UNIVERSITY OF LIVERPOOL

Department of Electrical and Electronics Engineering

Student ID: 201454022

Name: Vikas Anaokar

Programme: MSc Microelectronic Systems

Elec 473 Digital System Design

Assignment 1: Serial Communication

Contents

List of Figures

List of Equations

1. Introduction

Universal Asynchronous receiver transmitter (UART) is a serial data transfer protocol [1]. UART can transmit at speed ranging from 4.8Kbps to 115.2Kbps. UART consists of a transmitter and receiver working in tandem to transmit and receive data. Transmitter receives data bytes from microprocessor or micro-controller. The bits are packetized and transmitted in sequential fashion, i.e. 1 bit at a time. The transfer rate of these bits is known as the baud rate. Some standard baud rates are 4800 bps, 9600 bps up to 15.2 Kbps [1].

The timing diagram of a UART transmitter-receiver is shown below:

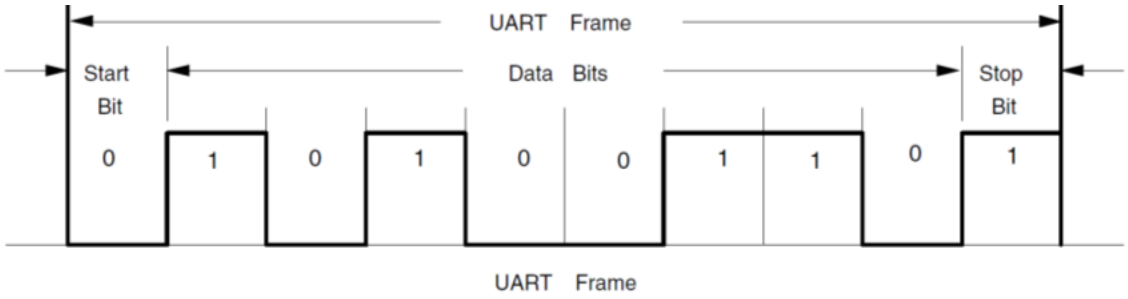

Figure 1 Timing diagram of UART [2]

UART packet contains LSB as the start bit, which is a 1->0 transition indicating the start of transmission. The next set of bits are the data bits received from the processor. The next bit is parity bit calculated from the data bit based on odd or even parity. The last bit or the MSB of the packet is a stop bit which is binary 1.

2. UART Transmitter

A general UART transmitter Block Diagram is shown below:

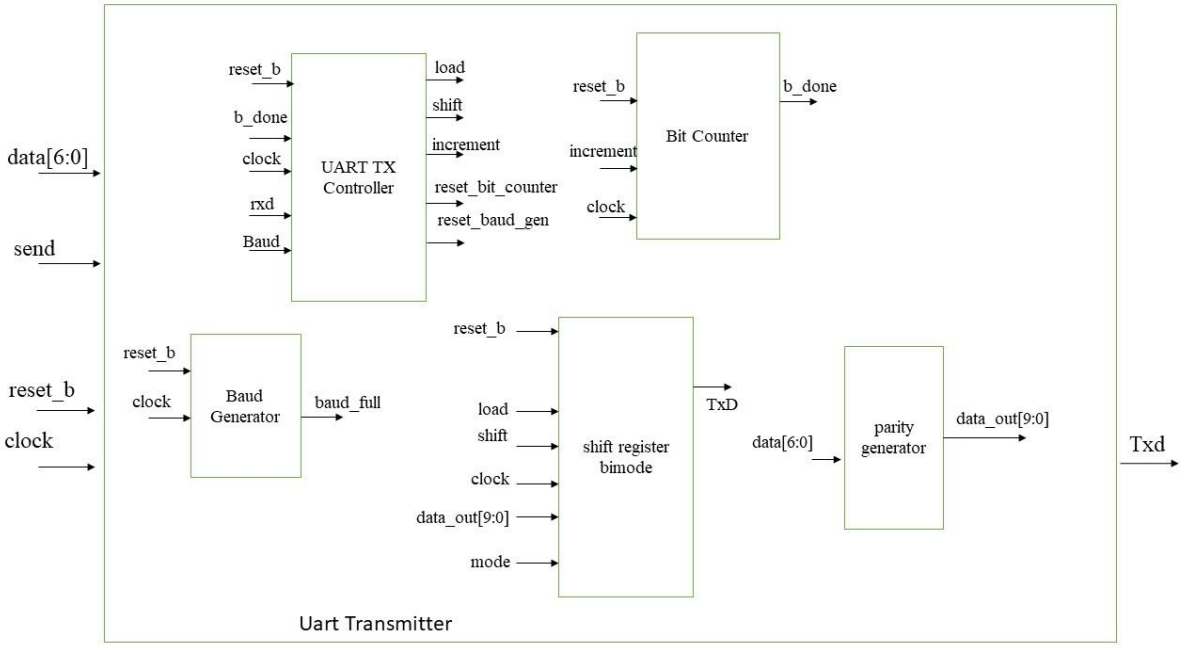

Figure 2 UART transmitter block diagram

The responsibility of the transmitter is to send data received on data pin on the TxD pin which is 1-bit signal according to the baud rate specified. The Transmitter takes the system clock, which is 50 Mhz for Altera DE2 Board. The reset_b is active low reset which will reset all the submodules as well as stop the transmission and send idle bits on the TxD line.

Data is 7-bit input given from the switches on the FPGA. Switches SW8 to SW2 is used for this purpose. Reset_b is also a switch on the FPGA. The choice was made reset to map to SW0. Clock is sourced from internal PLL mapped to pin PIN_N2 of FPGA.

The transmitter consists of 2 primary submodules. First one is architecture where the design choice is to have a simple architecture with modules which assist controller. This method reduces the complexity of the controller. These modules are bit counter, baud generator, parity generator and shift register. The second submodule called controller controls these modules. The controller is the central brain of the whole transmitter. It is responsible for the correct flow of data while maintaining adherence to the timing of the protocol.

Following are the input and output of the transmitter:

Figure 3 UART Transmitter input outputs

Verilog Code of UART transmitter with all blocks integrated is in Appendix 7.1.

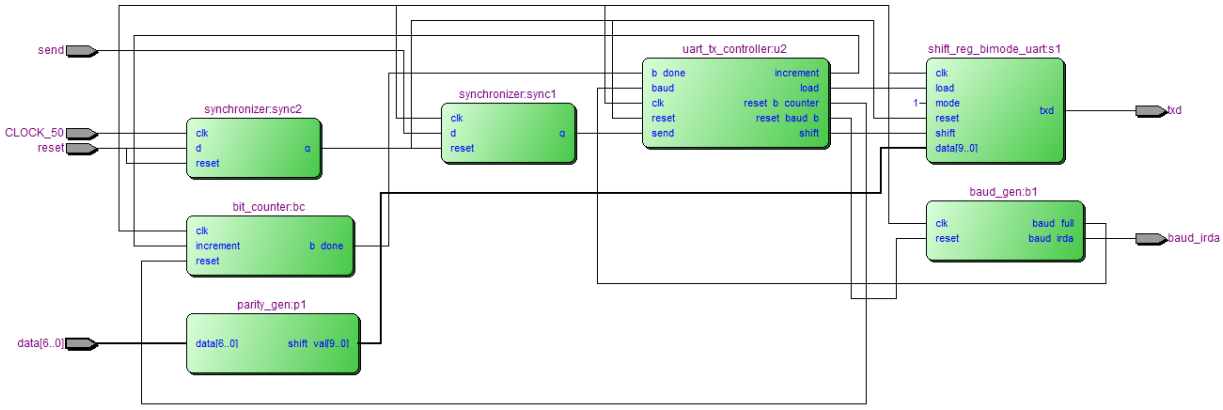

Figure 4 RTL Viewer of UART Transmitter

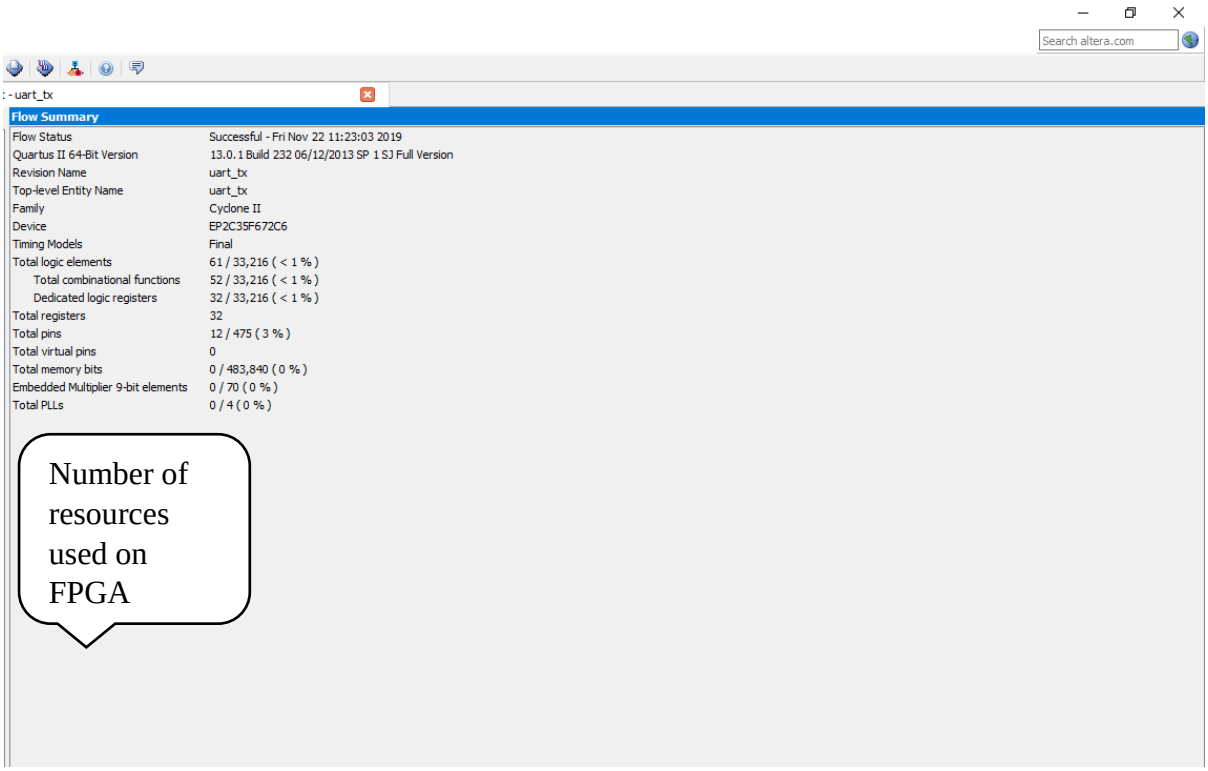

Figure 5 Compilation Report for UART Transmitter

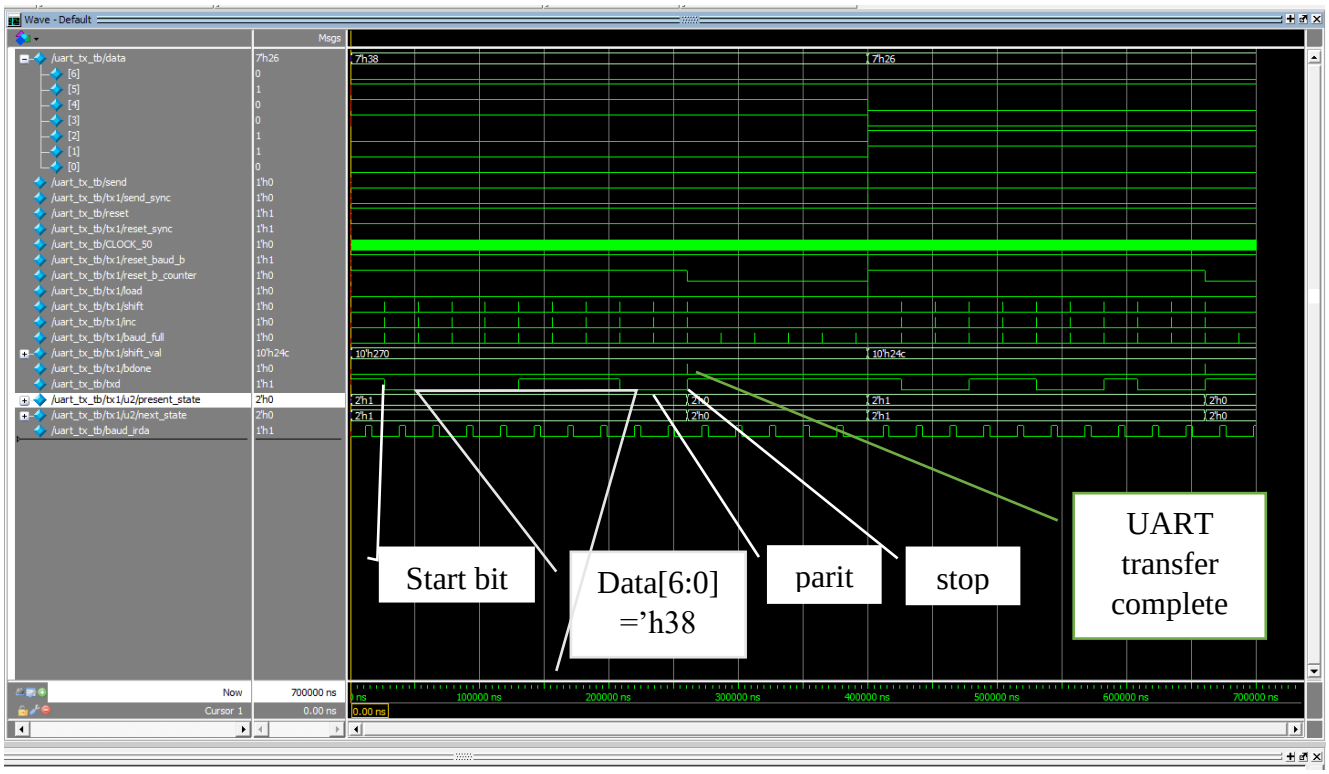

Figure 6 Simulation results of UART Transmitter

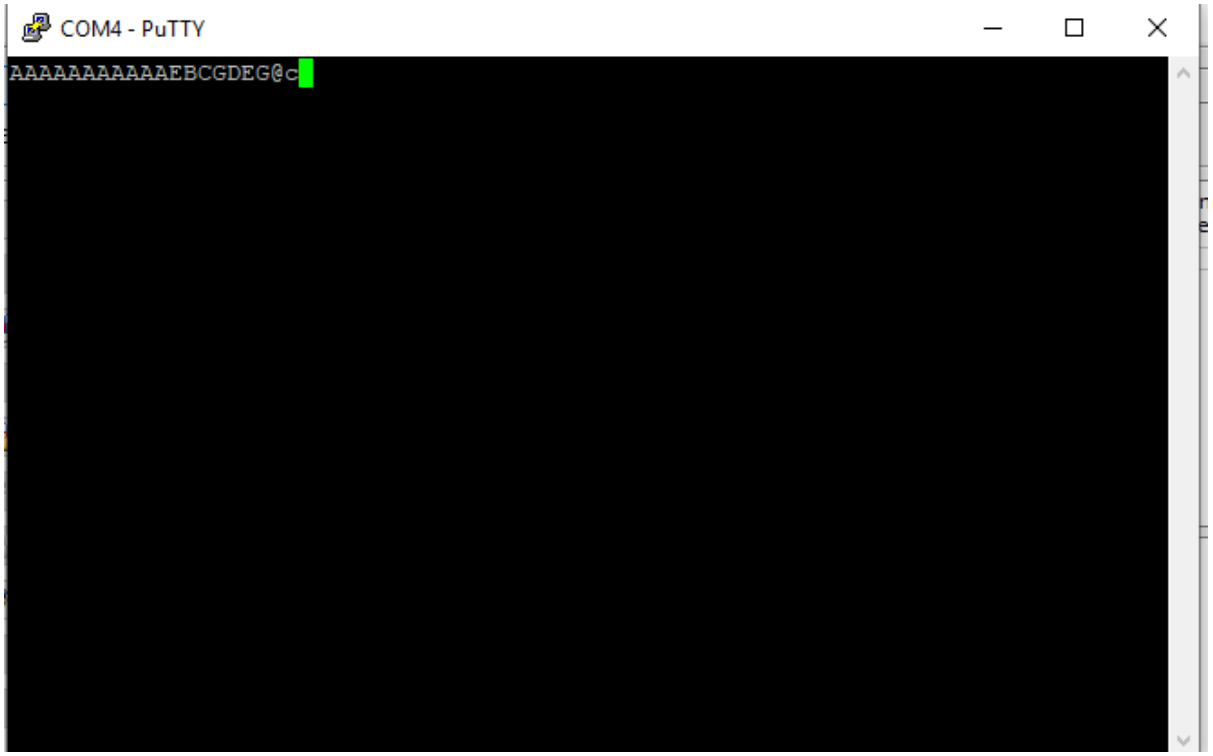

Figure 7 Putty output when FPGA is sending data using UART transmitter

It contains the following modules:

2.1 Bit counter

2.1.1 Specification

Count the number of bits in a UART packet. So, it is the controller that controls a sequential up counter circuit which runs on system 50 Mhz Clock and reset. The increment signal is input to the counter, which acts like a enable signal. When the signal is high, the counter increments by 1 to the next value. Once the value reaches the number of bits specified in the parameter, then the bit counter makes b_done signal high. It is the responsibility of the controller to reset the counter as well as control the increment input.

Additional Discussion: Bit counter is made reusable to be used inside the UART receiver. Also, if the specification changes, i.e. number of stop bits are made two or input data bits are changed from 7 to 8 in the transmitter, this bit counter shall work without any modifications in the design. This design decision to make it reusable. It was achieved using a parameterised module declaration.

