UART COMMUNICATION PROTOCOL

USING SYSTEMVERILOG

Ishraq Tashdid
Hardware Verification Trainee Engineer
Dynamic Solution Innovators.

28 FEBRUARY, 2022.

Table of Contents

ABOUT THIS MANUAL

The Universal Asynchronous Receiver/Transmitter (UART) performs serial-to-parallel conversions on data received from a peripheral device and parallel-to-serial conversion on data received from the CPU. The UART includes control capability and a processor interrupt system that can be tailored to minimize software management of the communications link.

This manual first looks at the theory and idea behind the UART communication protocol, breaking down the design and data format of the transmitter and receiver and the architecture behind them. It next discusses the design and state digram of the UART module. The state diagrams are kept simple so that the reader may find it easy to understand and implement the idea by themselves in any hardware descirption language (HDL).

The manual then discusses regarding the testbench for the designed UART module. The testbench is written in SystemVerilog, similar to the design module. Testbench examples are given and shown with steps for a better understanding for the reader. Waveform diagrams are also added so that the reader may find it easier to visualize the solution.

In the end, some error cases and their outputs were shown to explain possible error cases during the operation of the UART module. All the necessary codes are given in Appendix A along with the github link of the module.

1. INTRODUCTION

UART, or universal asynchronous receiver-transmitter, is one of the most widely used device-to-device communication protocols. This document demonstrates how to utilize UART as a hardware communication protocol by following the conventional process. When correctly configured, UART can function with a wide range of serial protocols that include delivering and receiving serial data.

UART is a hardware communication mechanism that employs asynchronous serial transmission at a variable speed. Asynchronous indicates that there is no clock signal to synchronize the transmitting device's output bits to the receiving end. Data is sent bit by bit via a single line or wire in serial transmission. Two wires are used in two-way communication for effective serial data transmission. Serial communications need fewer hardware and cables, depending on the application and system requirements, which saves implementation costs.

UART is primarily used as a device-to-device hardware communication protocol in embedded systems, microcontrollers, and computers. UART is one of the few communication technologies that employs only two wires for delivering and receiving data. Despite being a commonly used hardware communication protocol, it is not always properly optimized. When employing the UART module inside the microcontroller, proper frame protocol implementation is sometimes overlooked.

Figure 1: UART UART 44-Pin PLCC, TL16C550CFN from Texas Instruments.

The aim of this project was to create a system that follows the basic UART communication protocol. The vision was to start with a very basic communication protocol that required understanding the theory behind the system as well as implementing in a standard hardware description language (SystemVerilog in this case). Although the recreation of the UART protocol has been done in a very simple manner, there is a plan to add more functionalities to the system in the future. After understanding the theory, we implemented the instructions logically and put necessary comments for the reader to understand. The transmitter and the receiver both were designed with the same baud rate. Although actual UART communication systems are asynchronous in behavior, in this project we simplified by keeping them synchronous i.e. having the same clock speed. It enabled us to achieve a better and simplified way to communicate among the devices.

2. DESIGN SPECIFICATION

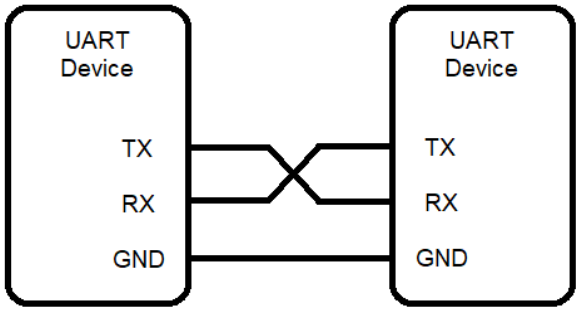

Figure 2: Simplified diagram of a UART communication protocol.

2.1. TRANSMITTER

The transmitter of the UART in this design includes an input pin to take inputs from the data bus. The UART transmitter behaves in FIFO mode. Based on the input taken from the input pin the UART sends the following data stream to the receiving device:

- Start bit as 0 to indicate the initialisation of data.
- Input data bits (8 bit data taken from the data bus).
- Stop bit as 1 to indicate the termination of data stream.

Some UART protocols include parity bit to add a layer of fault detection but we did not include such to keep the design simple.

2.2. RECEIVER

The receiver of the UART takes the serial data as input and converts to 8 bit data or 1 byte. It ignores the the start and stop bits. In case of parity bits, it checks the data bits to match with the parity bit. However, as mentioned before, parity bits are skipped in this implemetation of the UART protocol. The received data stream is:

- Start bit 0 is monitored to initiate the receiving system.
- Input data bits (8 bit serial data taken from the transmitter).
- Stop bit 1 is ignored and the 8 bit data is converted to byte.

2.3. DATA FORMAT

The UART transmits/receives the data in the following format:
1 Start bit + 8 bit Data bits + 1 Parity bit (Optional) +1 Stop bit.

Figure 3: UART Data Format.

2.4. TABLE OF SIGNALS, VARIABLES AND CONSTANTS

2.4.1. TRANSMITTER

NAME OF THE SIGNAL/ VARIABLE/ CONSTANT	TYPE	EXPLANATION
din	Input	Used as an input pin to take the input data bits from the bus.
wr_en	Input	Used to load the data from the data bus to the input of the transmitter.
txclk	Input	Clock for the transmitter.
txclken	Input	Clock enable pin.
tx	Output	Tranmission wire to receiver.
tx_busy	Output	Tranmistter busy signal.
data	Reg	To store input data.
bitpos	Reg	To indicate the index position of data register.
state	Reg	Used to indicate the current state of the system
STATE_IDLE	Parameter	First state of the FSM, default state.
STATE_START	Parameter	Second state of the FSM, adds start bit 0
STATE_DATA	Parameter	Third state of the FSM, sends data bit to the receiver.
STATE_STOP	Parameter	Final state of the FSM, adds stop bit 1 to the bit stream.

