b00000 : addr + 1; // Increment address cyclically
        data_out <= mem[addr];
    end

endmodule
```

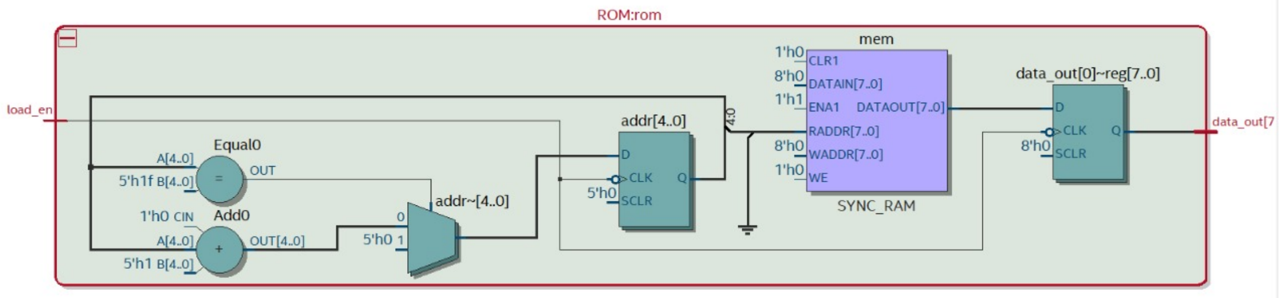


Figure 1.2: RTL View of ROM.v

1.4 UART Transmitter - transmitter.v

Initially, code is written in the form that when *wr_en* is logic 0, it transmits data until it goes high. To transmit data with edges of the *wr_en* we tried different methods.
We implemented the FLAG method – 2 flags to identify the negative edge of the *wr_en*.

We Trigger Tx with positive edges of the *wr_en* (giving a chance to load the data from the ROM at the negative edge – button pressing and we feed the data to Tx at the positive edge – button release) It was more reliable. Tx was implemented as a state machine.

```
module transmitter( input wire [7:0] data_in, //input data as an 8-bit regsiter/vector
                            input wire wr_en, //enable wire to start
                            input wire clk_50m,
                            input wire clken, //clock signal for the transmitter
                            output reg Tx, //a single 1-bit register variable to hold transmitting bit
                            output wire Tx_busy //transmitter is busy signal
                            );

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 00000000
reg [2:0] bit_pos = 3'h0;         //bit position is a 3-bit register/vector, initially equal to 000
reg [1:0] state = TX_STATE_IDLE;  //state is a 2 bit register/vector,initially equal to 00

reg flag1 = 1'b0;
reg flag2 = 1'b1;

always @(posedge wr_en)
begin
    flag1 <= ~flag1;
end

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 (flag1 == flag2)
            begin
                state <= TX_STATE_START; //assign the start signal to state
                data <= data_in; //we assign input data vector to the current data
                bit_pos <= 3'h0; //we assign the bit position to zero
                flag2 <= ~flag2;
            end
        end

    TX_STATE_START:
        begin //We define the conditions for the transmission start state
            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 until all bits have been transmit
                    state <= TX_STATE_STOP; // when bit position has finally reached 7, assign state to stop
                else
                    bit_pos <= bit_pos + 3'h1; //increment the bit position by 001
                Tx <= data[bit_pos]; //Set Tx to the data value of the bit position ranging from 0-7
            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 transmission has been completed
            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 transmitter is not idle.

endmodule
```

1.5 UART Receiver - receiver.v

```
    module receiver  (input wire Rx,
                        output reg ready,           // default 1 bit reg
                        input wire ready_clr,
                        input wire clk_50m,
                        input wire clken,
                        output reg [7:0] data   // 8 bit register
                        );
initial
begin
    ready = 1'b0; // initialize ready = 0
    data = 8'b0; // initialize data as 00000000
end

// Define the 3 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 to 00
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 0000
reg [7:0] scratch = 8'b0; // An 8-bit register assigned to 00000000

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:       // We define condtions for starting the receiver
        begin
            if (!Rx || sample != 0) // start counting from the first low sample
                sample <= sample + 4'b1; // increment by 0001

            if (sample == 15)           // once a full bit has been sampled
            begin
                state <= RX_STATE_DATA; //  start collecting data bits
                bit_pos <= 0;
                sample <= 0;
                scratch <= 0;
            end
        end



        RX_STATE_DATA:      // We define conditions for starting the data colleting
        begin
            sample <= sample + 4'b1;  // increment by 0001
            if (sample == 4'h8) begin // we keep assigning Rx data until all bits have 01 to 7
                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 sampled and
                state <= RX_STATE_STOP; // bit position has finally reached 7, assign state to stop
        end