2.1.2 Block Diagram

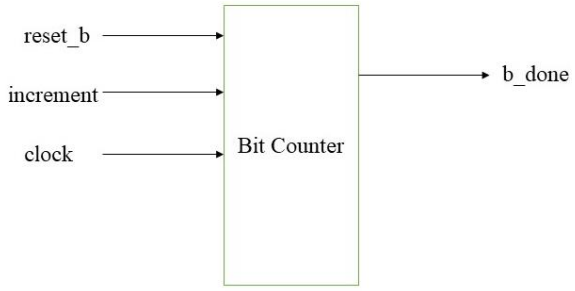

Figure 8 Bit counter Block Diagram

2.1.3 ASM Chart

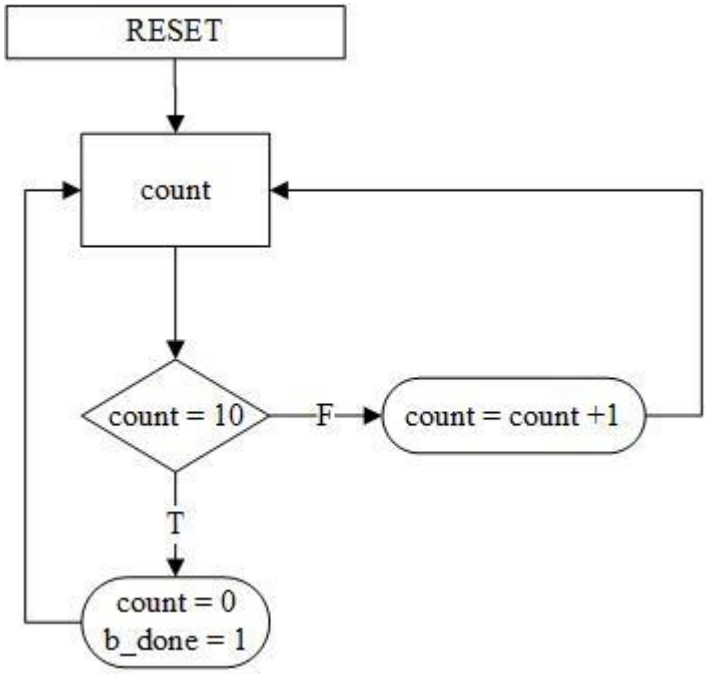

Figure 9 ASM chart for Bit counter

2.1.4 Verilog Code

```
module bit_counter #(parameter COUNT_VALUE=10)(b_done,increment,clk,reset);
        output reg b_done;
        input increment,clk,reset;
        reg [3:0]count,count_next; //internal variables to track the counter operation
        always@(posedge clk)
        begin
                if(reset == 1'b0) //active low synchronous reset
```

```
                begin
                        count = 4'd0; //make count=0 when reset is pressed
                end
        else
                begin
                count <= count_next; //put the count_next into count to create flip flops for counter

        end //else block ends
    end //always block ends
    always @(increment,count)
    begin
                count_next=count; //initialize the count_next with count value
                b_done = 1'b0; //by default b_done is 0
                if(count == COUNT_VALUE)
                        begin
                                b_done = 1'b1; //when counter value reaches COUNT_VALUE then make b_done 1 that means transmission/reception is over.
                        end
                        else
                        begin
                                if(increment == 1'b1)
                                begin
                                        count_next = count_next + 4'd1; //increment the counter by 1 if increment is high
                                end
                                else
                                begin
                                        count_next = count_next; //dont do anything if not incrementing, retain previous value.
                                end
                        end
```

end

endmodule

2.1.5 Simulation Result

In the beginning, the reset is applied to the bit counter. This reset is controlled by the controller. Increment is given at a fixed interval. After 10 increment intervals, b_done is 1 until reset is applied. Reset is active low and synchronous.

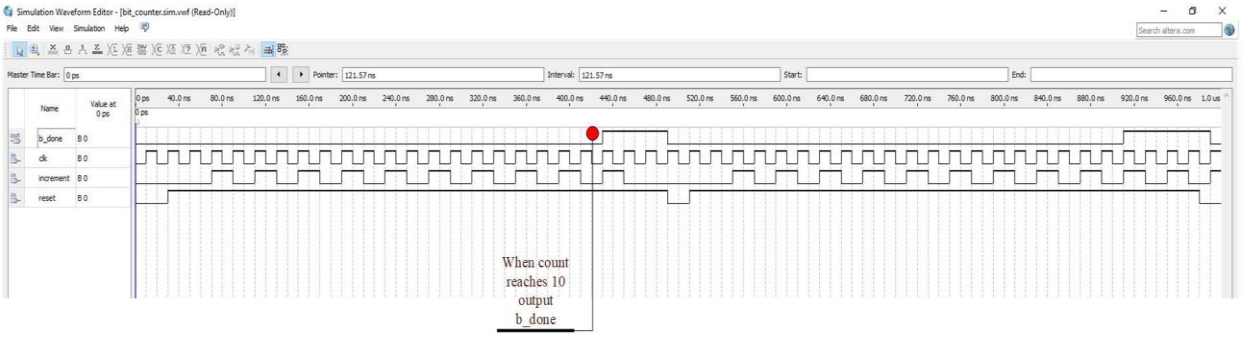

Figure 10 Simulation result of bit counter

2.1.6 RTL View

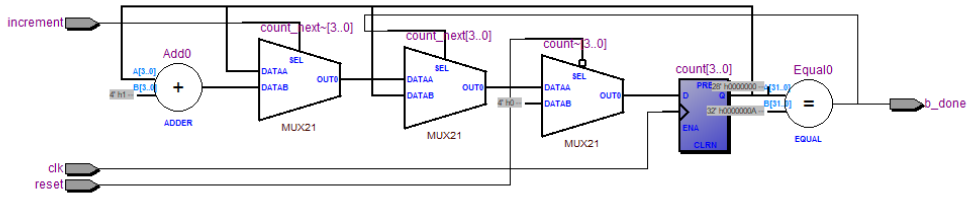

Figure 11 RTL View of Bit counter

2.2Shift Register

2.2.1 Specification

Shift register is needed as an architecture component for shifting the data bits serially. The shift register consists of flip-flops cascaded together in a chain-like structure. Between each flip-flop, there is a combinational logic which decides what data goes into the flip-flop [3].

2.2.2 Discussion and Design choice

Shift register was designed with two modes of operation selected by input “mode”:

2.2.2.1 Parallel In, Serial Out (PISO):

PISO mode is needed when instantiating the module in the UART transmitter.

In PISO mode, the shift register loads parallel data received from the parity generator when the load signal is high. Once data is loaded, the State machine waits for shift signal. One data bit is right-shifted only when the shift signal is high. The output can be seen on the output pin TxD which is 1 bit in size.

Shifting happens only on the active posedge of the clock, which is the system clock of 50 Mhz. The reset is connected to system reset reset_b, which is synchronous and active low in nature.

2.2.2.2 Serial In, Parallel Out (SIPO):

SIPO mode is needed when instantiating the module in UART receiver.

In SIPO mode, the shift register takes the input data serially from the rxd pin. Then, it is right shifts from MSB to LSB by 1-bit position. After shifting, the state machine waits for all shifts to happen. The load signal is input from the controller which decides if the data can be loaded in the output pin rx_data whose is length of UART packet.

Shifting happens only on the active posedge of the clock, which is a system clock of 50 Mhz. The reset is connected to system reset reset_b which is synchronous and active low in nature

The decision was to design the shift register to be modular. Hence, the data width was parameterised for input data. This decision allowed the flexibility to add extra stop bits or change the input data width.

Another critical design change is done to accommodate accidental shifts of 0's being transmitted on UART TX line. While shifting right, 0's is padded at the MSB side, instead of 0's, idle bits, i.e. 1's are padded. During reset, the shift register flip-flops are loaded with 1's and TxD is made 1. TxD is synchronised with the system clock to prevent any glitches or accidental transmission of 1's.

2.2.3 Block Diagram

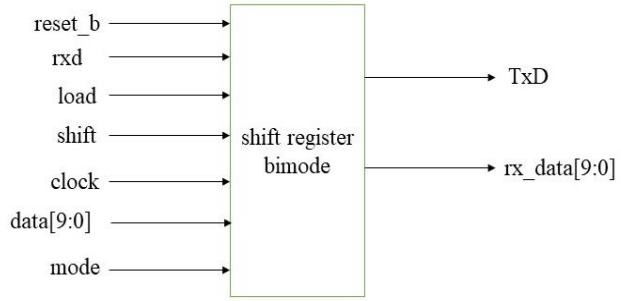

Figure 12 Block Diagram of Shift Register

2.2.4 ASM Chart

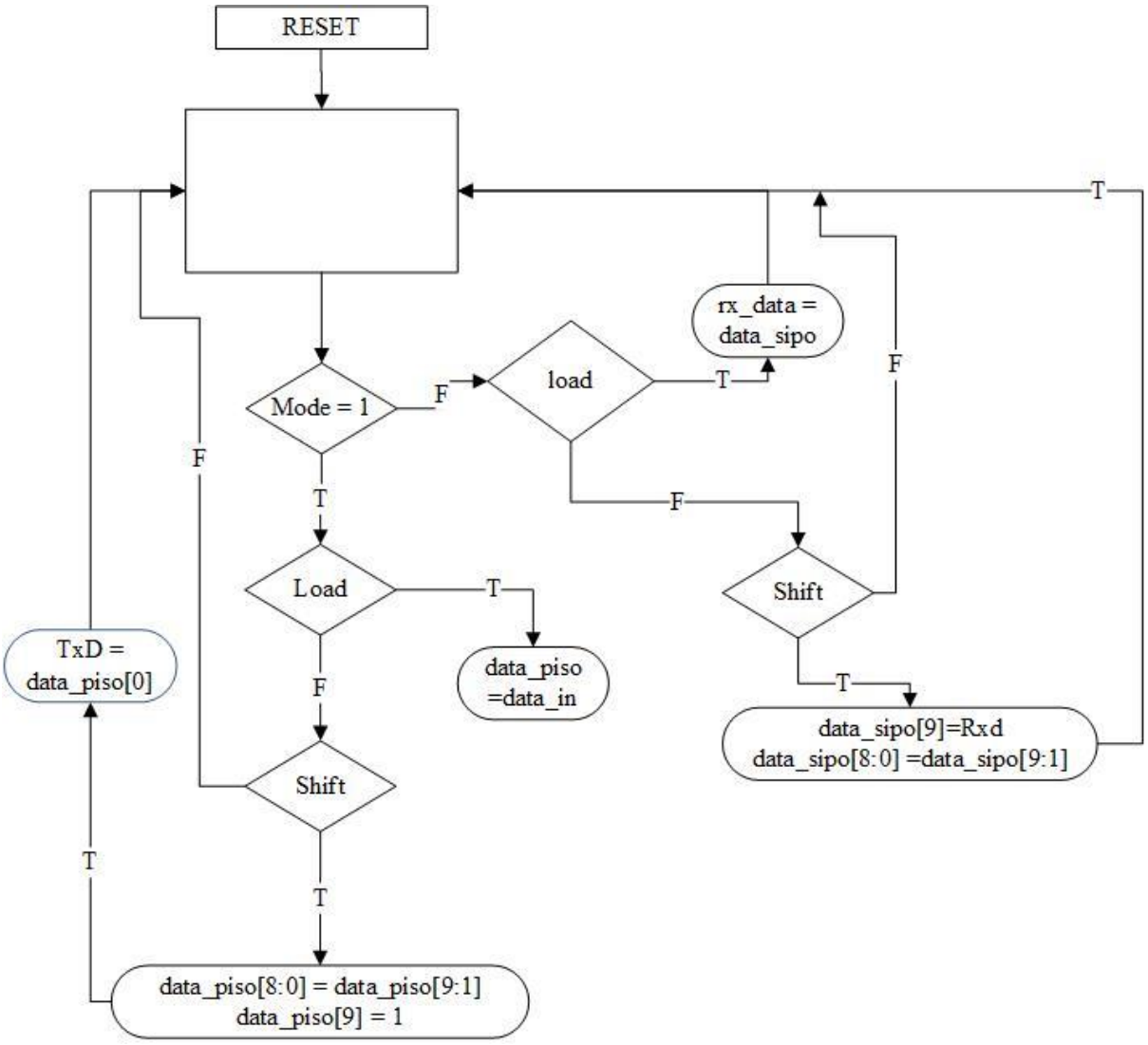

Figure 13 ASM Chart for Shift register

2.2.5 Verilog Code

```
module shift_reg_bimode_uart #(parameter SHIFT_DATA_WIDTH = 10)(txd,rx_data,load,shift,reset,clk,rxd,mode,data);

    output reg txd;

    output reg [SHIFT_DATA_WIDTH-1:0]rx_data;

    input load,shift,reset,clk,rxd,mode;

    input [SHIFT_DATA_WIDTH-1:0]data;

    reg [SHIFT_DATA_WIDTH-1:0]data_piso,data_sipo;

    localparam piso=1'b1,sipo=1'b0;


    always @(posedge clk)
```

```
begin
        rx_data=(load == 1'b1 && mode == sipo)?data_sipo: rx_data;
        if (reset == 1'b0)
        begin
                data_piso <= {SHIFT_DATA_WIDTH{1'b1}}; //make sure idle/stop bits are loaded on output line to avoid unexpected start of packet
                data_sipo <= {SHIFT_DATA_WIDTH{1'b1}};
                txd =1'b1; //make sure idle/stop bits are loaded on output line to avoid unexpected start of packet
        end
        else
begin
        case (mode)
        piso: //PISO when MODE =1
        begin
                if (load)
                begin
                        data_piso <= data; //parallel load of the data
                        txd <=1'b1; //make sure idle/stop bits are loaded on output line to avoid unexpected start of packet when loading data.
                end
                else if (shift)
                begin
                        data_piso[SHIFT_DATA_WIDTH-2:0]<=data_piso[SHIFT_DATA_WIDTH-1:1]; //left shift operation i.e msb<--lsb by 1 bit.
                        data_piso[SHIFT_DATA_WIDTH-1] <= 1'b1; //make sure idle/stop bits are padded instead of 0. it may cause unexpect start of packet.
                        txd <= data_piso[0]; //put output as the lsb of data to be transmitted.
                end
                else
                begin
```

```
                data_piso<=data_piso;
            end
        end
        sipo://SIPO when MODE =0
        begin

            if(shift)
            begin
                data_sipo[9]<=rxd;
                data_sipo[8:0]<= data_sipo[9:1]; //right shift data by 1 bit i.e msb --> lsb by 1 bit
            end
            else
            begin
                data_sipo<=data_sipo; //dont do anything if shift is 0 retain old value.
            end
        end
        default:
        begin
            data_piso <= 1'bx; //dont care condition when neither mode is selected
            data_sipo <= 1'bx; //dont care condition when neither mode is selected
        end
        endcase
end
end
endmodule
```

2.2.6 Simulation Result

In PISO Mode, reset is applied in the beginning. txd output is idle as it is synchronized with a clock, the output becomes 1 only when it receives a positive edge of the clock. After the reset is removed, shifts are given, and 10-bit data is provided, the shifted value is seen on the output of txd.

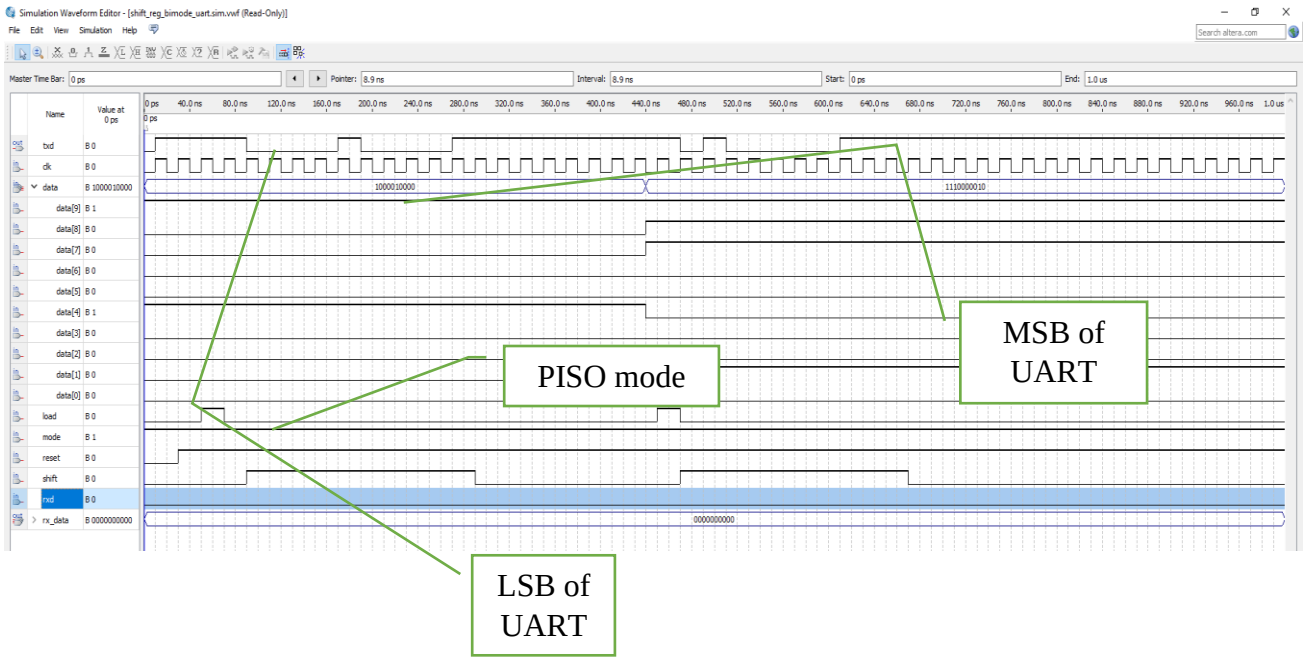

Figure 14 Simulation result of Shift register in PISO mode

In SIPO mode, at the beginning, the system is reset, then the rxd input is driven by the serial data. At the end of data transmission, the load signal is given while the shift is made 0. When load is received, the shift register outputs the parallel data available on the data_out.