2.4.2. RECEIVER

NAME OF THE SIGNAL/ VARIABLE/ CONSTANT	TYPE	EXPLANATION
rx	Input	Used to take serial input from the transmitter.
rxclk	Input	Clock for the receiver.
rxclken	Input	Clock enable pin.
dout	Output Reg	Stores converted byte data.
CLOCKS_PER_BIT	Parameter	Clock time needed for each bit.
tx_busy	Output	Tranmistter busy signal.
counter	Reg	Used for sampling the received data.
bitpos	Reg	To indicate the index position of data register.
state	Reg	Used to indicate the current state of the system
RX_STATE_IDLE	Parameter	First state of the FSM, default state.
RX_STATE_START	Parameter	Second state of the FSM, checks for start bit 0.
RX_STATE_DATA	Parameter	Third state of the FSM, converts serial data to 1 byte.
RX_STATE_STOP	Parameter	Final state of the FSM, checks for stop bit 1.

2.5. DESIGN LANGUAGE AND TOOLS USED

The design language for this project was SystemVerilog. The tool used for this project was ModelSim*-Intel® FPGA Starter Edition, Version-19.1 on a Linux Operating System based computer.

3. DESIGN DIAGRAM

3.1. BLOCK DIAGRAM

The following shows the block diagram of the design under test (DUT).

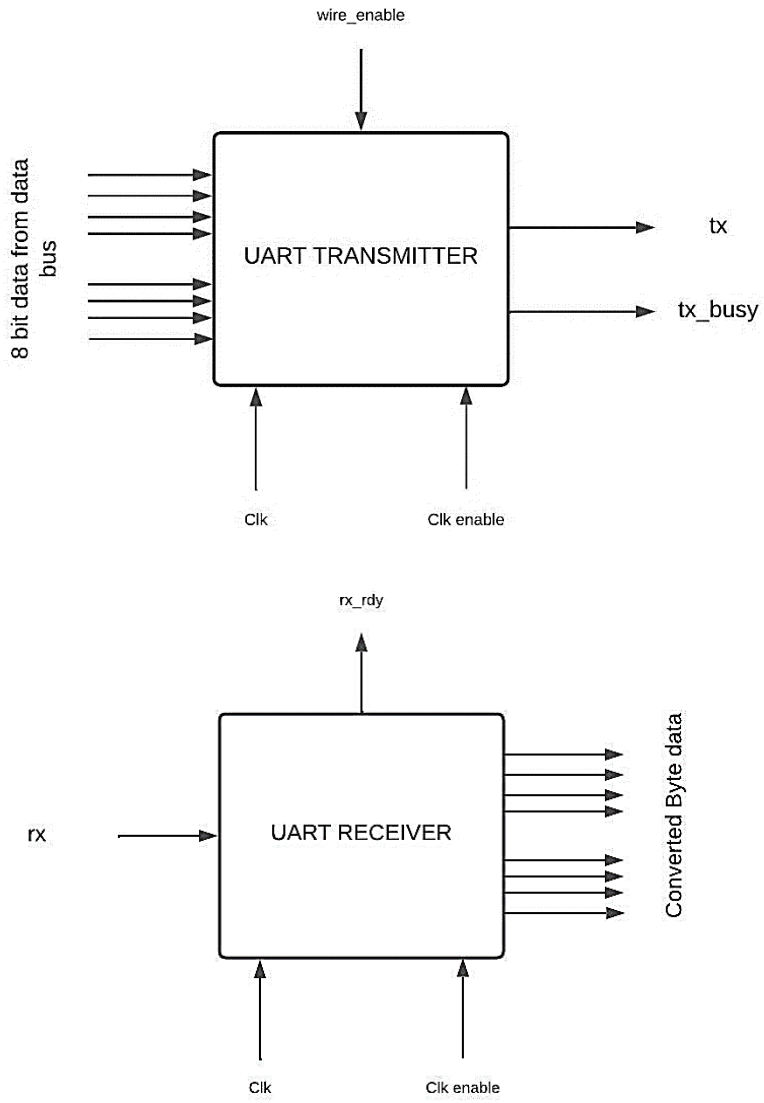

Figure 3: Block Diagram of DUT: Transmitter (top) and Receiver (bottom).

3.2. STATE DIAGRAM

The following shows the state diagram of both the UART transmitter and receiver:

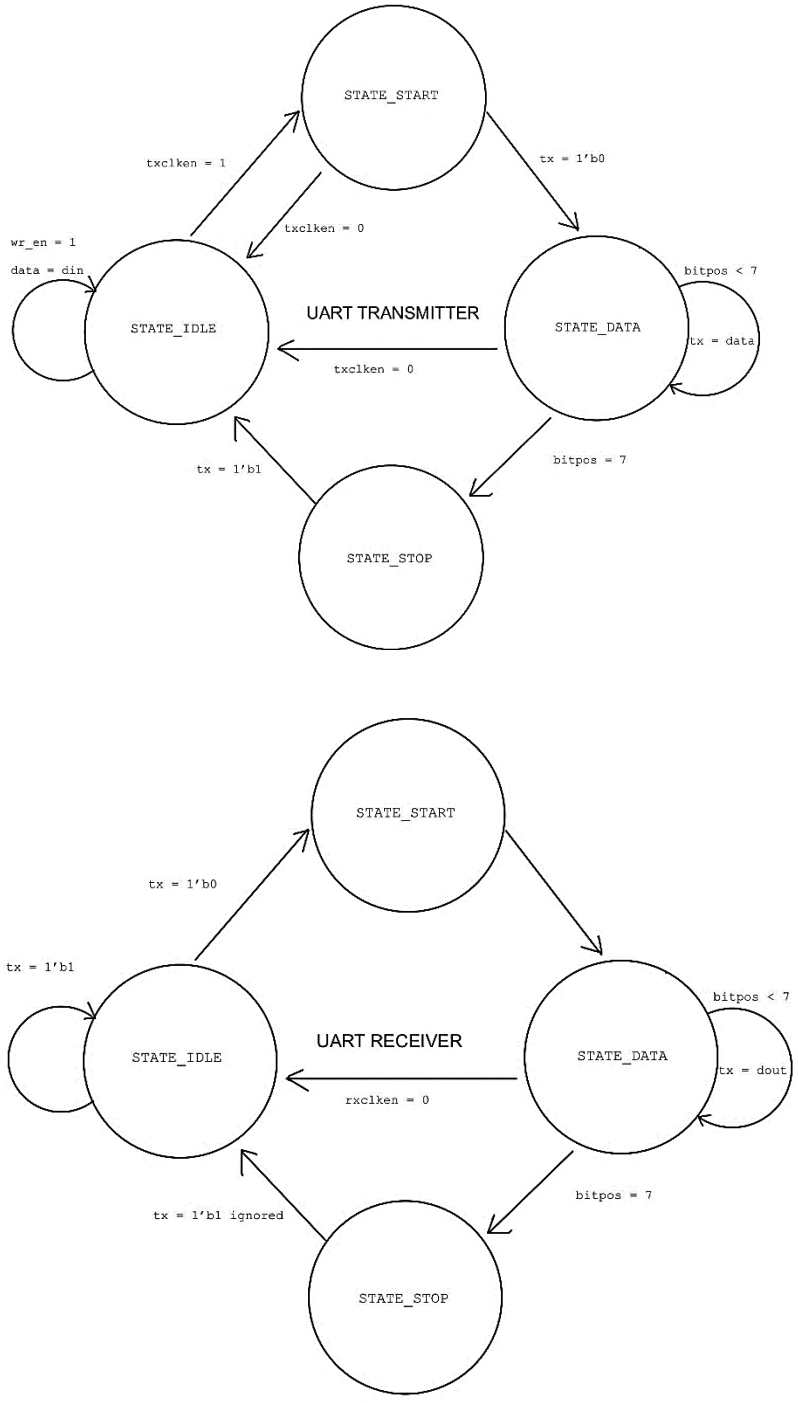

Figure 4: State Diagram of DUT: Transmitter (top) and Receiver (bottom).

4. VERIFICATION

4.1. VERIFICATION PLAN

The verification of a UART communication module is very straightforward. We have to monitor whether the byte sized data is converted to serial bit stream in the transmitter and whether that serial output (now input to the receiver) is converted back to byte sized data. The verification task was divided into few steps:

- **Transmitter**
  - Byte data is taken as input.
  - Clock is generated, clock pin is enabled.
  - Wire is enabled to take the byte data into the designated register (data).
  - Output is checked whether data is serially sent with 0 as start bit and 1 as stop bit with tx_busy high during the whole operation.

Similarly, the receiver block is also tested for verification. Steps for the receiver block is as follows:

- **Receiver**
  - Serial data is taken as input from transmitter.
  - Clock is generated, clock pin is enabled.
  - Output is checked whether data is converted to byte sized packet.
  - rx_busy is high during the operation period.

4.2. VERIFICATION METHODOLOGY

For verification, a testbench code in Systemverilog language was written with proper inputs. The transmitter block and the receiver block was at first verified individually. Note that, since the receiver side has a sampling speed, intentional delay was added in the receiver testbench code in accordance with the sampling rate. The testbench output was observed for different patterns of input data and showed succesful results in all of the cases. State changes were monitored as maintained by the design code. No anomaly was found in the testbench during this phase.

Next both the transmitter and receiver module was verified together under one testbench. In this case, the transmitter and receiver had to coordinate between eachother with their data. The transmitter transmitted in accordance with the baud rate and the receiver sampled each data and converted to a byte packet. There was no delay manually added to the testbench this time as the transmitter was sending data with proper baud rate.

4.3. VERIFICATION LANGUAGE AND TOOLS USED

The verification language for this project was SystemVerilog. The tool used for this project was ModelSim*-Intel® FPGA Starter Edition, Version-19.1 on a Linux Operating System based computer.

4.4. FUNCTIONAL VERIFICATION

Functional verification was done on the basis of ideal scenario with different input patterns. In all cases the wire enable pin was kept high, clock was also enabled. The inputs included different corner cases like 1010 1010, 1111 0000, 0000 0000, 1111 1111 etc. In all cases, the receiver was able to regenerate the bit stream to byte data. Sample code of the functional verification is shown in Appendix A.

4.5. ERROR CASE VERIFICATION

Error case verification and correction is one of the most essential part of the complete design of a module. Error cases help find the blind spots for the design module. In this part, all the inputs were randomly tested with *$random.* After that, some special cases were tested.

- Case 1 – No Wire enable

In this case, the data was generatedthrough the testbench and the clock was also enabled. However, the data is taken only when the wire is enabled. Therefore, the transmitter does not take the data.

- Case 2 – Clock is not enabled

In this case, the clock is generated, but the pin is not enabled. Wire is also enabled, so that the input is taken into the transmitter. But since the clock enable pin is not high, the transmitter always remains in the idle state. Although it checks if the clock enable is high when it goes to the start state, but since the enable pin remains low, it returns back to idle state.