        RX_STATE_STOP:
        begin
            /*
             * Our baud clock may not be running at exactly the
             * same rate as the transmitter.  If we think 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;
```

```verilog
                    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 to START
            end

            endcase
        end
end

endmodule
```

Chapter 2

Testbenches

2.1 Testbench for Baudrate - baudrate_tb.v

```verilog
    `timescale 1ps / 1ps

module baudrate_tb;

  // Inputs
  reg clk_50m;

  // Outputs
  wire Rxclk_en;
  wire Txclk_en;

  // Instantiate the baudrate module
  baudrate dut (
    .clk_50m(clk_50m),
    .Rxclk_en(Rxclk_en),
    .Txclk_en(Txclk_en)
  );

  // Clock generation
  always #10 clk_50m = ~clk_50m;

  // Stimulus
  initial begin
    clk_50m = 0;

    // Add reset here if necessary

    #1000 $finish; // Finish simulation after 1000 time units
  end

  // Monitor
  always @(posedge clk_50m)
  begin
    if (Rxclk_en == 1)
        begin
            $display("Rxclk_en: %b, Txclk_en: %b", Rxclk_en, Txclk_en);
        end
  end

endmodule
```

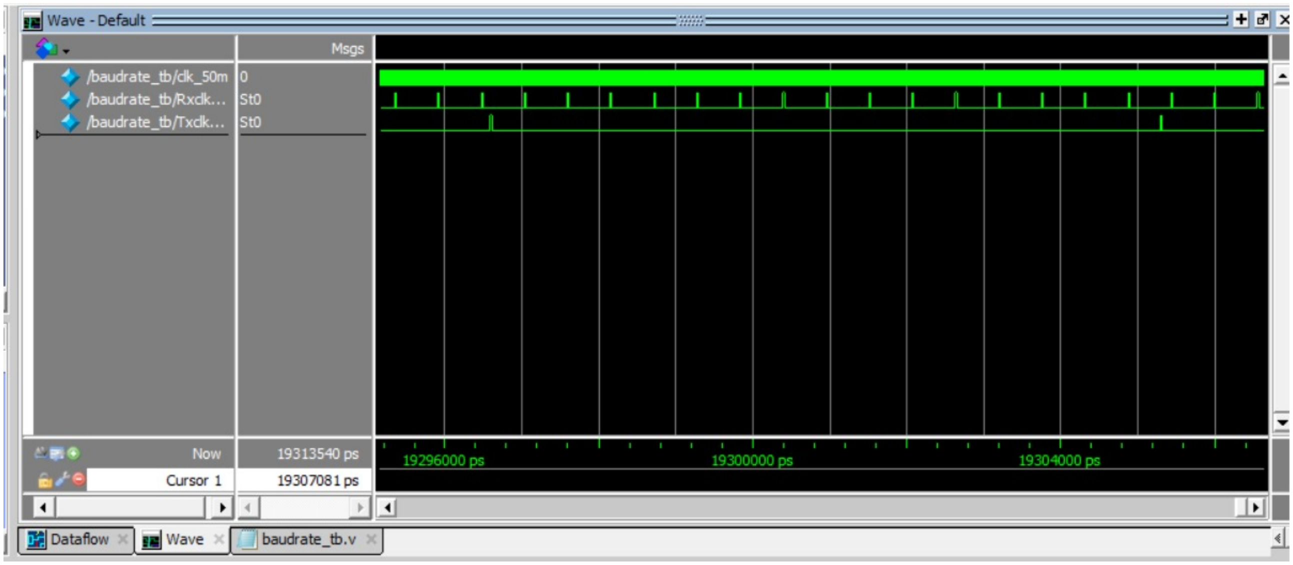


Figure 2.1: Baudrate Waveform

2.2 Testbench for UART Implementation - testbench.v

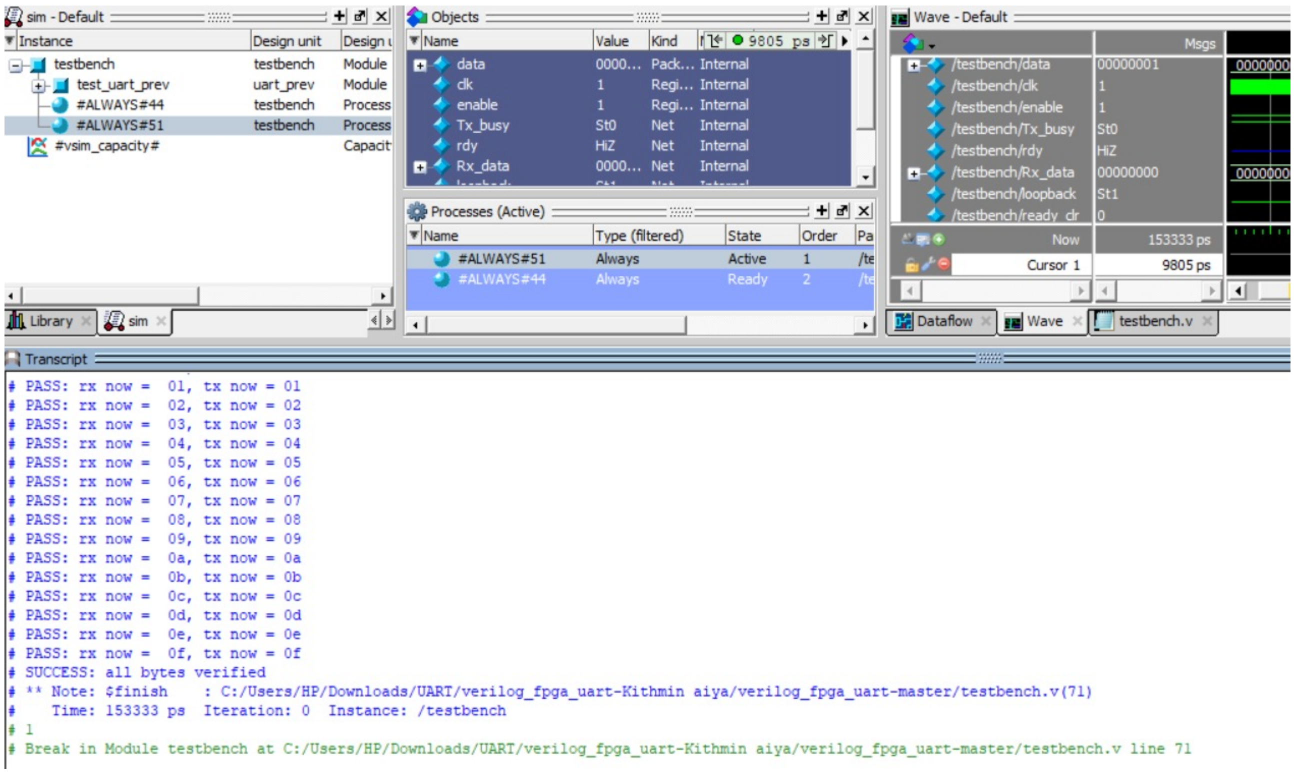
```
    //This is a simple testbench for UART Tx and Rx.
//The Tx and Rx pins have been connected together creating a serial loopback.
//We check if we receive what we have transmitted by sending incremeting data bytes.