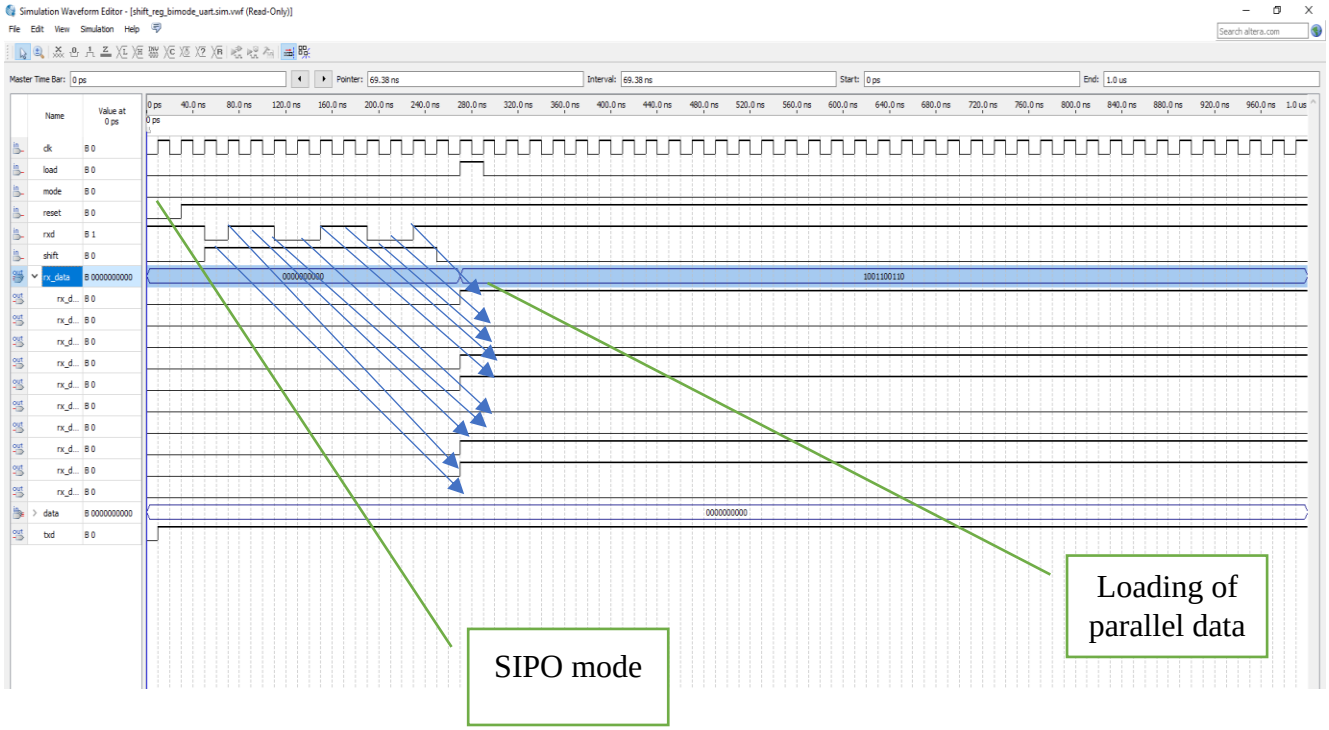

Figure 15 Simulation results of Shift register in SIPO mode

2.2.7 RTL View

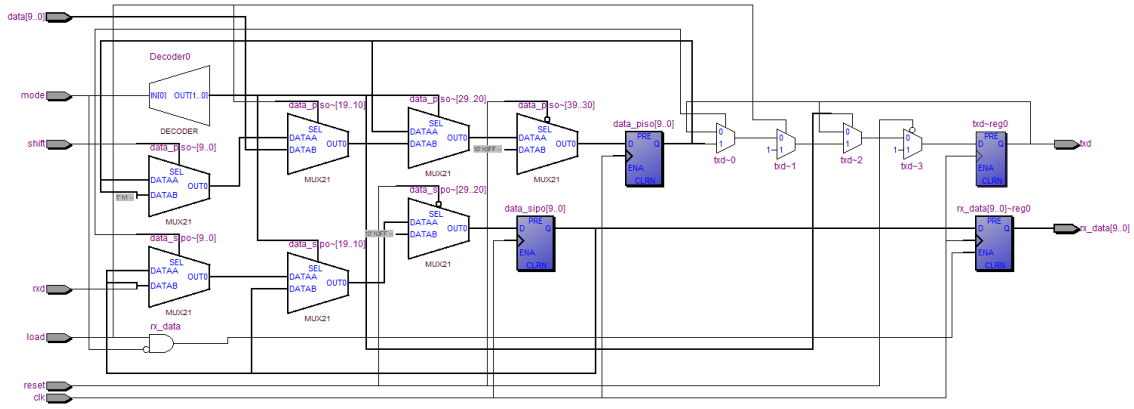

Figure 16 RTL view of Shift register

2.3Parity Generator

2.3.1 Specification

Parity generator is a combinational circuit which contains two sub logics:

1. Parity calculation: As odd parity is required; parity is calculated by taking an XNOR of the input data [4]. Parity generation equation is given by Equation 1, where n is the size of data.

$$parity = data[n-1] \odot data[n-2] \odot data[n-3] \ldots \odot data[0]$$

Equation 1 Parity Generation

2. UART Packet formation: Calculated parity bit is concatenated with input data bits. Along with that, start bit with value 0 is added at LSB while stop bit with value one is added at MSB.

The output of parity generator is given to shift register. Principle of modularity is maintained by passing data width as the parameterised inside module declaration.

2.3.2 Block Diagram

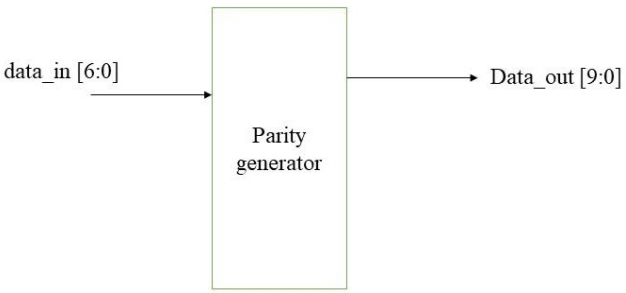

Figure 17 Block diagram of the parity generator

2.3.3 ASM Chart

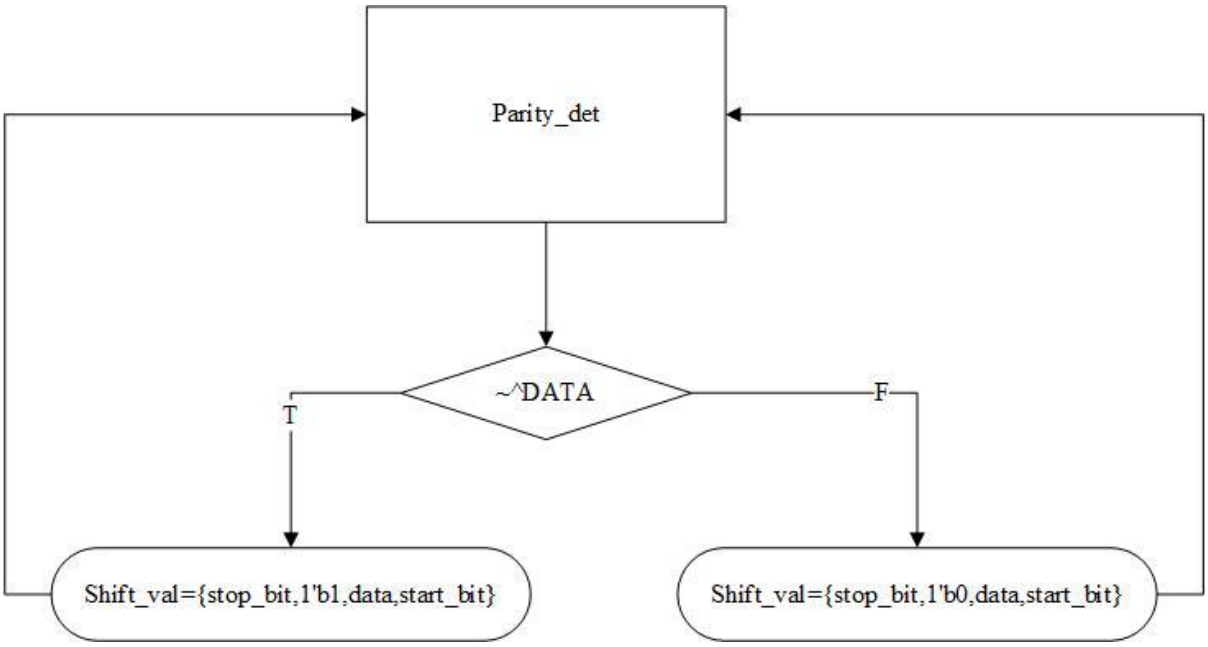

Figure 18 ASM chart for parity generator

2.3.4 Verilog Code

```
module parity_gen #(parameter DATA_WIDTH =7)(shift_val,data);

        output [DATA_WIDTH+2:0]shift_val;

        input [DATA_WIDTH-1:0]data;

        wire parity,start_bit,stop_bit;


        assign parity = ~^data; //odd parity calculation -> xnor operation


        assign start_bit = 1'b0; //pad the start bit

        assign stop_bit = 1'b1; //pad the stop bit

        assign shift_val = {stop_bit,parity,data,start_bit}; //concatenation step to form UART
packet

endmodule
```

2.3.5 Simulation Result

Input data is given randomly at random intervals. The parity is calculated and annotated in the simulation result. For eg., in the first set of data, there are 4 number of 1's. For odd parity, the parity bit is 1. For the second set of data, it is 0 and so on. After parity is calculated, output data is complete UART packet with MSB containing stop bit and LSB as start bit padded.

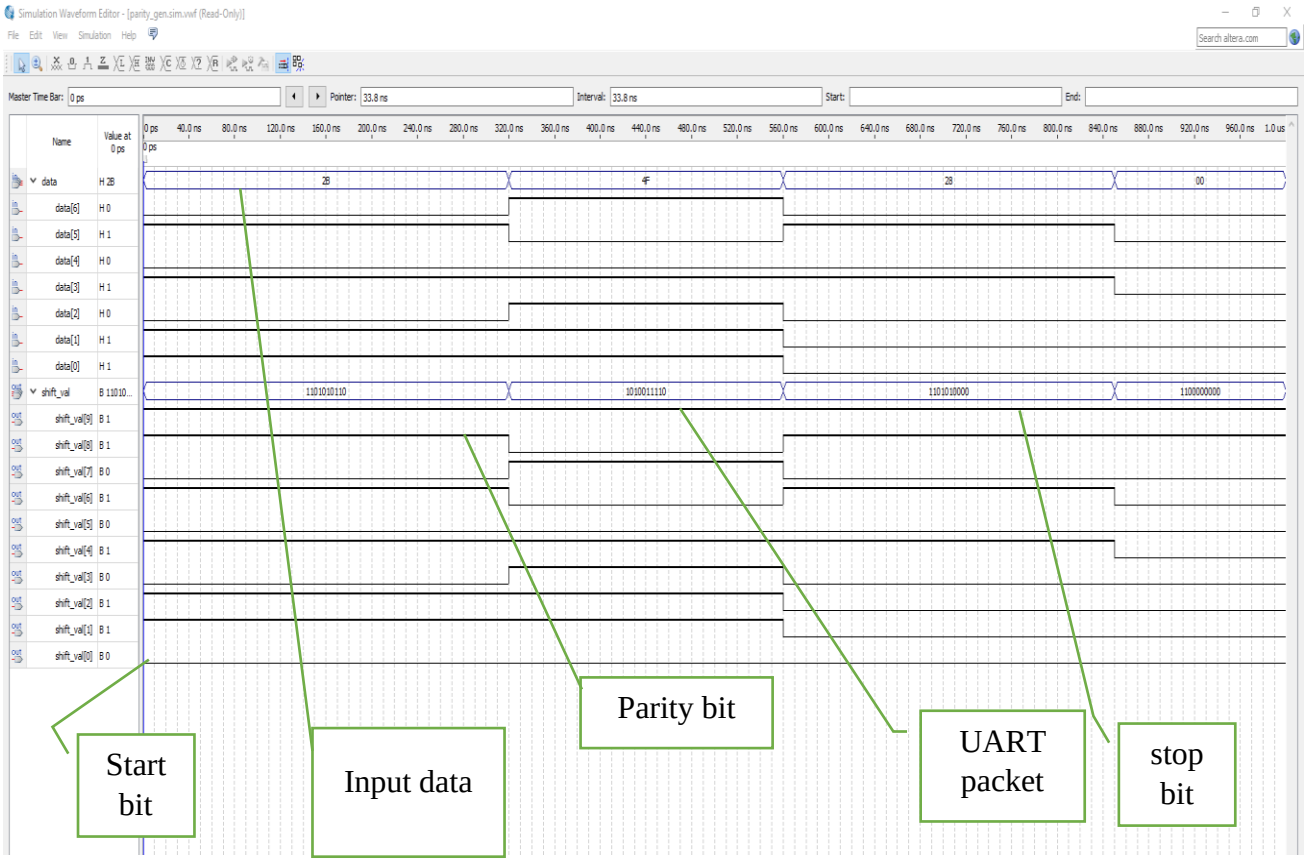

Figure 19 Simulation result of parity generator

2.3.6 RTL View

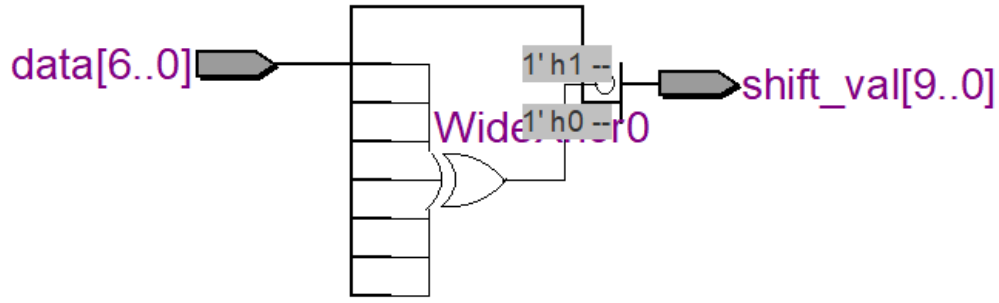

Figure 20 RTL View of Parity generator

2.4Baud Generator

2.4.1 Specification

The baud generator is a counter which counts the number of clock cycles. Equation 2 determines the counter end count value.

$$number\ of\ clocks\ to\ be\ counted = \frac{frequency\ of\ clock}{baud\ rate}$$

Equation 2 Baud count calculation

As per the calculation in Equation 3 and approximating to the nearest decimal place, the count value is 1302.

$$number\ of\ clocks\ to\ be\ counted = \frac{50 \times 10^6}{38400} = 1302$$

Equation 3 calculation of baud count for baud rate of 38400 bps

The baud generator will count up to 1302 clock cycles and make the baud signal high to the controller indicating the baud time. It reset to 0 on next clock cycle and repeats.

Another additional feature of the baud generator is that it outputs binary 1 on baud_half when it reaches half the count, i.e. when the count reaches 651. For IrDA modulation, baud_irda is output pulse of value 1 whose pulse width is 3/16 of baud. So, baud generator outputs 1 when the counter value is between 570 and 813.

2.4.2 Block Diagram

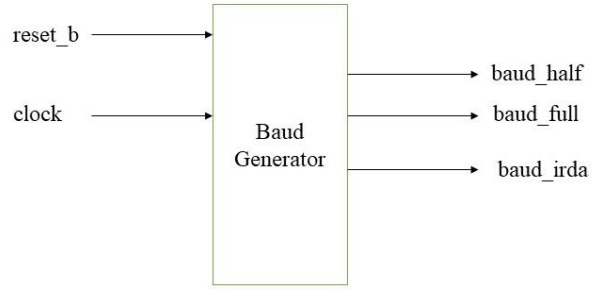

Figure 21 Block diagram of the baud generator

2.4.3 ASM Chart

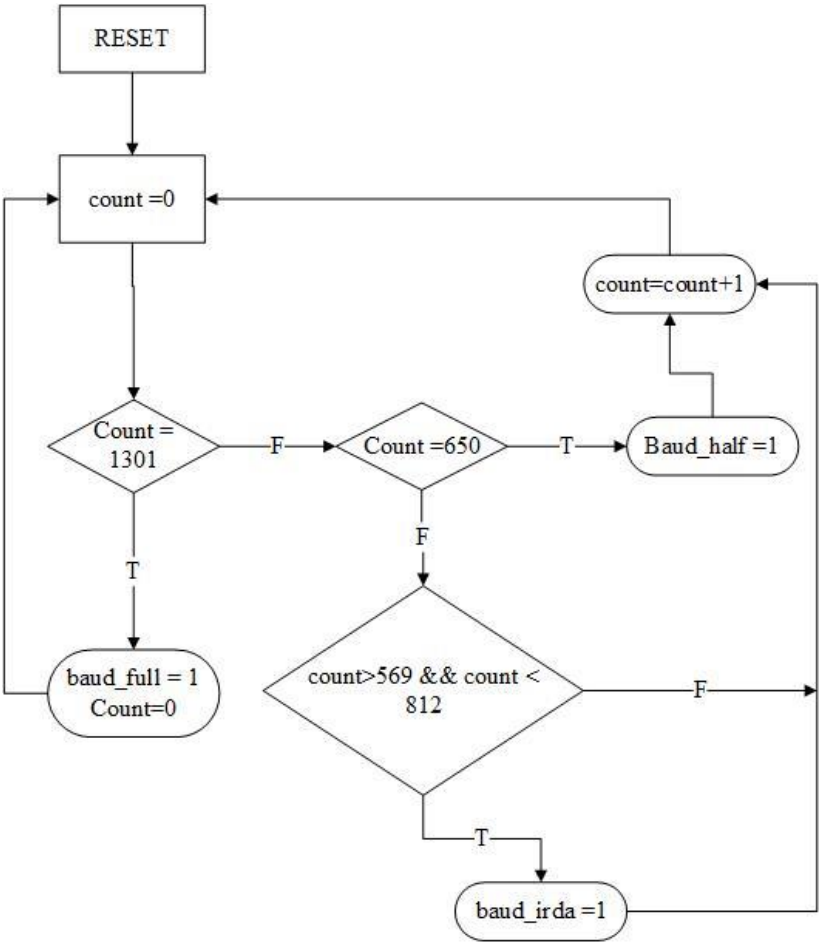

Figure 22 ASM chart for baud generator

2.4.4 Verilog Code

```
module baud_gen(baud_half,baud_full,baud_irda,clk,reset);
        output reg baud_full,baud_half,baud_irda;
        input clk,reset;
        reg [10:0]count,count_next;
        always@(posedge clk)
                begin
                        if(reset == 1'b0) //active low synchronous reset
                                begin
                                        count = 10'd0;//make count=0 when reset is pressed
                                end
                        else
                                begin
```

```
                count = count_next;//put the count_next into count to create flip flops for counter
                end
        end
        always@(count)
        begin
                count_next = count;//initialize the count_next with count value
                baud_full = 1'b0; //initialize baud_full
                baud_half =1'b0;//initialize baud_half
                baud_irda =1'b0;//initialize baud_irda
                if(count == 11'd1301)
                begin
                        baud_full = 1'b1; //make baud_full only if full baud rate of 38400 is reached.
                        count_next =0; //make count_next =0 for next packet
                end

                else
                        count_next=count + 11'd1; // increment the counter if the value doesnt reach the required baud count
                if(count == 11'd650)
                        baud_half = 1'b1; //baud_half is one when counter reaches half baud count value
                if(count >= 569 && count <=812) //to generate irda pulse of at the edge of 7/16 valid for 3/16 baud period and then 0 for rest 6/16 cycle.
                begin
                        baud_irda =1'b1; //keep baud_irda 1 for 3/16 baud period
                end
        end
endmodule
```

2.4.5 Simulation Result

In the beginning for a short duration, reset_b is given at around 10.24us to reset the baud generator. The period of clk is 20 ns. So, when baud generator reaches the count of 1302 clock

cycles or after 26.040 us period, baud_full output is 1. Baud_half output is 1 after 13.020 us when reset transition from 0 to 1. Baud_irda is 1 for a period of 3/16 of baud. The baud generator resets when baud_full is 1. The internal counter starts counting and keeps output in a repeated manner until reset is made 0.