- Case 3 – Receiver clock not enabled

In this scenario, the receiver clock is made low at first although the data is being transferred from the transmitter to the receiver and after some delay the pin is made high. Therefore, as a result, the data does not pass properly.

- Case 4 – Proper input

This is a continuation of case 3, where a proper input is given (similar to functionality check) but as the receiver is processing the previous input (expects 10 bit data from the transmitter), the input in this case is not passed properly through the receiver.

- Case 5 – Solve for case 4

Here, everything is reset and the receiver is made to wait to finish all the pending tasks it was doing. Then everything is generated properly through the testbench and the desired output is found.

5. VERIFICATION DIAGRAM

5.1. BLOCK DIAGRAM

The following shows the block diagram of the testbench along with the DUT.

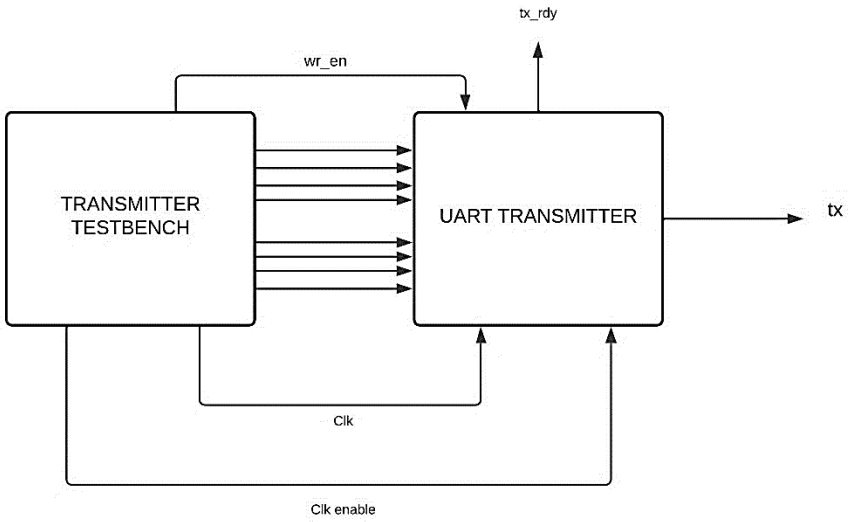

Figure 5: Block Diagram of transmitter testbench environment.

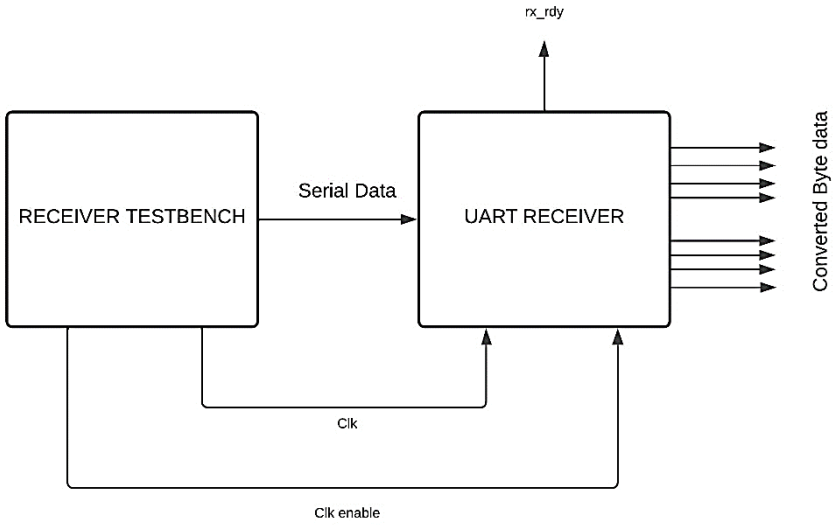

Figure 6: Block Diagram of Receiver testbench environment.

5.2. DATA FLOW DIAGRAM

The following shows the data flow diagram of the overall testbench and the design under test (DUT). The clock and input data is generated from the testbench along with the enable pins are made high. The transmitter adds necerssary additional bits (start bit and stop bit) and receiver detects the start bit and converts data to byte.

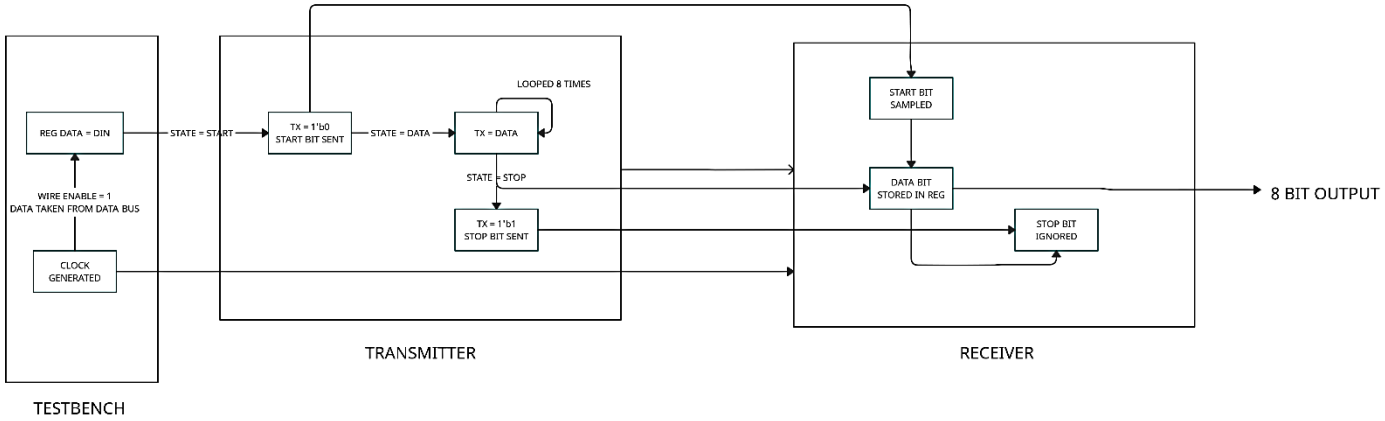

Figure 7: Data flow diagram of the overall design module including testbench environment.