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

module testbench();

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

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

wire loopback;
reg ready_clr = 0;

uart_prev test_uart_prev(.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, testbench);
    enable <= 1'b1;
    #2 enable <= 1'b0;
end



always
begin
    #1 clk = ~clk;
end
```

```



always @(posedge ready)
begin

    #2 ready_clr <= 1;
    #2 ready_clr <= 0;

    if (Rx_data == data)
    begin
        $display("PASS: rx now =  %x, tx now = %x", Rx_data, data);
    end

    if (Rx_data != data)
    begin
        $display("FAIL: rx now =  %x, tx now = %x", Rx_data, data);
    end


    if (Rx_data == 8'b00001111)
    begin
        $display("SUCCESS: all bytes verified");
        $finish;
    end

    data <= data + 1'b1;
    enable <= 1'b1;
    #2 enable <= 1'b0;


end
endmodule
```

Chapter 3

Simulation Results and Timing Diagram

3.1 Simulation Results



Instance	Design unit	Design u
testbench	testbench	Module
test_uart_prev	uart_prev	Module
#ALWAYS#44	testbench	Process
#ALWAYS#51	testbench	Process
#vsim_capacity#		Capacit

Name	Value	Kind	
data	0000...	Pack...	Internal
clk	1	Regi...	Internal
enable	1	Regi...	Internal
Tx_busy	St0	Net	Internal
rdy	HiZ	Net	Internal
Rx_data	0000...	Net	Internal

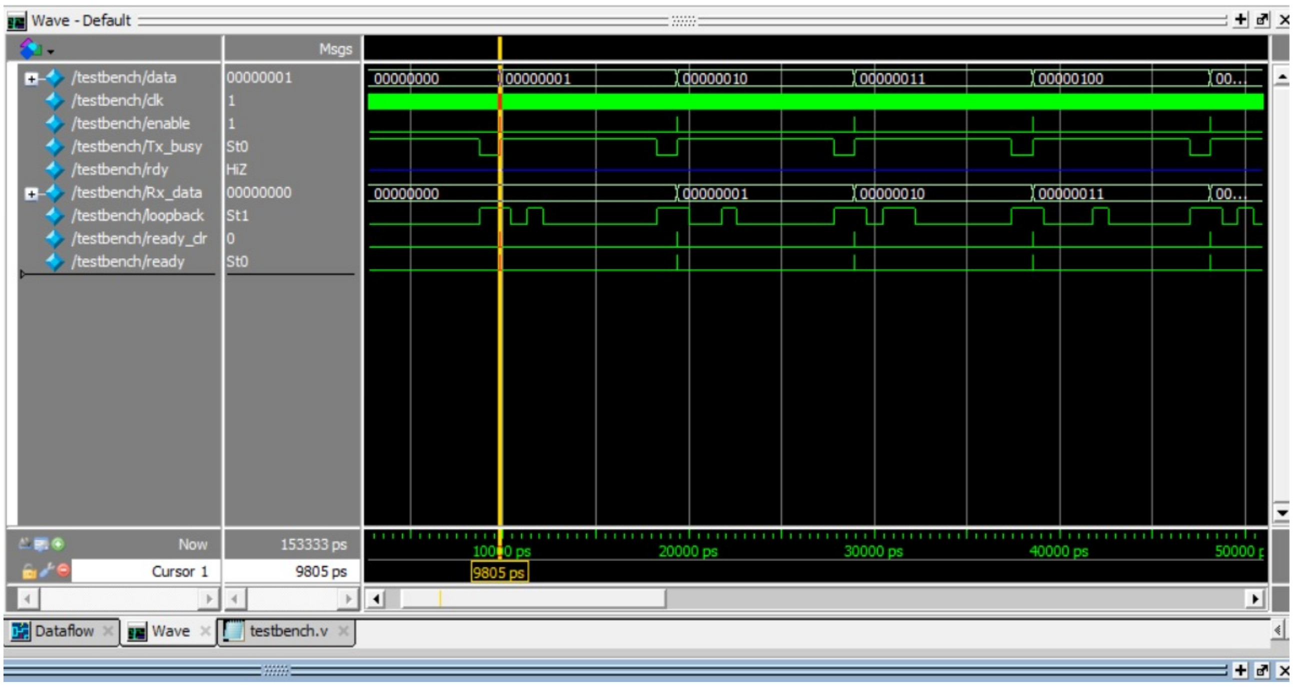
Name	Type (filtered)	State	Order	Pa
#ALWAYS#51	Always	Active	1	/te
#ALWAYS#44	Always	Ready	2	/te

	Msgs
/testbench/data	00000001
/testbench/clk	1
/testbench/enable	1
/testbench/Tx_busy	St0
/testbench/rdy	HiZ
/testbench/Rx_data	00000000
/testbench/loopback	St1
/testbench/ready_clr	0
Now	153333 ps
Cursor 1	9805 ps

```
# PASS: rx now =  01, tx now = 01
# PASS: rx now =  02, tx now = 02
# PASS: rx now =  03, tx now = 03
# PASS: rx now =  04, tx now = 04
# PASS: rx now =  05, tx now = 05
# PASS: rx now =  06, tx now = 06
# PASS: rx now =  07, tx now = 07
# PASS: rx now =  08, tx now = 08
# PASS: rx now =  09, tx now = 09
# PASS: rx now =  0a, tx now = 0a
# PASS: rx now =  0b, tx now = 0b
# PASS: rx now =  0c, tx now = 0c
# PASS: rx now =  0d, tx now = 0d
# PASS: rx now =  0e, tx now = 0e
# PASS: rx now =  0f, tx now = 0f
# SUCCESS: all bytes verified
# ** Note: $finish    : C:/Users/HP/Downloads/UART/verilog_fpga_uart-Kithmin aiya/verilog_fpga_uart-master/testbench.v(71)
#    Time: 153333 ps  Iteration: 0  Instance: /testbench
# 1
# Break in Module testbench at C:/Users/HP/Downloads/UART/verilog_fpga_uart-Kithmin aiya/verilog_fpga_uart-master/testbench.v line 71
```

Figure 3.1: Simulation Results

3.2 Timing Diagram



```
Wave - Default
                         Msgs
/testbench/data          00000001
/testbench/clk           1
/testbench/enable        1
/testbench/Tx_busy       St0
/testbench/rdy           HiZ
/testbench/Rx_data       00000000
/testbench/loopback      St1
/testbench/ready_clr     0
/testbench/ready         St0

Now                      153333 ps
Cursor 1                 9805 ps
```

Figure 3.2: Timing Diagram