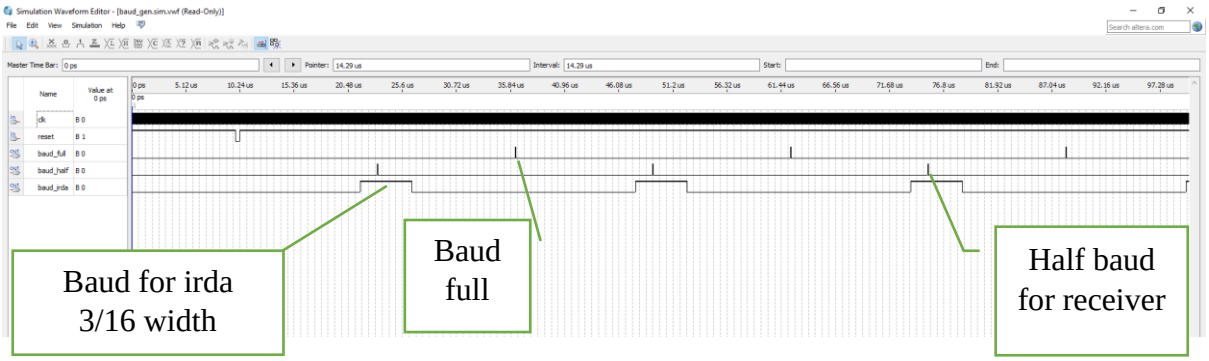

Figure 23 Simulation Result of Baud generator

2.4.6 RTL View

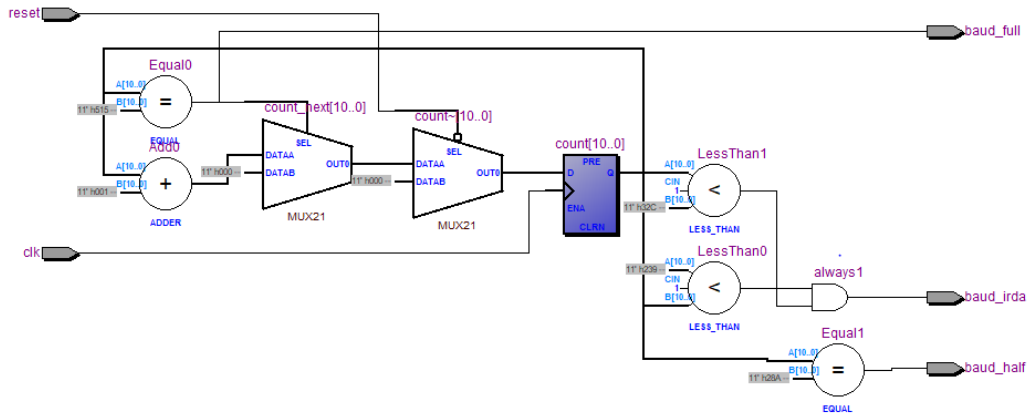

Figure 24 RTL View of baud generator

2.5UART Transmitter controller

2.5.1 Specification

Controller is the brain of the system. It is responsible for issuing the control signal to all the modules like shift register, bit counter and baud generator. It uses the control signals based on a state machine.

In the first state (reset_st), when the user presses the send key, the controller must reset the baud generator, bit counter, make load signal high to load the shift register with UART packet and then transition to a next state (load_st).

In this state, controller checks if all bits are sent by polling the b_done signal. If b_done is not high then it waits for baud generator to issue baud pulse when it receives the baud, the controller issues a shift command to shift register. It will also increment the bit counter. Next state is the same load_st state until all the bits are transferred serially. The b_done is high and then the state transitions to next_frame state.

In next_frame state, send signal is checked, i.e. it checks if the user has pressed and hold the send key at that instant. If the user has hold send then wait for the user to release the key and

transition to reset_st state for the next data packet. If the user has not held the key, then wait for the next frame and keep TxD line idle.

2.5.2 Design Choice and Discussion

A small design change is done to eliminate sending extra idle bit before transmission of the packet. This design is achieved by resetting the baud generator in reset_st state. This design also makes shift register input and parity generator output smaller by 1 bit. Controller will work for any specification changes in data width or number of stop bits.

2.5.3 Block Diagram

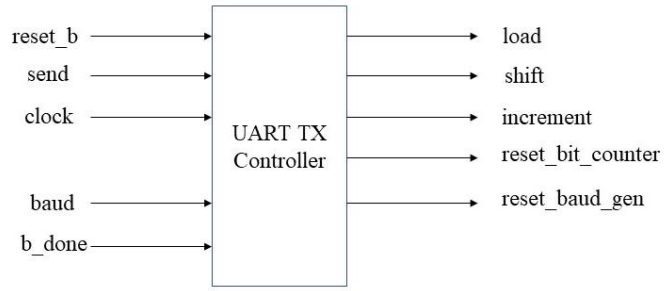

Figure 25 Block diagram of UART TX controller

2.5.4 ASM Chart

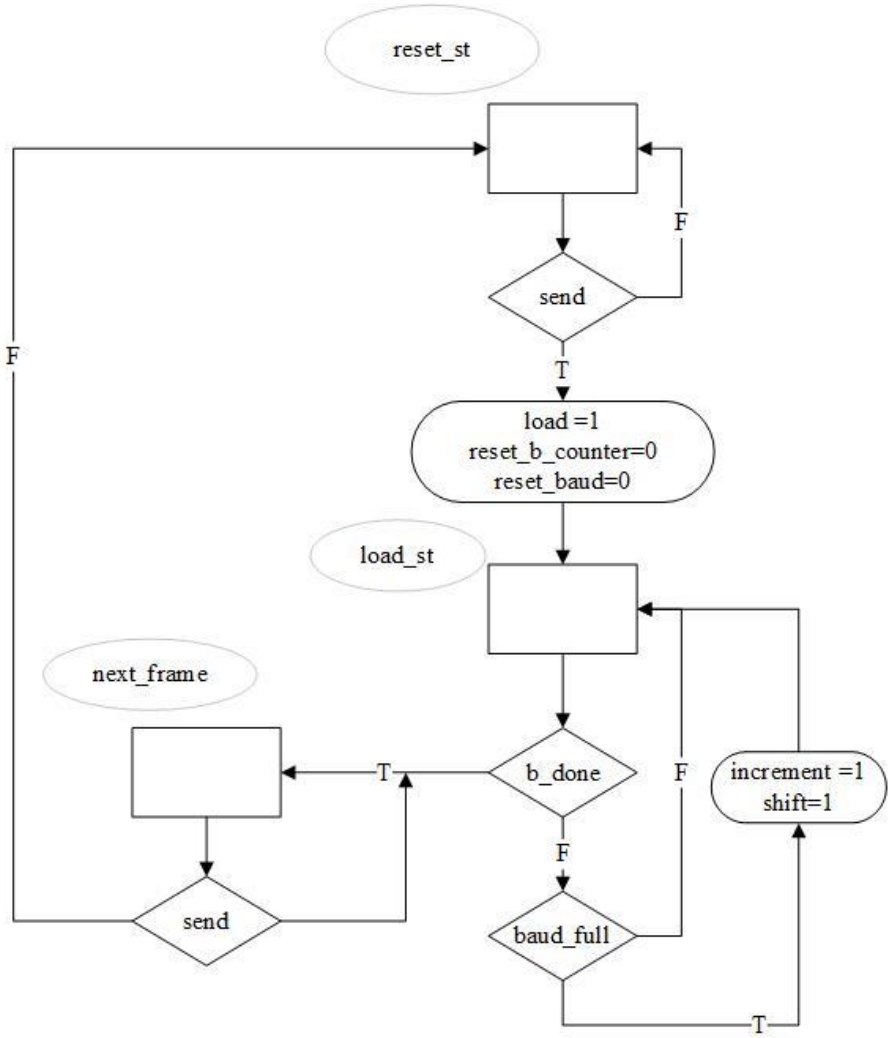

Figure 26 ASM chart for UART TX Controller

2.5.5 Verilog Code

```
module
uart_tx_controller(increment,load,shift,reset_b_counter,reset_baud_b,send,baud,b_done,clk,reset);

    output reg increment,load,shift,reset_b_counter,reset_baud_b;

    input baud,b_done,clk,reset,send;


    parameter reset_st = 2'b00,load_sr=2'b01,next_frame=2'b10; //total 3 states

    //declare state variables

    reg [1:0]next_state;

    reg [1:0]present_state;
```

```
always@(posedge clk)
begin
	if(reset == 1'b0) //synchronous active low reset
	begin
		present_state <= reset_st; //go to reset state when system is reseted
	end
	else
	begin
		present_state <= next_state; //if not reset then next state is assigned to
present_state
	end
end

always@(present_state,baud,b_done,send)
begin
	reset_baud_b=1'b1; //initalize baud reset
	case (present_state)
	reset_st:
	begin
		if(send == 1'b1)
		begin
			load =1'b1;
			shift=1'b0;
			reset_b_counter=1'b0;
			reset_baud_b=1'b0;
			next_state=load_sr;
			increment = 1'b0;
		end
		else
		begin
			reset_b_counter=1'b0;
```

```
                next_state=reset_st;
                load =1'b0;
                shift=1'b0;
                increment = 1'b0;
        end
end
load_sr:
begin
        if(b_done)
        begin
                next_state = next_frame;
                reset_b_counter=1'b0;
                load =1'b0;
                shift=1'b0;
                increment = 1'b0;
        end
        else
        begin
                if(baud == 1'b1)
                begin
                        next_state = load_sr;
                        increment=1'b1;
                        shift=1'b1;
                        load =1'b0;
                        reset_b_counter=1'b1;
                end
                else
                begin
                        next_state=load_sr;
                        load =1'b0;
```

```
                        reset_b_counter=1'b1;
                        increment = 1'b0;
                        shift=1'b0;
                    end
            end
end
next_frame:
begin
            if(send)
            begin
            next_state=next_frame;
            load =1'b0;
            shift=1'b0;
            reset_b_counter=1'b1;
            increment = 1'b0;
            end
            else
            begin
            next_state=reset_st;
            load =1'b0;
            shift=1'b0;
            reset_b_counter=1'b0;
            increment = 1'b0;
            end
end
default:
begin
            next_state =reset_st;
            load =1'b0;
            reset_b_counter=1'b0;
```

```
                increment = 1'b0;
                shift=1'b0;
            end
        endcase
        end
endmodule
```

2.5.6 Simulation Result

First, the system is reset by applying active low synchronous reset, then, send is kept high for some clock cycles. The controller detects the send and outputs load signal active high. The output of reset_baud_counter and reset_baud_generator as active low. For each baud received, shift and increment is given as output with value 1. B_done is given as input to the controller when 10 bits are shifted out. The bit counter is reset. And the controller waits for the next packet.

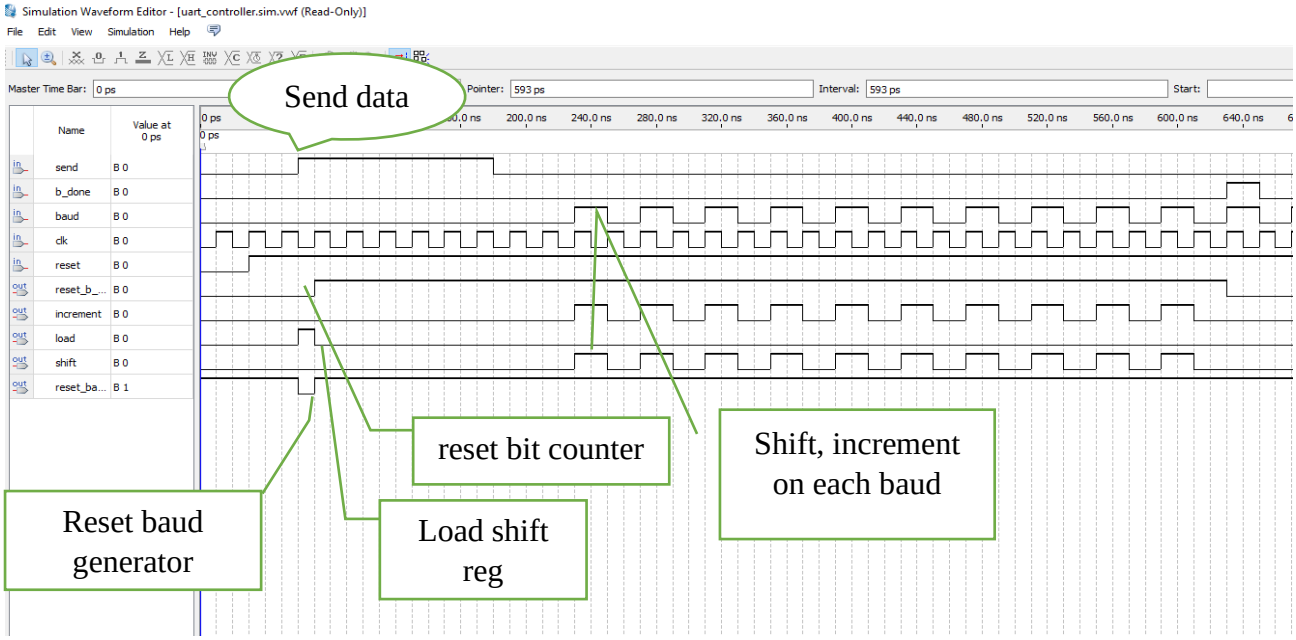

Figure 27 Simulation Result of UART TX Controller

2.5.7 RTL View

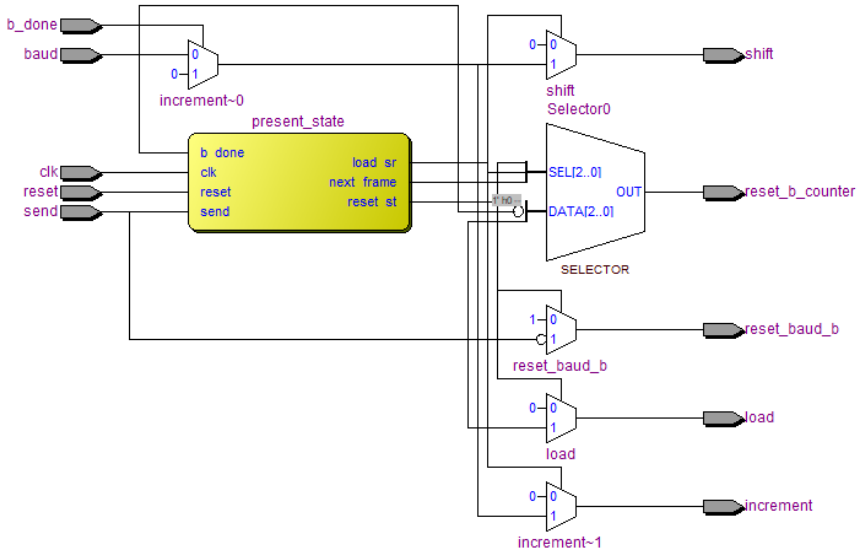

Figure 28 RTL view of UART TX Controller

2.6Synchronizer

Synchronizer is needed to send and reset_b signal. Send is active high asynchronous input given through a push button key on DE2 FPGA. This is asynchronous input can cause glitches and unexpected state machine transitions inside the UART Controller logic. Hence, to eliminate them, a chain of two d flip flops cascaded together is connected between send input of the transmitter to send input of the controller. This makes the design fully synchronous.