6. VERIFICATION RESULTS

6.1. FUNCTIONAL TEST CASE SCENARIO

For the verification of the functional test case, ModelSim simulation tool was used as mentioned before. For the functional verification, the testbench was given proper inputs to drive the DUT. First the transmitter and the receiver was driven separately with the testbench.

6.1.1. TRANSMITTER TESTBENCH

For the transmitter, the clock was generated from the testbench and clock enable was made high. An 8 bit data was given to the DUT and the wire was enabled to pass the data through to the transmitter. The output was observed to find serial input of the given data. Start bit (0) and stop bit (1) were also checked whether they were added to the data stream.

TESTBENCH CODE FOR TX

```
`timescale 1ns / 10ps
`include "transmitter.sv"
module txtb();
reg [7:0] din;
reg txclk;
reg wr_en;
reg txclk_en;
wire tx;
wire tx_busy;

transmitter DUT (     .din(din),
                      .txclk(txclk),
                      .tx(tx),
                      .wr_en(wr_en),
                      .txclken(txclk_en),
                      .tx_busy(tx_busy));

initial begin
        forever #1 txclk = ~txclk;
end

initial begin
       txclk=1'b0;
       din= 8'b01100011;
       wr_en=1'b1;
       txclk_en=1'b1;
#11000;

$stop;
end
endmodule
```

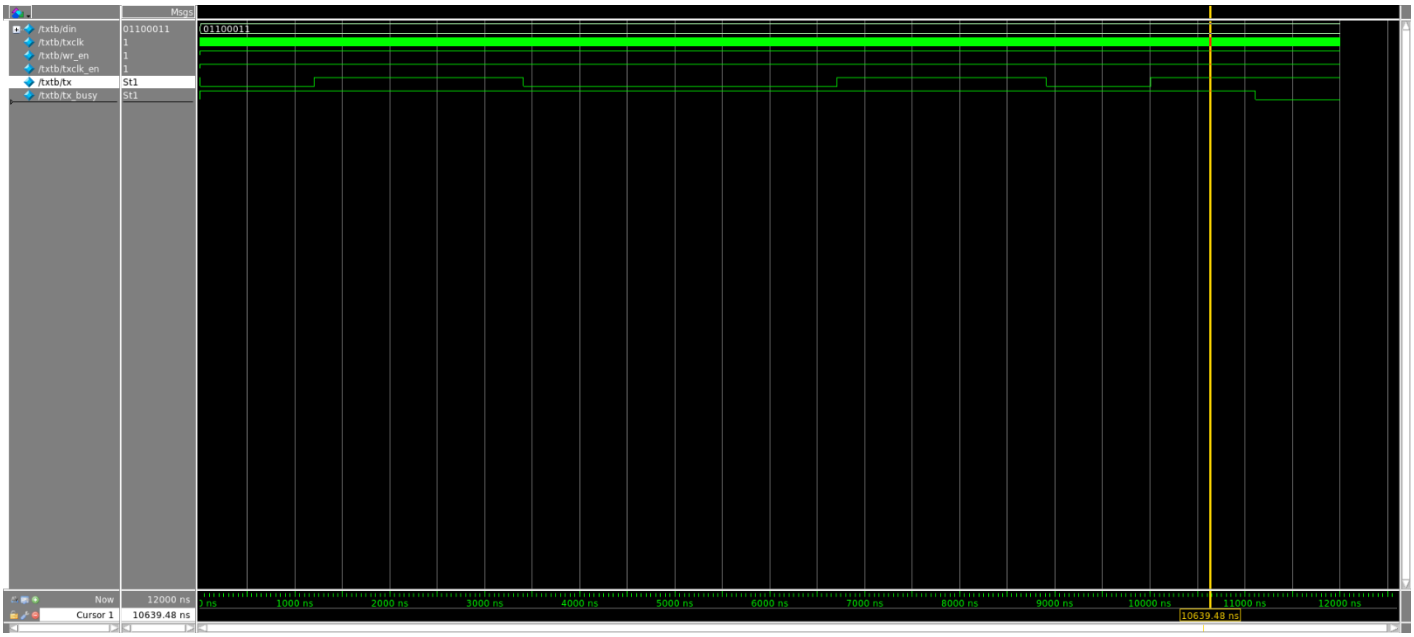

Figure 8: Waveform of the transmitter testbench.

In this figure, we can see that the transmitter takes the input 0110 0011, and the *tx_busy* is kept high during the process. After passing the input through the *tx* pin, the *tx_busy* becomes low. The clock enable pin is kept high throughout the process.

6.1.2. RECEIVER TESTBENCH

For the receiver part, clock was again generated the enable pins were made high. State transitions were monitored closely whether the correct transitions are made in the right time. Since the receiver is sampled at a certain frequency, intentional delay was added manually so that the receiver samples the data properly. Although this was not necessary when the whole module, transmitter and reciever, both were tested togehther in the next section. Finally, the output is checked to see whether the bit stream was converted to byte packet.

