Department of Electronic & Telecommunication Engineering

University of Moratuwa

EN2111 Electronic Circuit Design

Design UART Implementation in FPGA

Group 38

Name	Index
W. A. D. N. M. Wanniarachchi	210679B
T. R. Wathudura	210682D
M. P. Wickramarathne	210703V

This is submitted as a partial fulfillment for the module.

May 8, 2024

Contents

1. Introduction

Universal Asynchronous Receiver Transmitters (UARTs) are widely used communication protocols for serial communication between electronic devices. They are popular due to their simplicity and ease of implementation. In this project, we will implement a UART transceiver system on two separate Field Programmable Gate Arrays (FPGAs) on a DE0-Nano board.

UARTs facilitate asynchronous serial communication, meaning that the data being transmitted does not require a clock signal to synchronize the sender and receiver. This reduces the complexity of the design. UARTs transmit data one bit at a time over a single wire.

Our design will be described using Verilog hardware descriptive language (HDL) and will be simulated using ModelSim software. The DE0-Nano board will be used as the target platform for implementation using Quartus Prime software. This report will detail the design process, including the Verilog code for the UART transmitter and receiver modules, the co-simulation environment setup in ModelSim, and the results of the simulation.

2. Verilog RTL Code

(a) Top level module

```
1  module uart(input wire [7:0] data_in, //input data
2              input wire wr_en,
3              input wire clear,
4              input wire clk_50m,
5              output wire Tx,
6              output wire Tx_busy,
7              input wire Rx,
8              output wire ready,
9              input wire ready_clr,
10             output wire [7:0] data_out,
11             output [7:0] LEDR,
12             output wire Tx2//output data
13             );
14  assign LEDR = data_in;
15  assign Tx2 = Tx;
16  wire Txclk_en, Rxclk_en;
17 baudrate uart_baud(  .clk_50m(clk_50m),
18                     .Rxclk_en(Rxclk_en),
19                     .Txclk_en(Txclk_en)
20                     );
21 transmitter uart_Tx( .data_in(data_in),
22                     .wr_en(wr_en),
23                     .clk_50m(clk_50m),
24                     .clken(Txclk_en), //We assign Tx clock to enable clock
25                     .Tx(Tx),
26                     .Tx_busy(Tx_busy)
27                     );
```

```
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
```

(b) Transmitter

```
module transmitter(  input wire [7:0] data_in, //input data as an 8-bit regsiter/vector
                     input wire wr_en, //e nable 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

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 data
         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 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 transmitted from
            state <= TX_STATE_STOP; // when bit position has finally reached 7, assign state to stop transmi
         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
```

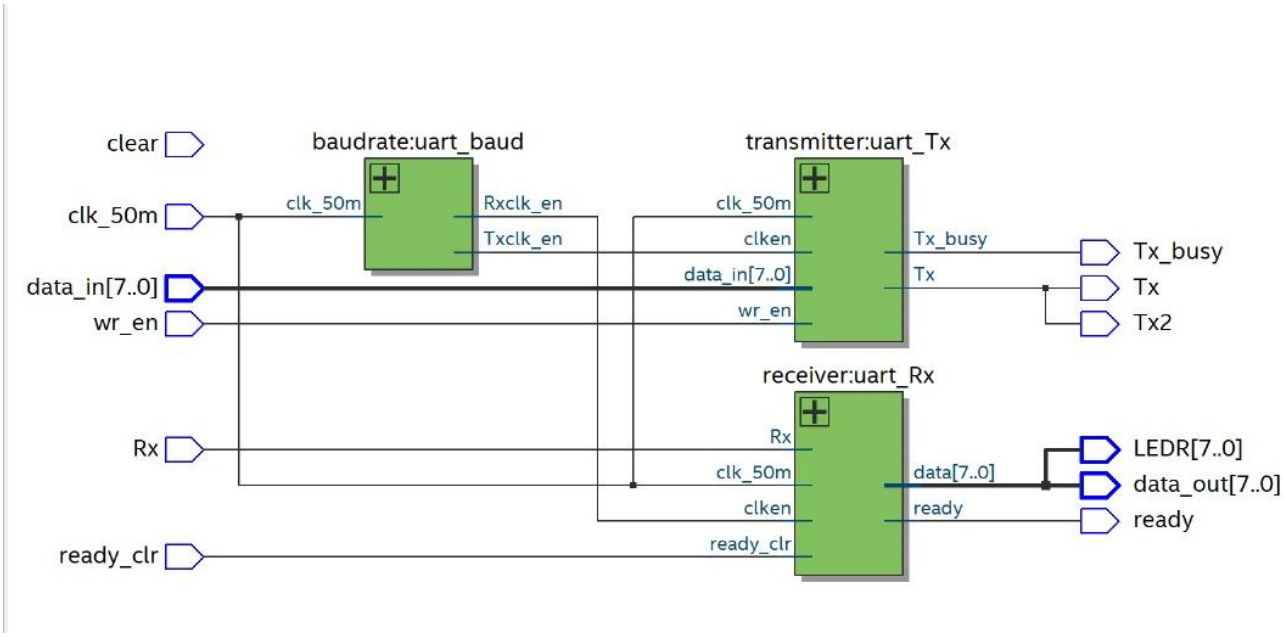
(c) Receiver

```
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 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 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 000
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: begin // We define condtions for starting the receiver
        if (!Rx || sample != 0) // start counting from the first low sample
           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 bits
           bit_pos <= 0;
           sample <= 0;
           scratch <= 0;
        end
     end
     RX_STATE_DATA: begin // We define conditions for starting the data colleting
        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 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 to START
     end
     endcase
  end
end

endmodule