2.6.1 Block Diagram

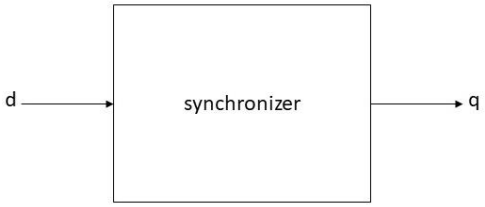

Figure 29 Block Diagram of Synchronizer

2.6.2 ASM Chart

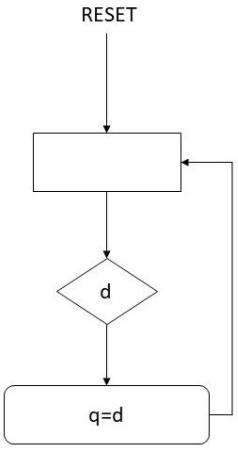

Figure 30 ASM chart of synchronizer

2.6.3 Verilog Code

```
module d_ff_edge(q,q_bar,d,clk,reset);
output q,q_bar;
input d,clk,reset;
reg q;

assign q_bar= ~q;
always @(posedge clk)
        begin
                if(reset == 0)
                        q=0;
                else
                        begin
                                q=d;
                        end
        end
endmodule
module synchronizer(q,d,clk,reset);
output  q;
input d,clk,reset;
```

wire d1;

d_ff_edge d_ff1(.q(q),.q_bar(),.d(d1),.clk(clk),.reset(reset)); //first stage of synchronizer

d_ff_edge d_ff2(.q(d1),.q_bar(),.d(d),.clk(clk),.reset(reset)); //second stage of synchronizer

endmodule

2.6.4 Simulation Result

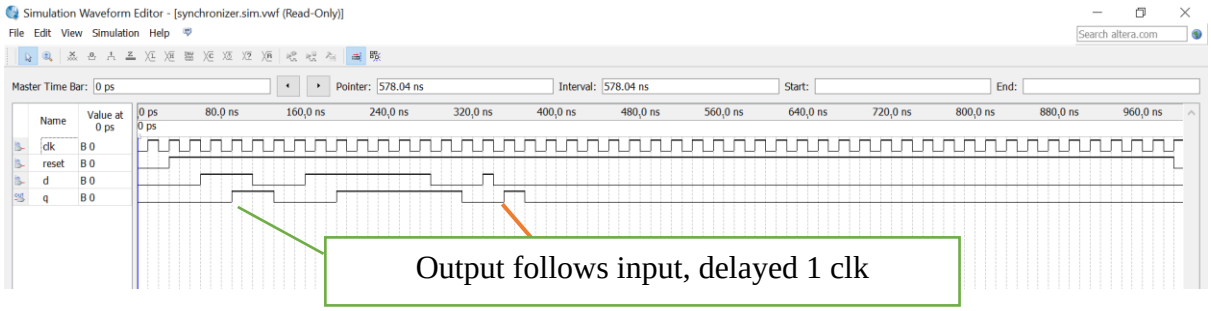

Figure 31 Simulation result of synchronizer

2.6.5 RTL View

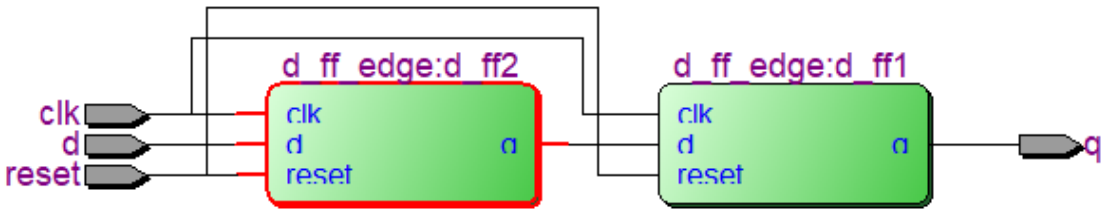

Figure 32 RTL View of Synchronizer

3 UART Receiver

UART receiver is responsible for receiving serial data from a UART transmitter. The input data is 1 bit. The Receiver uses 50 MHz system clock, and reset_b is given from FPGA switch.

The UART Receiver controller issues control signals to other modules like shift register to shift data whenever a new data bit is received. The data bits are shifted using a shift register in serial in parallel out mode. It also controls the increment and reset of bit counter. It has the additional responsibility for a correct sampling of data in the middle of the pulse, so it controls the reset of baud generator. Finally, once all bits are loaded in flip flops, the load signal is given to shift register to retrieve parallel data. The data is passed to an error detector to detect data and packet integrity. Frame error or parity error is output using onboard FPGA LEDs.

Verilog code for integration of blocks is given in Appendix 7.2.

The input and output of the receiver are:

Figure 33 Input and Output of UART Receiver

A general UART receiver Block Diagram is shown below:

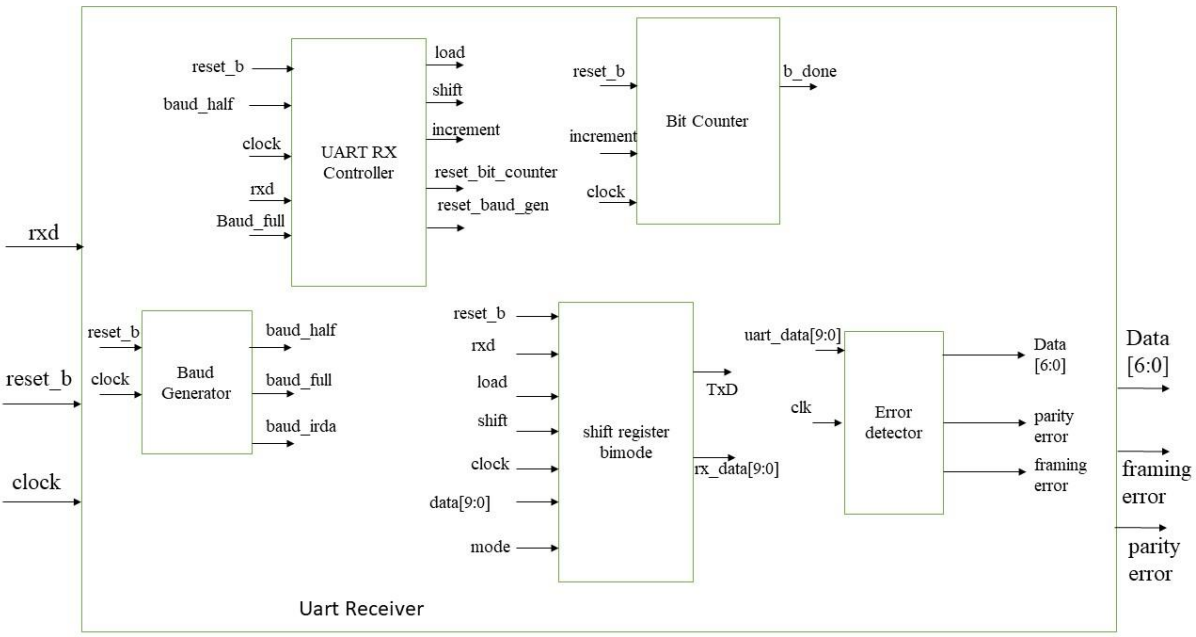

Figure 34 Block Diagram of UART Receiver

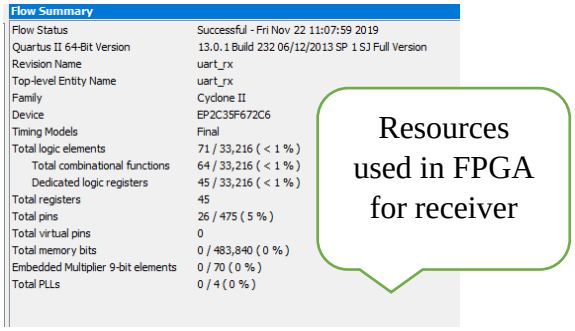

Figure 35 Compilation report of UART receiver

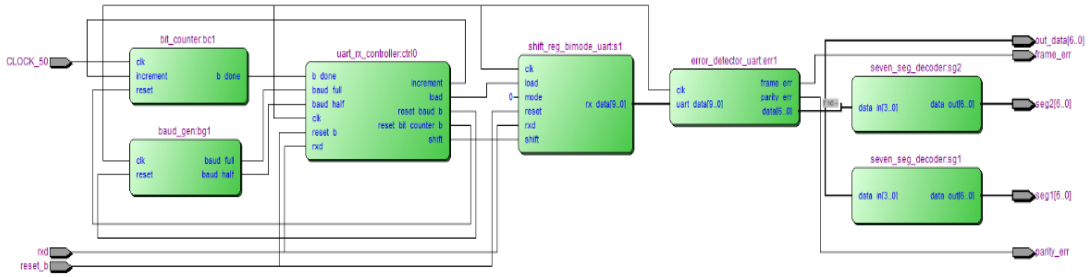

Figure 36 Schematic of UART receiver

To simulate and get accurate prediction in the simulations, the transmitter designed and verified on FPGA is connected to the rxd signal of receiver. The data is input to transmitter along with reset_b and send with appropriate timing. Model sim was used to simulate with a simple testbench.

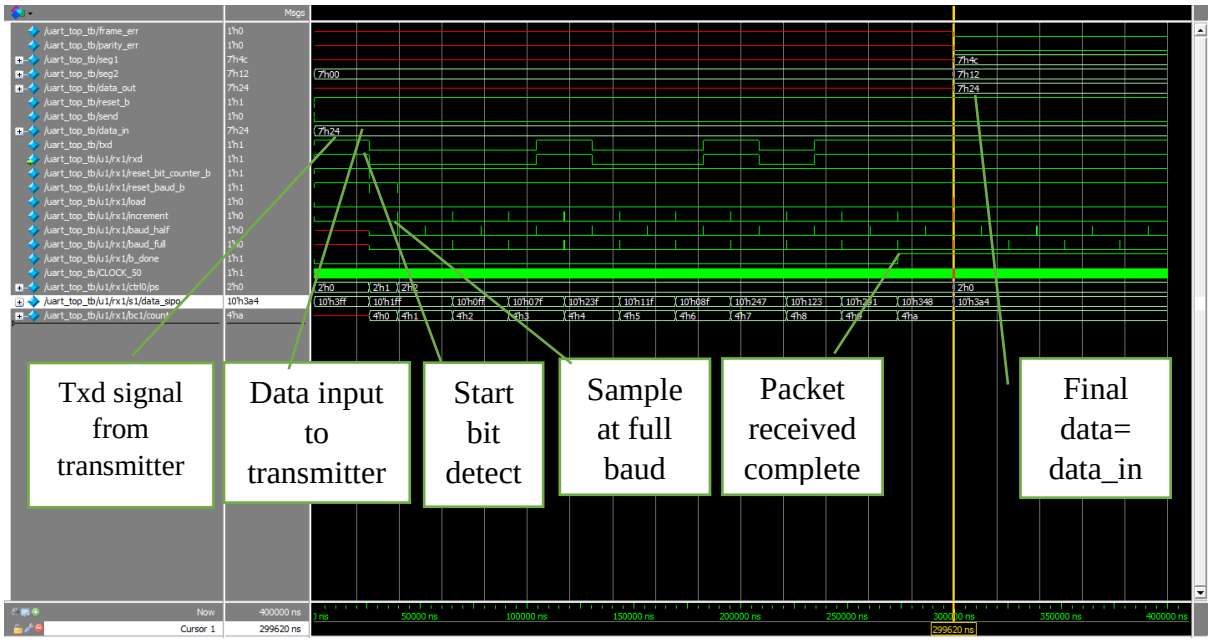

Figure 37 Simulation result of complete UART Receiver system

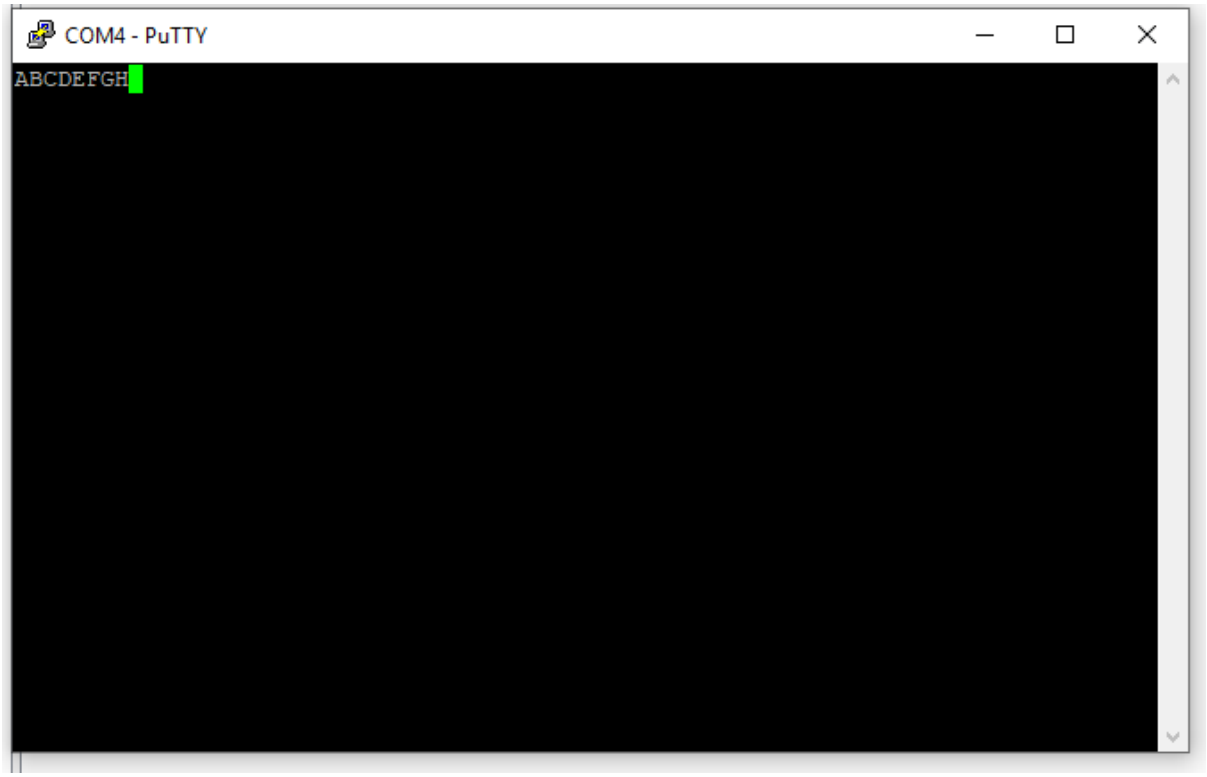

Figure 38 Putty output when FPGA is receiving data from computer using UART interface

The main modules inside a UART receiver are:

3.1 Bit counter

Bit counter is reused from UART transmitter. Bit counter b_done is connected to UART Receiver Controller. Controller of receiver controls increment and Reset. Refer to Section 2.1 for specification, block diagram, ASM charts, simulation results, RTL view and Verilog code.

3.2 Shift Register

Shift register mentioned in section 2.2 is used in SIPO mode, the mode signal is made 0 while instantiating in the receiver. Refer to Section 2.2.2.2 for details on operation in SIPO mode and section 2.2.4 for ASM charts. Refer to Figure 15 for simulation results.

3.3 Baud Generator

Baud generator is used that was designed in Section 2.4. The baud_half is used for detecting the middle of rxd pulse for beginning of start bit. Baud_full is used for receiving the next set of bits.

3.4 UART RX controller

3.4.1 Specification

UART receiver controller detects the UART packet by polling for transition from idle to start of transmission bit. Once detected, the baud counter and bit counter are reset. To sample data at the middle of the data pulse, the controller changes the state to rx_start_bit state and waits for half baud. After half baud is received, the baud generator is reset, and the state is changed to rx_data state. Now the data will be sampled every full baud. The time is the middle of rxd data. The controller checks for b_done signal to see if all the 10 bits are received if not then for each full baud signal, shift and increment signal is given to shift register and bit counter

respectively. After 10 baud cycles, completion of the packet is indicated by bit counter using b_done signal input to the controller. The controller resets to reset_st state and waits for the next start of packet. Finally, once all bits are loaded in flip flops, the load signal is given to shift register to retrieve parallel data.

3.4.2 Block Diagram

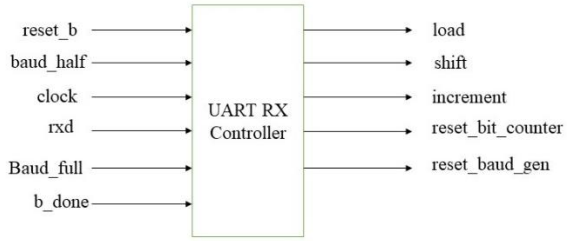

Figure 39 Block Diagram of the RX controller

3.4.3 ASM chart

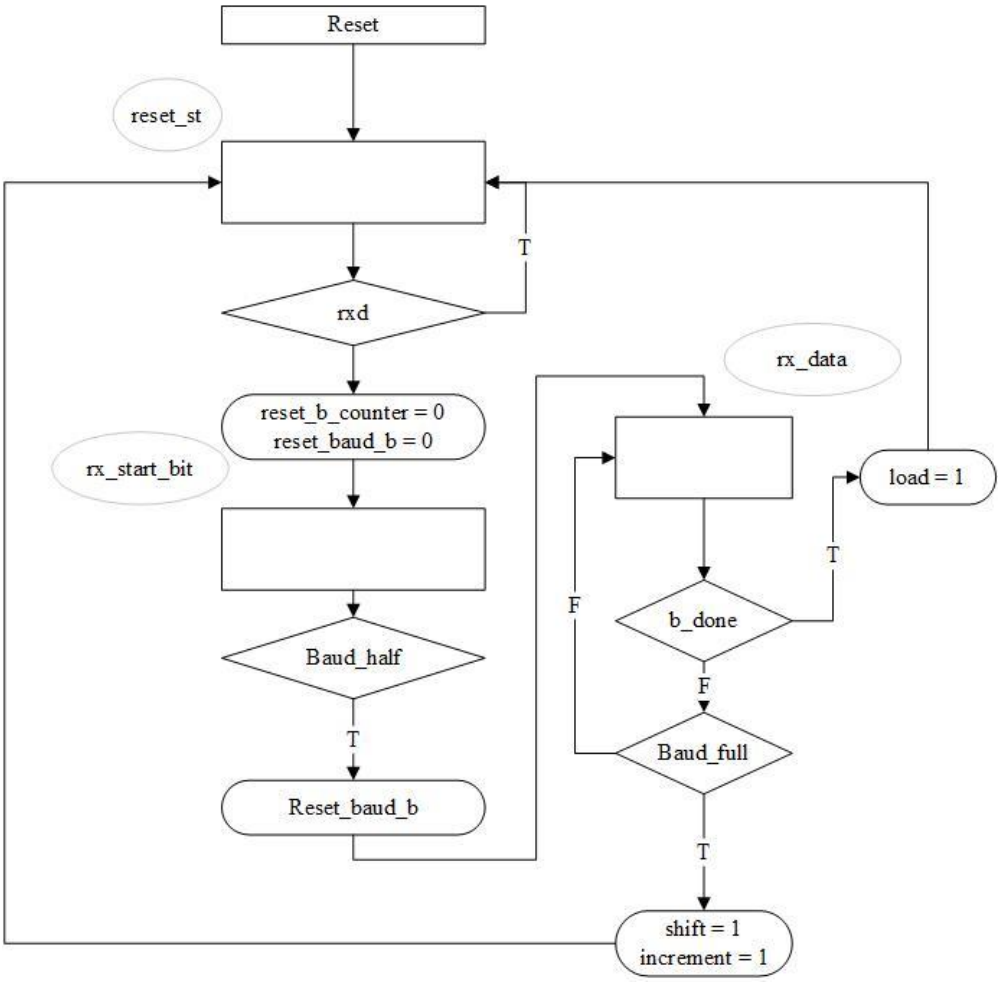

Figure 40 ASM chart of RX Controller

3.4.4 Verilog Code

```
module
uart_rx_controller(increment,load,shift,reset_bit_counter_b,reset_baud_b,rxd,baud_half,baud
_full,b_done,reset_b,clk);

        output reg increment,load,shift,reset_bit_counter_b,reset_baud_b;

        input baud_half,baud_full,b_done,reset_b,clk,rxd;

        reg [1:0]ps;

        reg [1:0]ns;

        parameter reset_st =2'd0,rx_start_bit =2'd1,rx_data=2'd2;

        always@(posedge clk)

        begin

                if(reset_b ==1'b0) //active low synchronous reset

                begin

                        ps <= 0; //reset to reset state

                end

                else

                        ps <= ns; //generate d flip flops / sequential logic to remember the
present state.

        end


        always @(ps,baud_half,baud_full,b_done,rxd,reset_b)

        begin

        //initialize the outputs and next states

        ns = ps;

        shift=1'b0;

        increment =1'b0;

        load =1'b0;

        reset_bit_counter_b =1'b1;

        reset_baud_b =1'b1;

        if(reset_b==1'b0)

        begin

        reset_bit_counter_b = 1'b0;//reset the bit counter when system is reset
```

```
reset_baud_b = 1'b0; //reset the baud generator when system is reset
end
case(ps)
    reset_st:
    begin

        if(rxd == 1'b0)
        begin

            reset_bit_counter_b =1'b0; //if the rxd is 0 then reset bit counter, reset baud generator and go to next state
             reset_baud_b=1'b1;
            ns = rx_start_bit;
        end
    end
    rx_start_bit:
    begin
        if(baud_half)
        begin
            reset_baud_b = 1'b0; //reset baud generator again to synchronize it on midddle of rxd pulse
            ns = rx_data; //go to next state for processing next bit of data packet
            shift =1'b1; //shift when first half baud is detected i.e start bit
            increment =1'b1; //increment bit counter to signify start bit
        end
        else
        begin
            ns = rx_start_bit; //wait for baud_half to arrive . dont change state
        end
    end
```

```
                rx_data:
                begin
                        if(baud_full)
                        begin
                                shift =1'b1; //wait for full baud to arrive and sample in middle of
rxd pulse
                                increment =1'b1; //increment bit counter to count the bits of uart
packet
                                if(b_done ==1'b1) //wait for all bits to be counted and shifted
                                begin
                                        ns = reset_st; //go to reset_st and wait for next packet
                                        load =1'b1; //load the data from sipo shift register to get
valid uart packet
                                end
                                else
                                        ns = rx_data; //wait for data transmission to complete
                        end
                end
                default:
                begin
                        ns = 2'bx; //illegal state so dont care condition
                end
        endcase

        end
endmodule
```