TESTBENCH CODE FOR RX

```
`timescale 1ns / 10ps
`include "receiver.sv"
module rxtb();
        reg rx;
        //wire rdy;
        //reg rdy_clr;
        reg rxclk;
        reg rxclken;
        wire [7:0] dout;
        reg rx_busy;
        receiver DUT ( .rx(rx),
                                    //.rdy(rdy),
                                    //.rdy_clr(rdy_clr),
                                    .rxclk(rxclk),
                                    .rxclken(rxclken),
                                    .rx_busy(rx_busy),
                                    .dout(dout));
        initial begin
          forever #1 rxclk = ~rxclk;
        end
```

```
        initial
 begin
          rxclk=1'b0;
          //rdy_clr=1'b1;
          #10;
          rxclken=1'b1;
          #10;
          rx=1'b0; //start bit
          #1200;
          rx=1'b1; //bit 1
          #1300;
          rx=1'b0;//bit 2
          #1200;
          rx=1'b1;//bit 3
          #1100;
          rx=1'b0; //bit 4
          #1100;
          rx=1'b1;//bit 5
          #1100;
          rx=1'b1;//bit 6
          #1100;
          rx=1'b0; //bit 7
          #1100;
          rx=1'b0;//bit 8
          #1100;
          rx=1'b1;//Stop bit
          #1200;
$stop;
end
endmodule
```

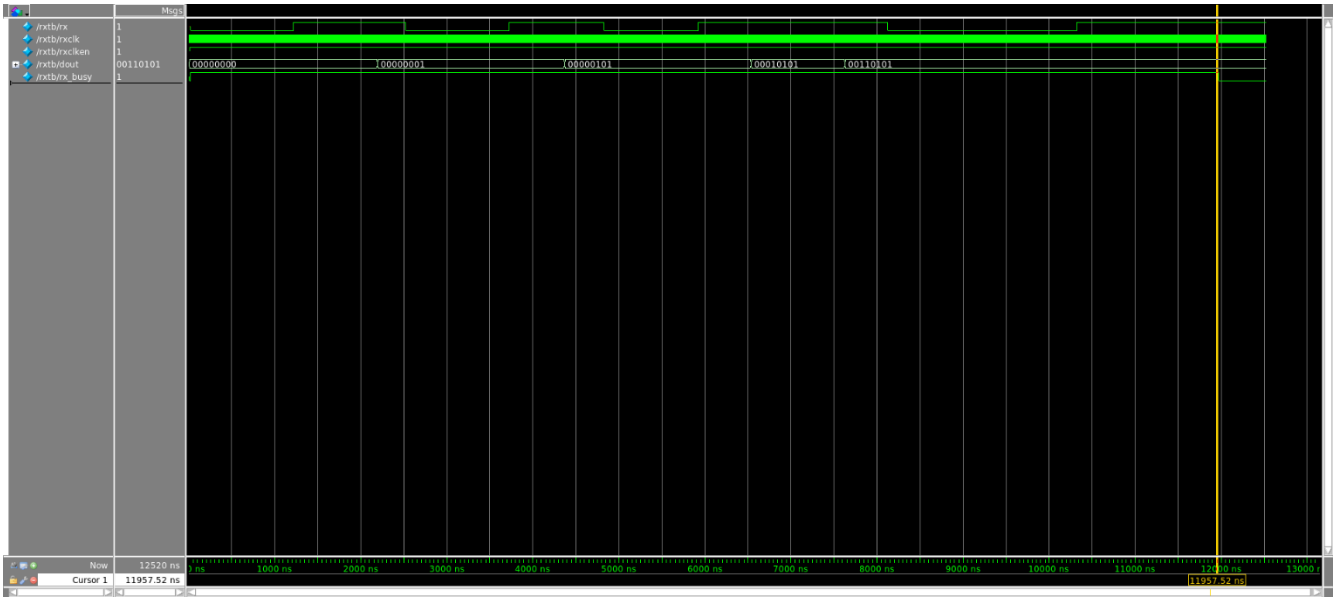

Figure 9: Waveform of the receiver testbench.

In this figure, the receiver was given 0 0011 0101 1 with 0 and 1 being the start and stop bit (at the start and end respectively). As it can be seen that the receiver takes the input and converts them to byte packet and outputs in the *dout* pin.

6.1.3. UART MODULE TESTBENCH

Finally, the complete UART module was tested to find proper collaboration between them. The UART module was checked to see if the given byte data was properly recovered after passing through the transmitter and receiver. Various input cases were tested to make sure the module worked perfectly.

TESTBENCH CODE FOR UART

```
`timescale 1ns / 10ps


//`include "uart.sv"
`include "transmitter.sv"
`include "receiver.sv"
module tb();
                  reg [7:0] din;     //input data
                  reg wr_en;         //Wire enable pin (allows data input)
                  reg clk;           //A 1GHz Clock
                  reg txclken;              //Clock enable pin
                  wire tx;           //Transmission wire
                  reg tx_busy ;      //Transmission busy
                  //reg rx;
                  //output reg rdy,
                  //input wire rdy_clr,
                  reg rx_busy;
                  reg rxclken;
                  wire [7:0] dout;


transmitter DUT_tran (              .din(din),
                                    .txclk(clk),
                                    .tx(tx),
                                    .wr_en(wr_en),
                                    .txclken(txclken),
                                    .tx_busy(tx_busy));
```

```
receiver DUT_rec (                  .rx(tx),
                                    //.rdy(rdy),
                                    //.rdy_clr(rdy_clr),
                                    .rxclk(clk),
                                    .rxclken(rxclken),
                                    .rx_busy(rx_busy),
                                    .dout(dout));


initial begin
         forever #1 clk = ~clk;
end


initial begin
        clk=1'b0;
        din= 8'b01100011;
        wr_en=1'b1;
        txclken=1'b1;


        rxclken=1'b1;


#11000;
$stop;
end
endmodule
```