```

(d) Baudrate

```
4  module baudrate  (input wire clk_50m,
5                    output wire Rxclk_en,
6                    output wire Txclk_en
7                    );
8  //Our Testbench uses a 50 MHz clock.
9  //Want to interface to 115200 baud UART for Tx/Rx pair
10 //Hence, 50000000 / 115200 = 435 Clocks Per Bit.
11 parameter RX_ACC_MAX = 50000000 / (115200 * 16);
12 parameter TX_ACC_MAX = 50000000 / 115200;
13 parameter RX_ACC_WIDTH = $clog2(RX_ACC_MAX);
14 parameter TX_ACC_WIDTH = $clog2(TX_ACC_MAX);
15 reg [RX_ACC_WIDTH - 1:0] rx_acc = 0;
16 reg [TX_ACC_WIDTH - 1:0] tx_acc = 0;
17
18 assign Rxclk_en = (rx_acc == 5'd0);
19 assign Txclk_en = (tx_acc == 9'd0);
20
21 always @(posedge clk_50m) begin
22    if (rx_acc == RX_ACC_MAX[RX_ACC_WIDTH - 1:0])
23       rx_acc <= 0;
24    else
25       rx_acc <= rx_acc + 5'b1; //increment by 00001
26 end
27
28 always @(posedge clk_50m) begin
29    if (tx_acc == TX_ACC_MAX[TX_ACC_WIDTH - 1:0])
30       tx_acc <= 0;
31    else
32       tx_acc <= tx_acc + 9'b1; //increment by 000000001
33 end
34
35 endmodule
36
```

3. RTL view



4. Testbench

```
1  //This is a simple testbench for UART Tx and Rx.
2  //The Tx and Rx pins have been connected together creating a serial loopback.
3  //We check if we receive what we have transmitted by sending incremeting data bytes.
4
5  //It sends out byte 0xAB over the transmitter
6  //It then exercises the receive by receiving byte 0x3F
7  //`include "uart.v"
8
9  module uart_TB();
10
11 reg [7:0] data = 0;
12 reg clk = 0;
13 reg enable = 0;
14
15 wire Tx_busy;
16 wire rdy;
17 wire [7:0] Rx_data;
18
19 wire loopback;
20 reg ready_clr = 0;
21
22 uart test_uart(.data_in(data),
23                .wr_en(enable),
24                .clk_50m(clk),
25                .Tx(loopback),
26                .Tx_busy(Tx_busy),
27                .Rx(loopback),
28                .ready(ready),
29                .ready_clr(ready_clr),
30                .data_out(Rx_data)
31                );
```

```
32 initial begin
33    $dumpfile("uart.vcd");
34    $dumpvars(0, uart_TB);
35    enable <= 1'b1;
36    #2 enable <= 1'b0;
37 end
38 always begin
39    #1 clk = ~clk;
40 end
41 always @(posedge ready) begin
42    #2 ready_clr <= 1;
43    #2 ready_clr <= 0;
44    if (Rx_data != data) begin
45       $display("FAIL: rx data %x does not match tx %x", Rx_data, data);
46       $finish;
47    end
48    else begin
49       if (Rx_data == 8'h2) begin //Check if received data is 11111111
50          $display("SUCCESS: all bytes verified");
51          $finish;
52       end
53       data <= data + 1'b1;
54       enable <= 1'b1;
55       #2 enable <= 1'b0;
56    end
57 end
58 endmodule
59
```

5. Simulation Results

6. FPGA implementation