3.4.5 RTL View

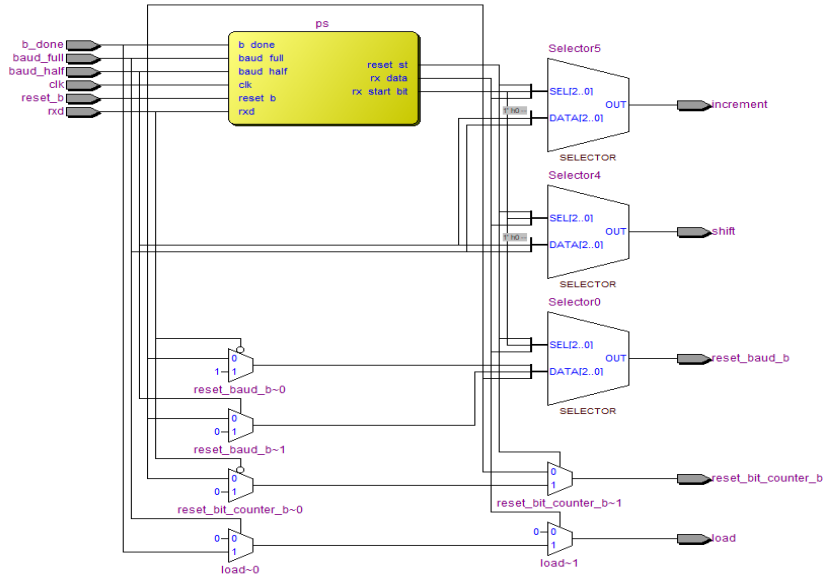

Figure 41 RTL view of UART RX Controller

3.4.6 Simulation Result

Firstly, the system is reset by the input of the active low synchronous reset. When reset is removed, rxd is provided to the controller, when rxd =0 is detected, baud generator and bit counter are reset and then reset is removed to start counting the bits. For each data bit on baud = 1, shift and increment are issued. Once all bits are transferred, b_done is given as input. The controller stops shift, increment and outputs 0 on these signals. Also, the load is issued as well as output to load the parallel data.

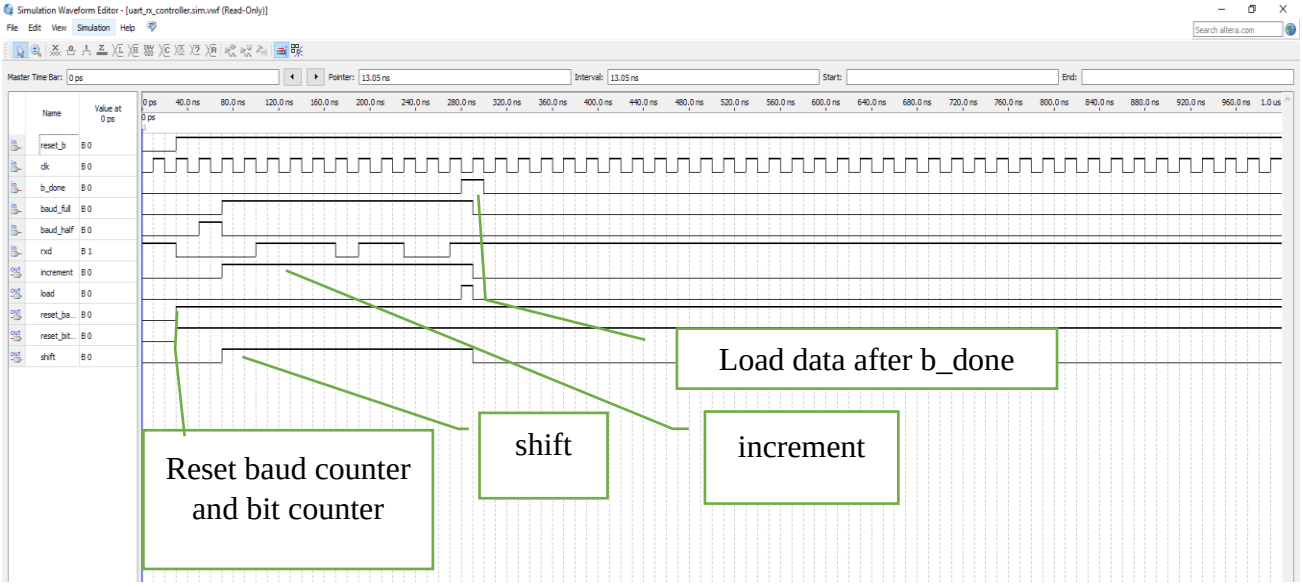

Figure 42 Simulation result of UART RX Controller

3.5 Error detector

3.5.1 Specification

UART packet consists of a data stream and the parity bit. The parity bit needs to match the parity of data received to make sure data integrity is maintained [4].

Parity is calculated from data extracted. Calculated parity is compared with penultimate bit of the packet. If it does not match then parity error LED glows.

Another data integrity check is framing of UART packet. The packet should receive in same order as it is transmitted. The start bit is checked for value 0, and the stop bit is 1.

After extracting data, error detector must make it available to 7 segment decoders to display value on seven-segment displays.

3.5.2 Block Diagram

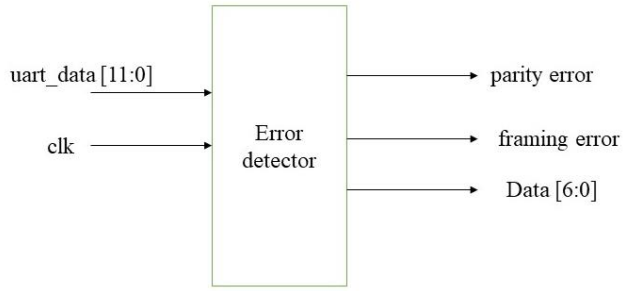

Figure 43 Block Diagram of Error Detector

3.5.3 ASM chart

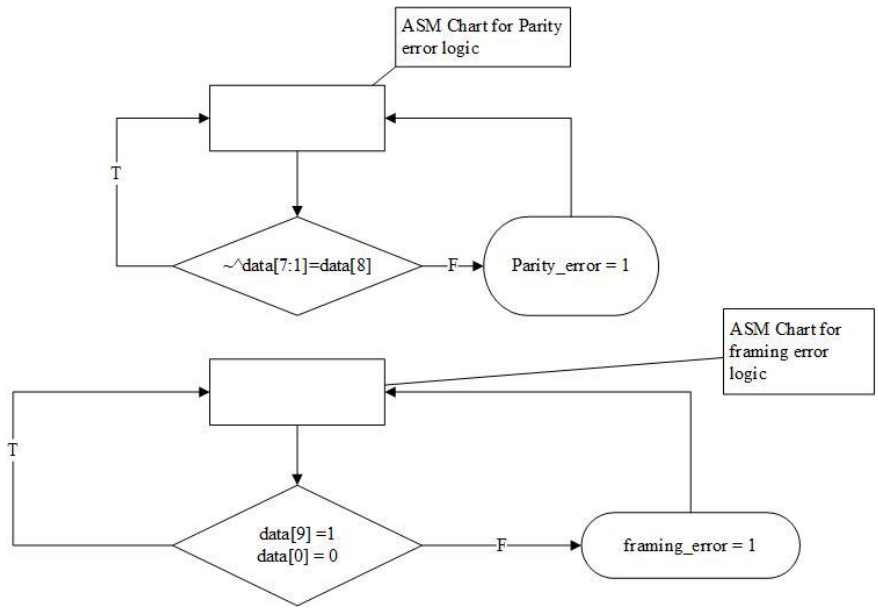

Figure 44 ASM Chart of Error detector

3.5.4 Verilog Code

```
module error_detector_uart #(parameter
DATA_WIDTH=4'd7)(frame_err,parity_err,data,uart_data,clk);

        output reg [DATA_WIDTH-1:0]data;
```

```
        output frame_err,parity_err;
        input [DATA_WIDTH+2:0]uart_data;
        input clk;


        assign parity_err=(uart_data[DATA_WIDTH+1]
!=(^uart_data[DATA_WIDTH:1]))?1'b1:1'b0;
        assign frame_err=((uart_data[DATA_WIDTH+2] != 1'b1) || (uart_data[0] !=
1'b0))?1'b1:1'b0;
//      assign data=(frame_err || parity_err)?'bx:uart_data[DATA_WIDTH:1];
        always @(posedge clk)
        begin
                data <= uart_data[DATA_WIDTH:1];
        end
endmodule
```

3.5.5 Simulation Result

Random data is given on random intervals to see the parity and framing error as well as extracted data. All the conditions are recorded in simulation. First, incorrect frame and correct parity along with data is sent, both framing and parity error is seen on output. Then, a correct UART packet is sent with the correct frame, and correct parity is given. The output of frame error and parity error is 0. Next, a packet with the wrong parity bit is given as input. The output is parity. Lastly, a packet with wrong framing error and parity is input, and output is parity error and framing error.

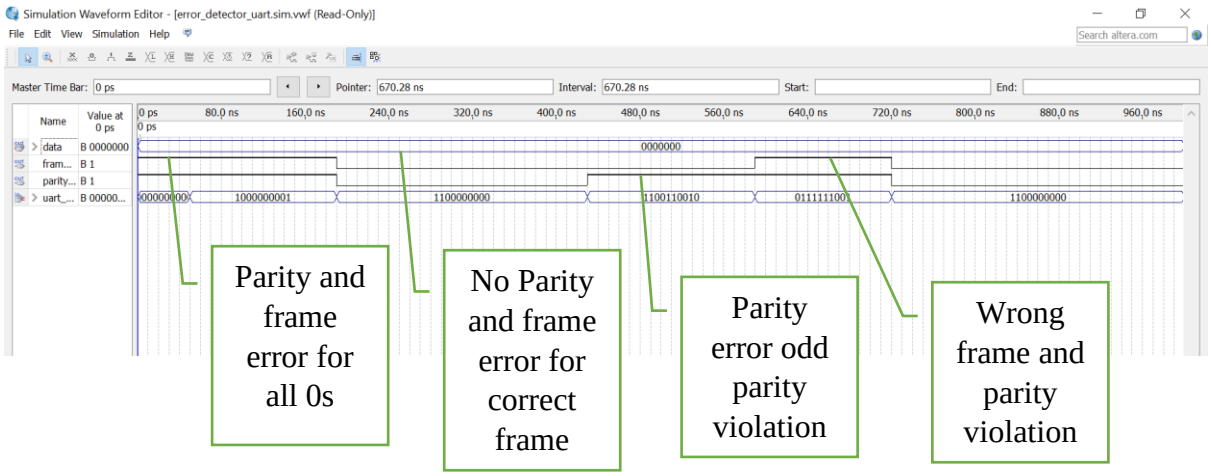

Figure 45 Simulation result of error detector

3.5.6 RTL View

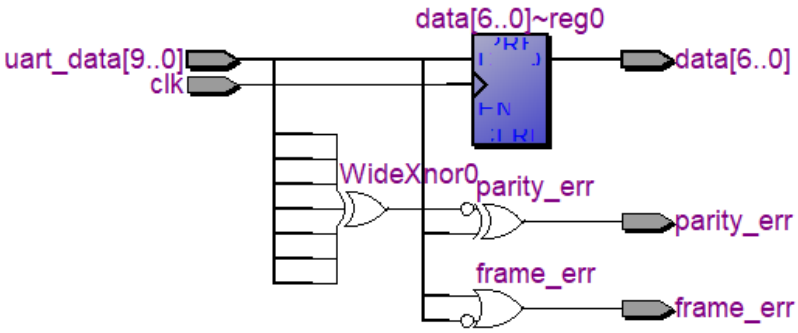

Figure 46 RTL view of the error detector

3.6 BCD to 7-Segment decoder

3.6.1 Specification

BCD data is 4-bit input to the decoder which converts it into necessary decoding for a 7-segment display. DE2 board HEX5 and HEX4 are active low 7-segment LEDs. Hence the output is 7-bit. An active-low bit means that the respective light of display will be ON. Figure 47 shows the 7-segment display led position. Figure 48 depicts the truth table, which is used to construct the ASM chart and Verilog code [5].

Figure 47 7-segment display on the DE2 Board [6]

Digit	A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	1	1	1	1
2	0	0	1	0	0	0	1	0	0	1	0
3	0	0	1	1	0	0	0	0	1	1	0
4	0	1	0	0	1	0	0	1	1	0	0
5	0	1	0	1	0	1	0	0	1	0	0
6	0	1	1	0	0	1	0	0	0	0	0
7	0	1	1	1	0	0	0	1	1	1	1
8	1	0	0	0	0	0	0	0	0	0	0
9	1	0	0	1	0	0	0	0	1	0	0

Figure 48 Truth Table of BCD to 7 Segment Decoder [5]

3.6.2 Block Diagram

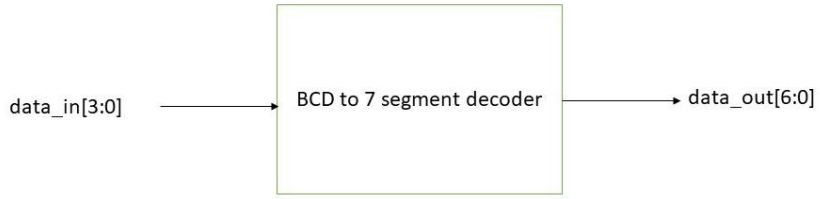

Figure 49 Block diagram of 7-segment decoder

3.6.3 ASM Chart

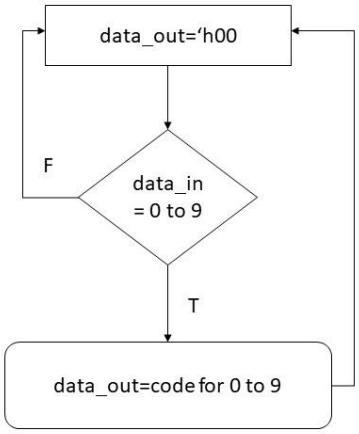

Figure 50 ASM chart for BCD to 7-segment decoder

3.6.4 Verilog Code

```
module seven_seg_decoder #(parameter DATA_WIDTH=7)(data_out,data_in);
        output reg[DATA_WIDTH-1:0]data_out;
        input [3:0]data_in;

        localparam ZERO=7'b000_0001;
        localparam ONE=7'b100_1111;
        localparam TWO=7'b001_0010;
        localparam THREE=7'b000_0110;
        localparam FOUR=7'b100_1100;
        localparam FIVE=7'b010_0100;
        localparam SIX=7'b010_0000;
        localparam SEVEN=7'b000_1111;
        localparam EIGHT=7'b000_0000;
        localparam NINE=7'b000_0100;
        localparam DEFAULT=7'b000_0000;

        always@(data_in)
        begin
```

```
        case(data_in)
        4'b0000: data_out = ZERO;
        4'b0001: data_out = ONE;
        4'b0010: data_out = TWO;
        4'b0011: data_out = THREE;
        4'b0100: data_out = FOUR;
        4'b0101: data_out = FIVE;
        4'b0110: data_out = SIX;
        4'b0111: data_out = SEVEN;
        4'b1000: data_out = EIGHT;
        4'b1001: data_out = NINE;
        default: data_out = DEFAULT;
        endcase
    end
endmodule
```

3.6.5 Simulation Result

For input data of BCD value 3, as per truth table, the output expected is that e and f will be off. Hence, the output seen on data_out is 0000110. Random set of data are given on a random interval. When the value exceeds 9, the output is 0000_000, which is the default value.