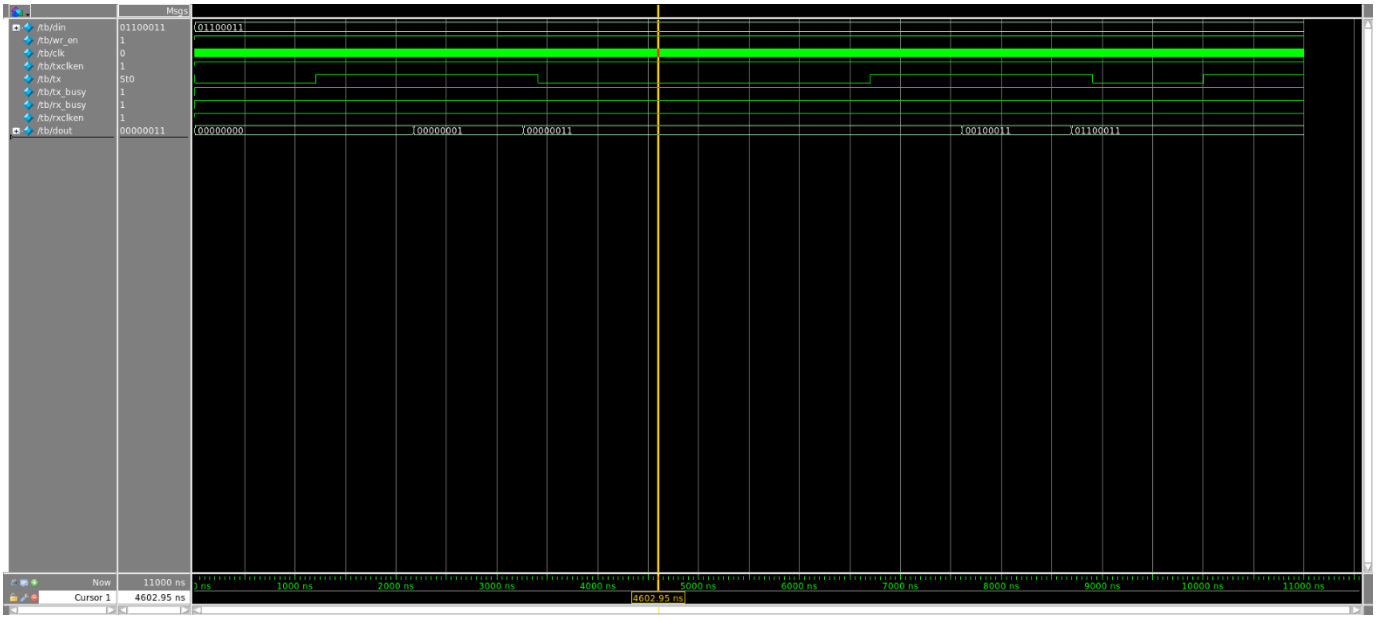

Figure 10: Waveform of the total UART testbench.

In this figure, the UART module was given 0110 0011 as an input. As it can be seen that the clock is generated, *tx pin* passes the output as an input to the receiver module. The receiver then converts this to a packet. As it can be seen in the *dout* wave pattern that it gradually changes from 0000 0000 to 0110 0011. So the output passes successfully.

6.2. ERROR TEST CASE SCENARIO

For error test case, we tested 5 scenarios. The error cases for selected to test for the corner and undesired cases and the design module was adjusted accordingly to tackle the error cases. Explanation regarding the strategy was previously discussed in section 4.5.

Case 1 – No Wire enable

PIN NAME	VALUE	DESCRIPTION
din	0110 0011	Proper input is given for the transmitter.
wr_en	0	Wire is not enabled.
txclk	1/0	Clock is generated through testbench.
txclken	1	Clock is enabled.

Case 2 – No Transmitter Clock enable

PIN NAME	VALUE	DESCRIPTION
Din	0110 0011	Proper input is given for the transmitter.
wr_en	1	Wire is enabled.
Txclk	1/0	Clock is generated through testbench.
Txclken	0	Clock is not enabled.

Case 3 – No Receiver Clock enable

PIN NAME	VALUE	DESCRIPTION
din	0110 0011	Proper input is given for the transmitter.
wr_en	1	Wire is enabled.
txclk	1/0	Clock is generated through testbench.
txclken	1	Clock is enabled.
rxclken	0/1	Kept low at first then made high.

Case 4 – Proper input given

PIN NAME	VALUE	DESCRIPTION
din	0110 0011	Proper input is given for the transmitter.
wr_en	1	Wire is enabled.
txclk	1/0	Clock is generated through testbench.
txclken	1	Clock is enabled.
rxclken	1	Clock is enabled.

Case 5 – Corrected for the receiver error

PIN NAME	VALUE	DESCRIPTION
din	xxxx xxxx	No input given.
wr_en	0	Wire is not enabled.
txclk	1/0	Clock is generated through testbench.
txclken	0	Clock is not enabled.
rxclken	0	Clock is not enabled.
DELAY FOR 20000 CLOCK CYCLE FOR THE RECEIVER PROCESSING TO CLEAR OUT.		
din	0110 0011	Proper input is given for the transmitter.
wr_en	1	Wire is enabled.
txclk	1/0	Clock is generated through testbench.
txclken	1	Clock is enabled.
rxclken	1	Clock is enabled.