Figure 51 Simulation result of BCD to 7-segment decoder

3.6.6 RTL View

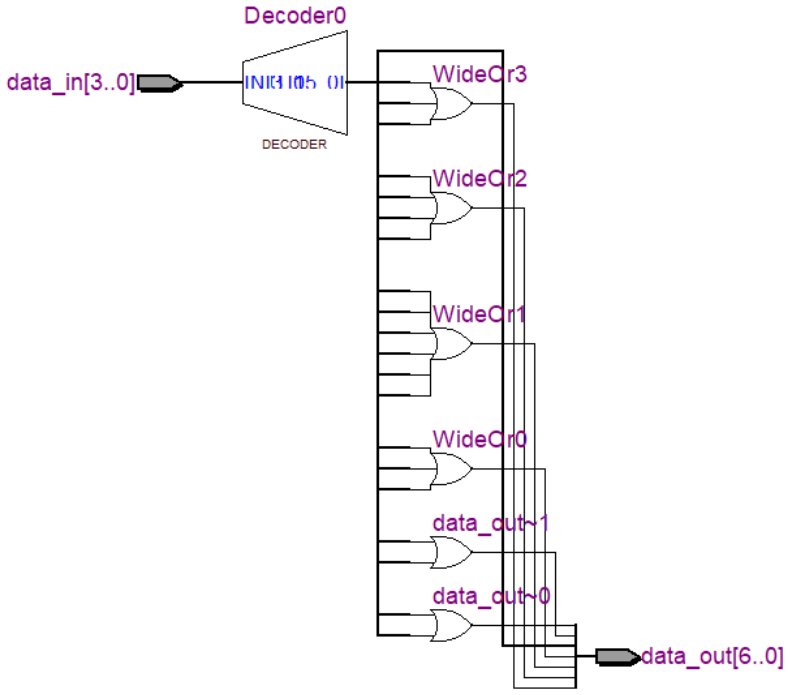

Figure 52 RTL View of BCD to 7 segment decoder

4 UART IrDA Transmitter

The UART IrDA Transmitter consists of a UART transmitter block connected to IrDA modulator. The modulator converts serial data into IrDA modulation.

UART IrDA Transmitter implements IrDA physical layer protocol v1.0. The timing diagram in Figure 55 depicts that when the TxD value is 0, the txir output must transmit a pulse of value 1. The width of the pulse is 3/16 of the baud. The pulse rising edge was decided to occur at 7/16 of baud, stay 1 for 3/16 period and then 0 for the rest of the baud period. For TxD value 1, no pulse is transmitted, and txir is 0 [7]. The IR LED is connected on the FPGA to txir.

PIN_AE24 is the IR LED TX pin on DE2 board. Refer to Appendix 7.3 for Verilog code of transmitter along with IrDA modulator.

Verilog code for integrated modulator and UART transmitter is in Appendix 7.3.

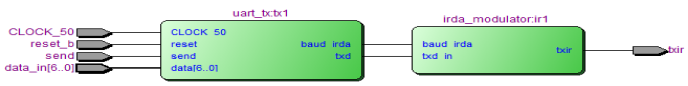

Figure 53 Schematic of UART IrDA transmitter

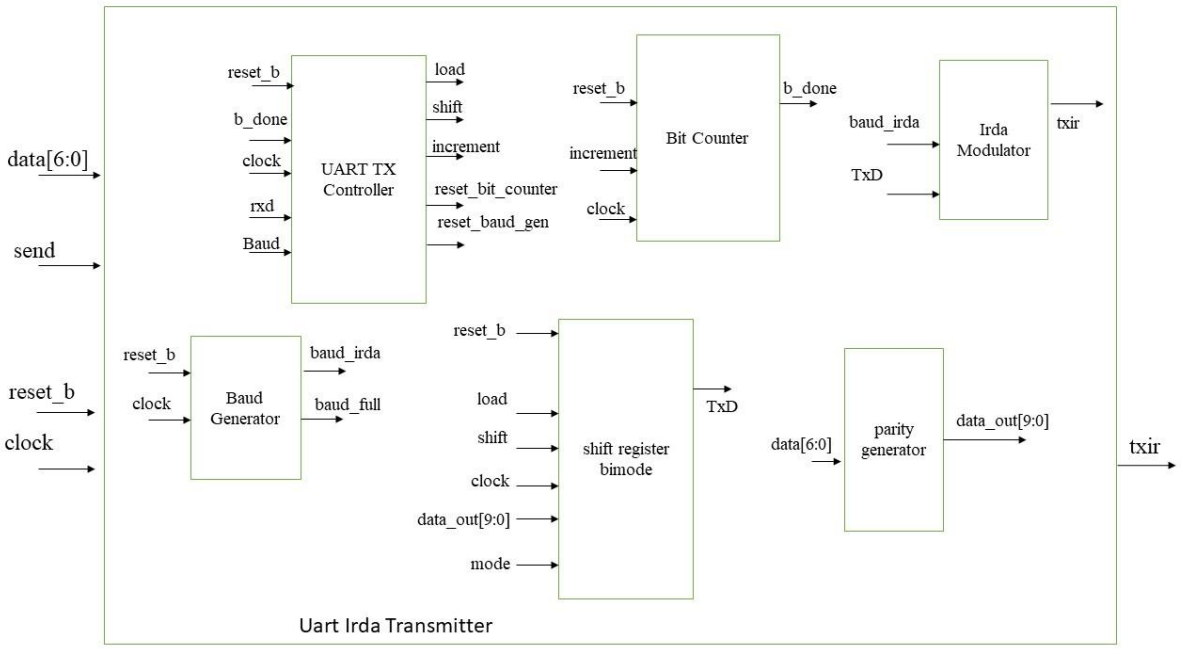

Figure 54 Block diagram of UART IrDA Transmitter

4.1 IrDA modulator

4.1.1 Specification

IrDA modulation timing diagram is shown in Figure 55. For every 0 of TxD, a pulse of width 3/16 of baud is given to the IR LED. The HSDL-3201 IR LED has stuck-on protection which disables the after typical 1.6 us and maximum 2.23 us time [8].

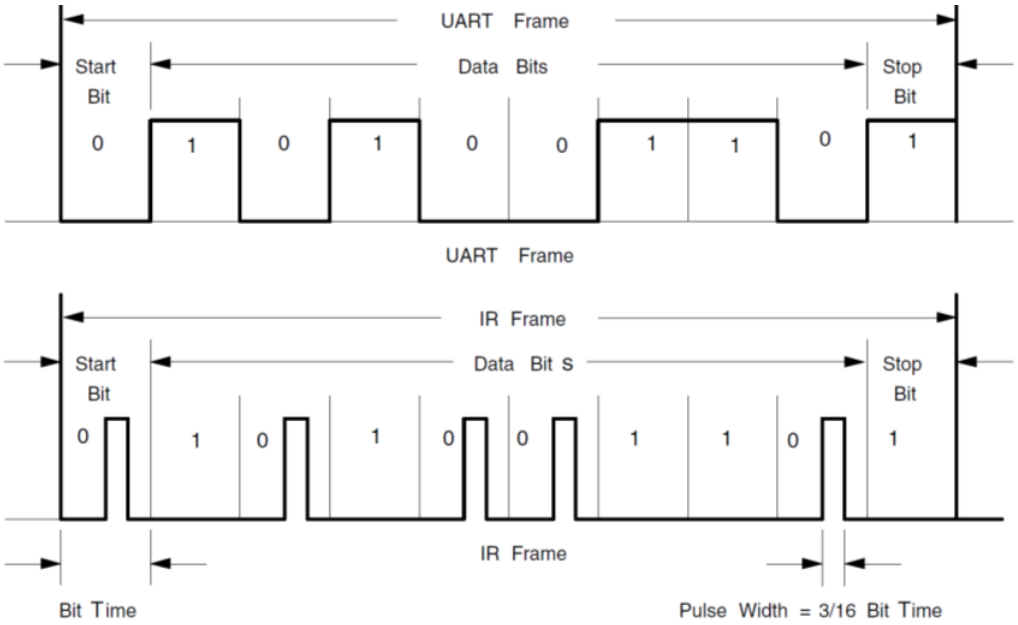

Figure 55 IrDA modulation timing [2]

IrDA modulator takes TxD input from UART transmitter and baud_irda signal, which has a pulse width of 3/16 of the baud. This signal is the output of the baud generator. Baud_irda is

passed to txir output when TxD signal is 0. Else, a 0 is passed to the IR LED. Hence, LED only glows when sending a data bit with value 0.

4.1.2 Block Diagram

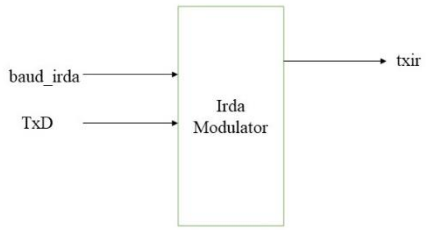

Figure 56 Block Diagram of IrDA modulator

4.1.3 ASM Chart

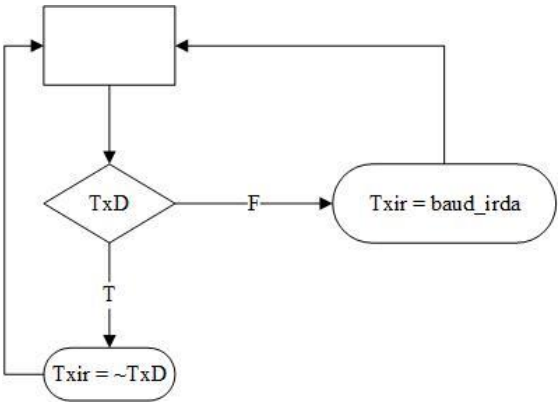

Figure 57 ASM chart of IrDA modulator

4.1.4 Verilog Code

```
module irda_modulator(txir,baud_irda,txd_in);
        output reg txir; //txir output fed to ir led
        input baud_irda,txd_in; //txd_in from uart transmitter and baud irda from baud generator
        always @(baud_irda,txd_in)
        begin
                if(txd_in == 1'b0)
                begin
                        txir = baud_irda; //only transmit pulse value 1 for 3/16 baud if 0 needs to be transmitted
                end
                else
                        txir = ~txd_in; //make txir 0 if the txd_in is 1.
```

end

endmodule

4.1.5 Simulation result

As annotated in Figure 58, the baud_irda is passed to the txir output when the txd of data is 0. When it is 1, no pulse from txir is passed, and the output is 0.

Figure 58 Simulation result of IrDA modulator

4.1.6 RTL View

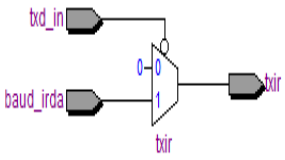

Figure 59 RTL view of IrDA modulator

5 UART IrDA Receiver

UART IrDA receiver will receive the signal from the IrDA transmitter. It implements the IrDA physical layer v1.0. DE2 Board has PIN_AE25 as the IR LED RX input. This IR LED receives the IR signal from the transmitter whose pulse width is 1.6us typical. The demodulator and receiver are combined to form the UART IrDA receiver. The demodulator demodulates the pulses received from IR LED. Then the receiver receives the serial data and converts to parallel form to be displayed on the 7-segment display.

Verilog code for integrated block of IrDA demodulator connected to UART receiver is in Appendix 7.4.

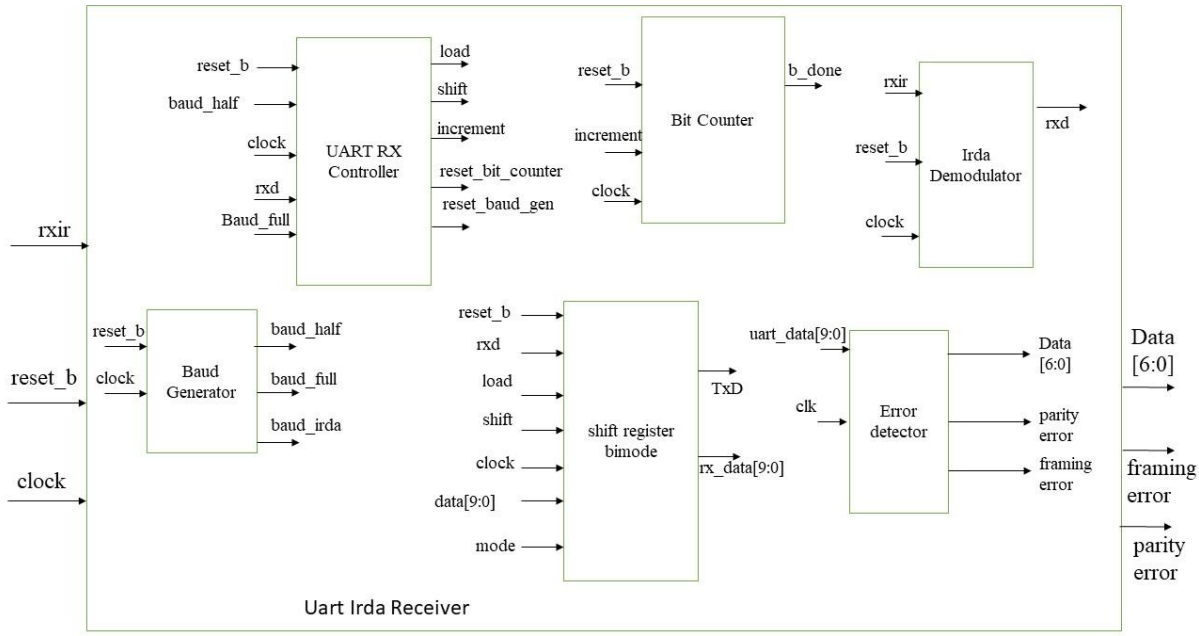

Figure 60 Block diagram of UART IrDA Receiver

5.1 IrDA demodulator

5.1.1 Specification

IrDA demodulator is responsible for capturing the 1 -> 0 transition which is start bit of transmission. The LED will be ON for just 1.6us [8]. The demodulator captures the transition and latches it on rxd output [7]. In order to avoid latches in design, state machine with flip flop was decided. Another possible solution to use latches instead of flip flop. But latches cause debug issues in design, and it is easier to debug a state machine. The demodulator is a state machine, in the state 00, when rxir is 0 the state changes to 01, and also the counter value is reset to 0. In state 01, the counter is incremented each time until counter reached baud_count, i.e. 1302. By default, rxd is set to 1, so when the rxir is not 0, then rxd is always 1.

5.1.2 Block Diagram

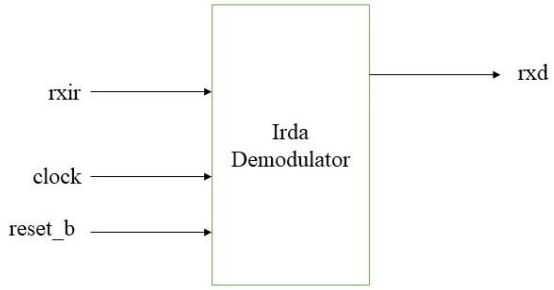

Figure 61 Block Diagram of IrDA demodulator

5.1.3 ASM Chart

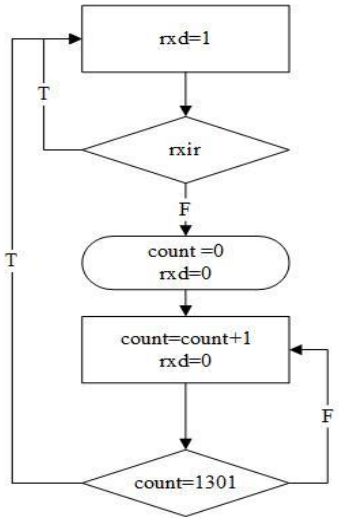

Figure 62 ASM chart of IrDA demodulator

5.1.4 Verilog Code

```
module irda_demodulator(rxd,rxir,CLOCK_50,reset_b);

output rxd;

input CLOCK_50,reset_b,rxir; //use system clk and reset and take rxir from IR LED

//reg [7:0]count;

reg [10:0]count,count_next; //counter variables for baud calculation

reg [1:0] ps,ns; //present state and next state variables


assign rxd = (ps == 2'b01) ? 1'b0:(ps == 2'b00 && rxir ==1'b0)?1'b0:1'b1; //if the ps is 01 then output is a 0 and this 0 untill if next bit is also 0 else it is 1.

always@(posedge CLOCK_50)

begin

        if(reset_b == 1'b0) //synchronous active low reset

        begin

        count <= 1'b0; //reset the counter if system is reset.

        ps <= 2'b00; //reset the present state to 00 state.

        end

        else

        begin

        count <= count_next; //if not reset then count_next is assigned to counter
```

```
        ps <= ns; //next state is assigned to present state
        end
end

always@(ps,count,rxir)
begin
                //initialize the regs
                count_next = count;
                ns = ps;

                case (ps)
                00: begin
                        if(rxir == 1'b0)
                        begin
                                ns=2'b01; //if a 1->0 transition is detected from IR led then go to next state 01 to start the counter and make rxd 0
                          count_next =0; //initialize counter to 0 when sampling a 0, do it on the 1->0 transition
                        end
                        else
                        begin
                                ns = 2'b00; //wait for 1->0 transition so dont change state
                        end

                end
                01: begin
                        count_next = count_next+1'd1; //increment the baud counter
                        if (count == 'd1301)
                        begin
                                ns=2'b00; //if the baud count is reached then go to 00 state for next bit
```

```
                end
                else
                begin
                        ns = 'b01; //wait for the baud count to be reached, the
counter must increment


                end
        end
        default:
        begin
                ns =2'b00; //wait for next bit if ebtered illegal state
        end
        endcase
end
endmodule
```

5.1.5 Simulation result

For simulation, the UART IrDA Transmitter was connected with demodulator input. The inverted txir is connected to rxir. This allows data flow to follow the UART protocol. Thus, if data input and data out is the same, we can say that the whole system is working correctly. Model sim was used with a simple testbench to perform simulation.

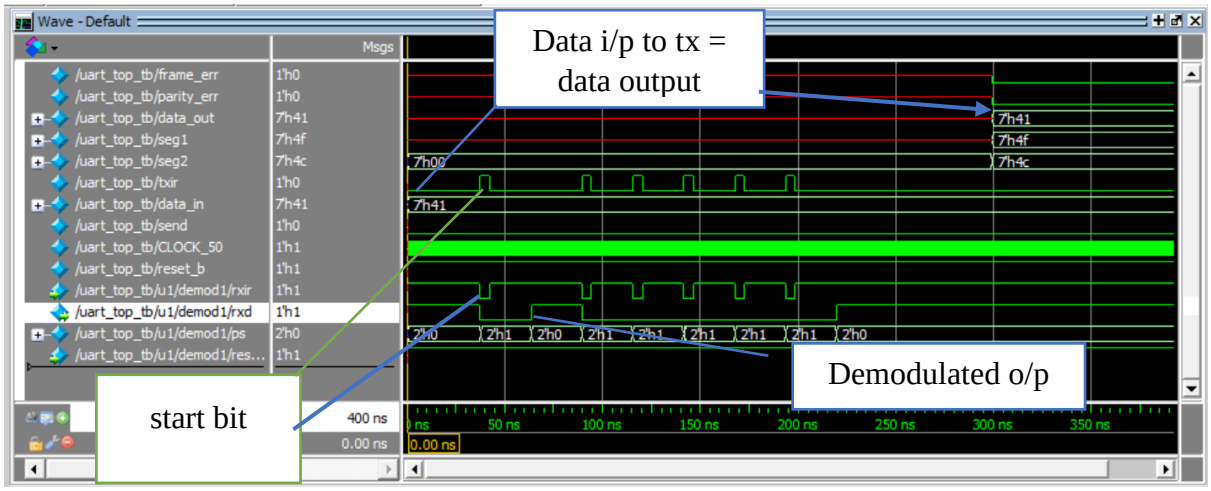

Figure 63 Simulation result of IrDA Receiver

5.1.6 RTL View

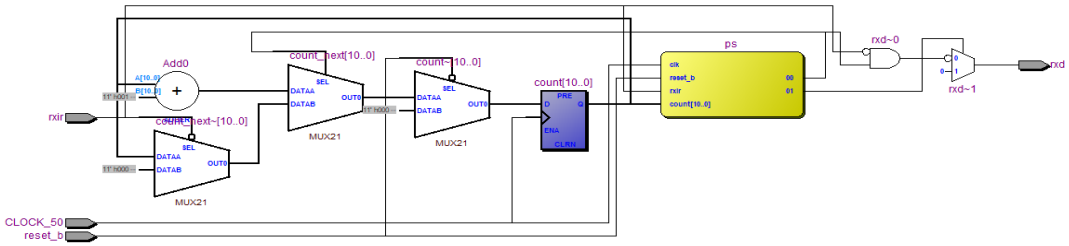

Figure 64 RTL view of IrDA modulator

6 Discussion

6.1 What worked?

All the simulations were verified and completed before testing on the FPGA board. The expectation is no issues to occur while verifying on FPGA by connecting it to PC using USB to UART cable. The system worked perfectly on the UART transmitter.

There were specific issues while verifying the receiver part when data typed from putty console was not visible on the screen. It was solved by watching the online tutorial and switching on the “force echo on” setting resolved the issue. The data was correctly seen on FPGA.

During testing of IrDA, demodulator trials were done with various design ideas. Finally, a state machine design is selected as it is less error prone than combinational circuits and latches, but there were some trade-offs which are discussed in Section 6.2. The design worked without any issues.

UART IrDA transmitter and receiver were also verified on FPGA board.

Since the sampling of data in the receiver is at the middle of baud time, it is stable design. There is a rare chance of missing the start bit of UART packet. Another strength of design is maintaining Fully synchronous design across modules which will not cause glitches to occur. For send input, synchronizers, i.e. chain of two d flip flops cascaded together were used. This eliminates the asynchronous nature of the send signal. Similarly, reset_b is also an asynchronous input through switch on FPGA which is passed through a d flip flop synchronizer chain. So, accidental asynchronous reset would be converted to fully synchronous, and any glitches will not affect design.

6.2 What did not work or What can be improved?

The design for IrDA demodulator works perfectly. But IrDA demodulator is implemented using a state machine which detects a 1->0 transition. The internal baud_counter is implemented in demodulator as a separate baud counter was required to have reset controlled by rxir 1->0 transition without changing the UART receiver design. UART controller resets baud generator on half baud inside UART receiver when 0 is detected on rxd. Hence, the output of the receiver baud generator cannot be used by the demodulator. This decision causes additional counter hardware inside demodulator. Some improvements can be made to reduce this overhead.

Another issue in demodulator is usage of edge detection of 1->0 transition inside 00 state. The data is not sampled in the middle of the rxir data. This can cause detection of start bit even if there is a slight glitch on UART transmitter LED. Accidental packet can be intercepted due to this. The design can be improved by sampling in the middle of 1.6us pulse duration. This may require additional hardware.

Using the same shift register and the baud generator created some warnings. The warnings are regarding connectivity checks. The warnings are caused by floating inputs and outputs in shift register and baud generator. These warnings occur when instantiating it in transmitter where input and output required by receiver were kept floating. Similarly, warnings occur in the receiver when input and output needed for the transmitter are kept floating. For eg., in shift register txd is only required in the transmitter when this output is floating in the receiver, Quartus shows connectivity check warning. Some other design improvements could have been used to mask these errors.

6.3 Full System testing

6.3.1 Transmitter Testing

First, all the modules inside the transmitter were connected using wires. The connections were checked after synthesis using Quartus RTL viewer. The next step is to verify the simulations. A simple test bench was written in Verilog where clock, reset_b, send and data_in were made reg while txd is made wire. The complete UART transmitter was instantiated and connected with the wire and reg. Using the initial block in the test bench module, all regs were initialized. Then using Verilog timing constructs, clock was created for 20ns period. Inputs were applied at different times. The code can be referred to in Appendix 7.5. The value seen at txd was matched with input data and packet structure. The data was accurate with accurate timing.

Hence, design was dumped and verified on DE2 FPGA board by connecting FPGA to PC using UART to USB cable. SW0 was used as reset. It was first turned ON first to reset the FPGA and then the switch was closed. Using Key2 on FPGA, a set of data inputs formed on SW8-SW2 was sent. Putty was used as console with settings in Figure 65 on PC. The output was ASCII value which was converted to binary by referring to the ASCII table depicted in Figure 66. The data was verified with the data given through switches SW2(LSB) to SW8(MSB). This was repeated for 4-5 data set.

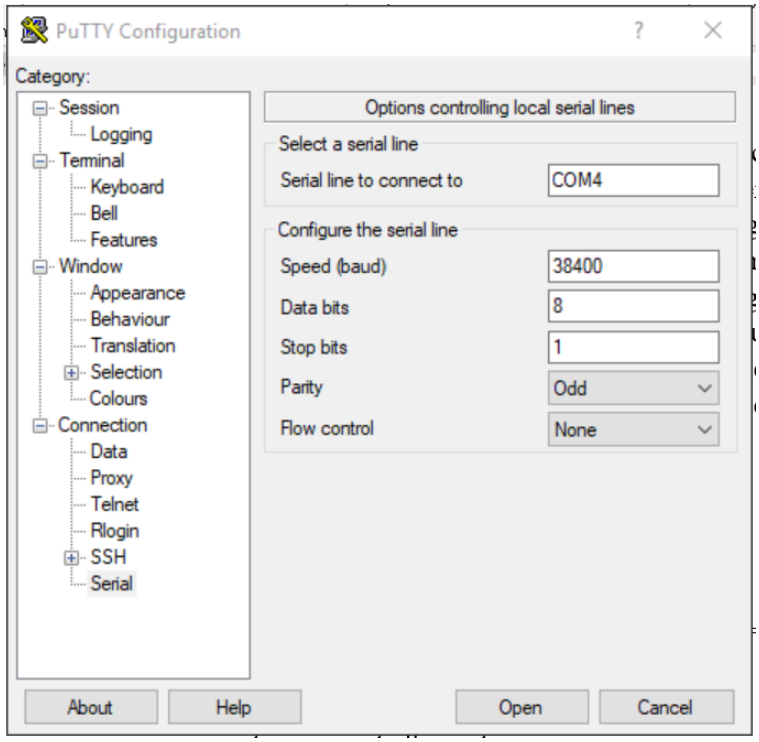

Figure 65 Putty settings

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Figure 66 ASCII Table [9]

6.3.2 Receiver Testing

Since the transmitter is working and verified model, it can be used as a stimulus generator during the simulations. Instantiating both UART Transmitter and receiver together and

connecting txd to rxd gives a complete UART system. The testbench in Appendix 7.5 can be used with modification of just changing the name of instantiating module. The inputs applied to transmitter and signals are probed inside the receiver sub-modules as well as receiver signals. This ensures no protocol violations occur while giving input to the receiver. Output data received is matched with data input through testbench. It was verified to be the same. The value coded by 7 segment display was matched with the truth table as well.

Next step is to verify the design on FPGA. The setup is the same as depicted in 6.3.1. Using putty again in same config as Figure 65. Inputs are given from the keyboard using Putty. The output is observed on the 7-segment LED Display HEX5 and HEX4. HEX5 is showing the upper half of the data, while on HEX4 the lower half of the data. Steps were repeated for multiple sets of data. The hex value displayed on 7-segment was matched with the ascii table.

The LEDG2-LEDG3 were used as LED for verifying the Frame and Parity error, respectively. The LED were OFF when data was sent from PC which indicated no error and data integrity is maintained.

6.3.3 UART IrDA Transmitter and Receiver Testing

The design for UART IrDA transmitter was instantiated with UART IrDA receiver. The connection was in such a way txir is passed through inverter and connected to rxir. The same test bench was used for simulation, as depicted in Appendix 7.5. Simulations are carried out, and the data in and data out were matched. Then the design of IrDA Transmitter was programmed to FPGA number 1. While the design of the IrDA receiver was programmed on FPGA 2. Both FPGA were kept in line of sight of IR LED. Reset was given to FPGA1 then using SW8-SW2, data bits were allocated. Using Key2, the data was transmitted from FPGA1 to FPGA2. FPGA2 7-segment display showed the output. The data sets were matched, and parity and frame error were checked. Data matched with input data. No frame or parity error detected as the LED were OFF. Multiple data set were sent from FPGA1. All the data were accurately detected by FPGA2. All data set had data integrity verified.

7 Appendix

7.1 Verilog code for UART Transmitter

```
module uart_tx(baud_irda,txd,send,data,CLOCK_50,reset);

output txd;

output baud_irda;

input [6:0]data;

input send,CLOCK_50,reset;


wire [9:0]shift_val;

wire inc,send_sync,baud_full,load,shift,reset_b_counter,bdone;

wire reset_sync,reset_baud_b;

parameter PISO = 1'b1, SIPO = 1'b0,DATA_WIDTH=7,SHIFT_DATA_WIDTH=DATA_WIDTH+3;
```

synchronizer sync1(.d(send),.q(send_sync),.clk(CLOCK_50),.reset(reset_sync)); //reset to be synchronized as it is a switch on fpga

synchronizer sync2(.d(reset),.q(reset_sync),.clk(CLOCK_50),.reset(reset)); //send needs to be sync. as its a key on fpga

//instantiate the controller and connect it to bit counter, baud generator, shift register

uart_tx_controller
u2(.increment(inc),.load(load),.shift(shift),.reset_b_counter(reset_b_counter),.reset_baud_b(reset_baud_b),.send(send_sync),.baud(baud_full),.b_done(bdone),.clk(CLOCK_50),.reset(reset_sync));

//baud generator uses system clk and reset from controller

baud_gen
b1(.clk(CLOCK_50),.reset(reset_baud_b),.baud_full(baud_full),.baud_half(),.baud_irda(baud_irda)); //baud irda passed as output for irda modulator, half baud floating, baud full to controller

bit_counter bc(.b_done(bdone),.increment(inc),.clk(CLOCK_50),.reset(reset_b_counter)); //b_done to be connected to controller, use system clk and reset

parity_gen #(.DATA_WIDTH(DATA_WIDTH)) p1 (.shift_val(shift_val),.data(data)); //take the data from the fpga and output the packet to shift register

//shift register uses system clk and reset , load and shift from controller, output is txd

shift_reg_bimode_uart #(.SHIFT_DATA_WIDTH(SHIFT_DATA_WIDTH)) s1 (.txd(txd),.rx_data(),.load(load),.shift(shift),.data(shift_val),.clk(CLOCK_50),.mode(PISO),.reset(reset_sync) ,.rxd());

endmodule

7.2 Verilog Code for UART Receiver

module uart_rx(frame_err,parity_err,seg1,seg2,out_data,rxd,CLOCK_50,reset_b);

parameter DATA_WIDTH=7; //define the data width as parameter passed to submodules

parameter sipo=1'b0; // use sipo mode in shift register

output [DATA_WIDTH-1:0]out_data; //output of data

output frame_err,parity_err; //output the frame error and parity error on to leds

output [6:0]seg1,seg2; //outputs for 7 segment displays

input CLOCK_50,reset_b,rxd; //get system clk and uart rxd

```
wire baud_half,baud_full;

wire reset_baud_b;

wire load,shift,reset_bit_counter_b,increment,b_done;

wire [DATA_WIDTH+2:0]uart_rx_data;



seven_seg_decoder sg1(.data_out(seg1),.data_in(out_data[3:0])); //connect lower 4 bits to 7 segment display HEX4

seven_seg_decoder sg2(.data_out(seg2),.data_in({1'b0,out_data[6:4]})); //connect upper 3 bits to another 7 segment display HEX5, 4th bit is grounded

uart_rx_controller ctrl0(.increment(increment),.load(load),.shift(shift),.reset_bit_counter_b(reset_bit_counter_b),.reset_baud_b(reset_baud_b),


.rxd(rxd),.baud_half(baud_half),.baud_full(baud_full),.b_done(b_done),.reset_b(reset_b),.clk(CLOCK_50)); //connect all the controller outputs to respective blocks and connect all inputs coming towards controller

shift_reg_bimode_uart #(.SHIFT_DATA_WIDTH(DATA_WIDTH+3))s1(.txd(),.data(),.rx_data(uart_rx_data),.load(load),.shift(shift),.reset(reset_b),.clk(CLOCK_50),.rxd(rxd),.mode(sipo)); //make mode = sipo for shift reg and connect to controller and system clk and reset

baud_gen bg1(.baud_half(baud_half),.baud_full(baud_full),.clk(CLOCK_50),.reset(reset_baud_b)); //connect baud gen to controller and system clk. reset controlled by uart controller

bit_counter #(.COUNT_VALUE(DATA_WIDTH+3)) bc1 (.b_done(b_done),.increment(increment),.clk(CLOCK_50),.reset(reset_bit_counter_b)); //connect bit counter to controller and system clk. reset controlled by uart controller

error_detector_uart #(.DATA_WIDTH(DATA_WIDTH))err1(.frame_err(frame_err),.parity_err(parity_err),.data(out_data),.uart_data(uart_rx_data),.clk(CLOCK_50)); //connect the error detector to 7 segment display

endmodule
```

7.3 Verilog code for UART transmitter with IrDA modulator

```
module uart_irda_tx(txir,data_in,send,CLOCK_50,reset_b);

output txir;

input [6:0]data_in;
```

```
input send;

input CLOCK_50,reset_b;

wire tx_d,baud_irda;

uart_tx
tx1(.baud_irda(baud_irda),.txd(tx_d),.send(send),.data(data_in),.CLOCK_50(CLOCK_50),.reset(reset_b)); //use send key and data from fpga and connect txd,baud_irda to irda modualator

irda_modulator ir1(.txir(txir),.baud_irda(baud_irda),.txd_in(tx_d)); //connect output txir to ir led and take input from uart transmitter

endmodule
```

7.4Verilog code for UART IrDA receiver with IrDA demodulator

```
module uart_irda_rx(frame_err,parity_err,seg1,seg2,data_out,CLOCK_50,reset_b,rxir);

output [6:0]seg1,seg2;

output wire [6:0]data_out;

output wire frame_err,parity_err;

input CLOCK_50,reset_b,rxir;

wire tx_d,baud_irda;

wire rxd;

irda_demodulator demod1(.rxd(rxd),.rxir(rxir),.CLOCK_50(CLOCK_50),.reset_b(reset_b)); //take the rxir as input from IR LED, connect the rxd to uart receiver. use system clk and reset

uart_rx
rx1(.frame_err(frame_err),.parity_err(parity_err),.seg1(seg1),.seg2(seg2),.out_data(data_out),.rxd(rxd),.CLOCK_50(CLOCK_50),.reset_b(reset_b)); //use the rxd demodulated from demoudulator and output data to 7 seg display.

Endmodule
```

7.5Verilog code for UART Transmitter Testbench

```
`timescale 1ns/1ps

module uart_tx_tb;//tb doesnt have input or outputs

//inputs are reg

reg [6:0]data;

reg send,CLOCK_50,reset;

//output are wires

wire txd,baud_irda;
```

```
initial
begin
//initialize all the inputs
CLOCK_50 =1'b0;
reset=1'b0;
send =1'b0;
data = 6'b0;

        #40 reset =1'b1; //remove reset after 40 ns
        data = $urandom(); //take random data from random generator
        #40 send = 1'b1; //send the uart data after 40 ns
        #40 send =1'b0; //user releases send key
        #400000 data = $urandom();send = 1'b1; //new data entered : take random from random generator and user pressed the send key
        #100 send =1'b0; //user releases the send key after 100ns.

end
always #10 CLOCK_50 = ~CLOCK_50; //20 ns period for 50 mhz clk cycle

uart_tx tx1(baud_irda,txd,send,data,CLOCK_50,reset); //instantiate uart transmitter
endmodule
```

8. References

[1] T. Instruments, “KeyStone Architecture Universal Asynchronous Receiver/Transmitter (UART) User Guide,” 2010. [Online]. Available: http://www.ti.com/lit/ug/sprugp1/sprugp1.pdf.

[2] J. Smith, *ELEC473 Digital System Design Assignment 1 Serial Communications*, 2019.

[3] M. C. M. Morris Mano, “Shift Registers,” in *Digital Design With an Introduction to the Verilog HDL*, PEARSON, 2013, pp. 258-266.

[4] M. C. M. Morris Mano, “Gate-Level Minimization,” in *Digital Design With an Introduction to the Verilog HDL,* PEARSON, 2013, pp. 106-108.

[5] E. Hub, “BCD to 7 Segment LED Display Decoder Circuit,” Electronics Hub, 6 July 2015. [Online]. Available: https://www.electronicshub.org/bcd-7-segment-led-display-decoder-circuit/. [Accessed 28 November 2019].

[6] Altera, “Using the 7-segment Display,” in *DE2 Development and Education Board User Manual,* Altera, 2007, p. 30.

[7] Microchip, “MCP2155 IrDA® Standard Protocol Stack Controller Supporting DCE Applications,” 2001-2013. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/21690B.pdf.

[8] Agilent, “HSDL-3201 IrDA® Data 1.4 Low Power Compliant,” 2007.

[9] “ASCII Table,” [Online]. Available: http://www.asciitable.com/.