ERROR CASE VERIFICATION CODE FOR UART

```
`timescale 1ns / 10ps
//`include "uart.sv"
`include "transmitter.sv"
`include "receiver.sv"



module errorcase1();
                  reg [7:0] din;     //input data
                  reg wr_en;         //Wire enable pin (allows data input)
                  reg clk;           //A 1GHz Clock
                  reg txclken;               //Clock enable pin
                  wire tx;           //Transmission wire
                  reg tx_busy ;      //Transmission busy



                  //reg rx;
                  //output reg rdy,
                  //input wire rdy_clr,
                  reg rx_busy;
                  reg rxclken;
                  wire [7:0] dout;




transmitter DUT_tran (               .din(din),
                                     .txclk(clk),
                                     .tx(tx),
                                     .wr_en(wr_en),
                                     .txclken(txclken),
                                     .tx_busy(tx_busy));
```

```
receiver DUT_rec (                  .rx(tx),
                                    //.rdy(rdy),
                                    //.rdy_clr(rdy_clr),
                                    .rxclk(clk),
                                    .rxclken(rxclken),
                                    .rx_busy(rx_busy),
                                    .dout(dout));
initial begin
         forever #1 clk = ~clk;
end
initial begin
        clk=1'b0;
        //wire not enabled
        din= 8'b01100011;
        wr_en=1'b0;
        txclken=1'b1;



/* explanation of the output
_________________________________
not even taking the input "d_in" into the transmitter. As the input
is depended on wire_enable so doesn't get transferred from the bus to the
transmitter without it.
*/


#1200;
```

```
        din= 8'b01100011;
        wr_en=1'b1; //wire is enabled this time
        txclken=1'b0; //clock_enable not enabled
/* explanation of the output
_____________________________
Input is taken. State is initiated as "IDLE".
However, as it changes state from "IDLE" to "START BIT" (as a result tx_busy turns 1).
But as the clock enable pin is 0 (not enabled), it keeps the din in the data register of the
transmitter and returns to IDLE state and waits for the next clock cycle to check whether
clock_enable has become 1 (tx_busy flickers from 1 to 0 as a result).
*/
#1200;

        clk=1'b0;
        din= 8'b11110000;
        wr_en=1'b1;
        txclken=1'b1;

        rxclken=1'b0; // receiver clock not enabled
#2000;
        rxclken=1'b1; //receiver clock enabled after 2000 clock cycle



#11000;

/* explanation of the output
_____________________________
Output doesn't fully pass through. Partially passes through as the clock is enabled after
2000 clock cycle the receiver receives error in the transmission.
*/
```

```
        din= 8'b11001100;
        wr_en=1'b1;
        txclken=1'b1;
        rxclken=1'b1;
#11000;

/* explanation of the output
_____________________________
receiver is still processing the last output. fails to identify new input.
Solve : reset the transmitter and receiver by making the wire and enable clock pins as 0.
Wait for a bit for the running processes to end.
*/
        din= 8'bxxxxxx;
        wr_en=1'b0;
        txclken=1'b0;
        rxclken=1'b0;
#20000;

        din= 8'b01100011;
        wr_en=1'b1;
        txclken=1'b1;
        rxclken=1'b1;

/* explanation of the output
_____________________________
Output works fine as desired.
*/

#11000;
$stop;

end
endmodule
```

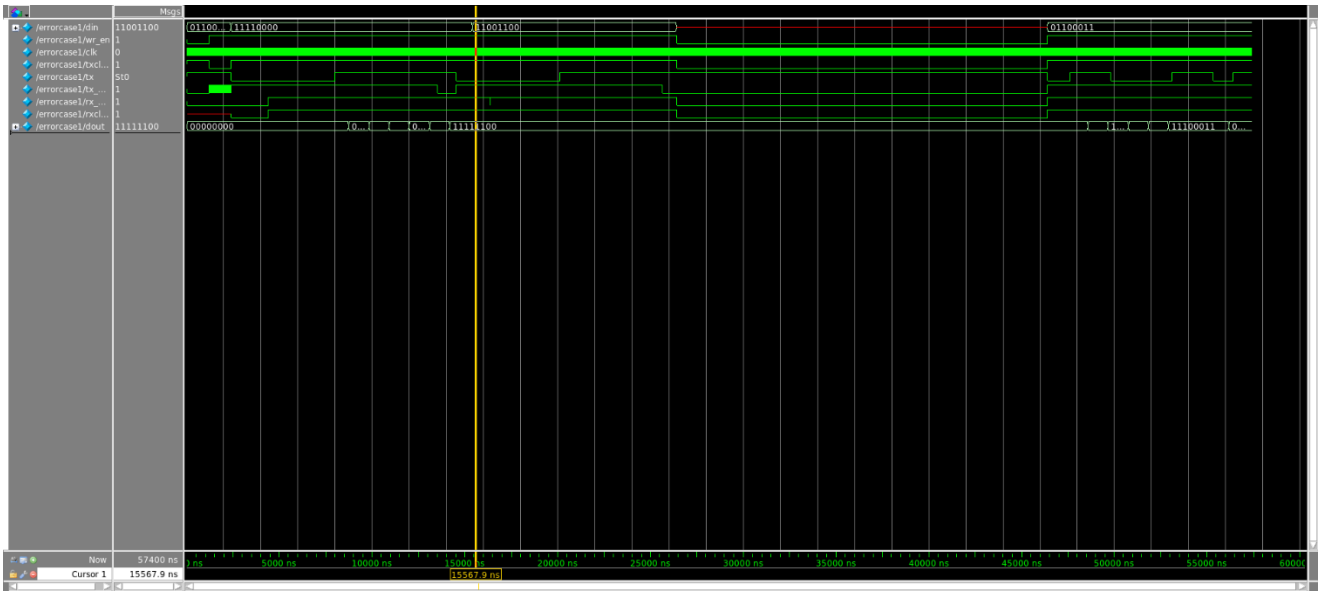

Figure 11: Waveform of the error case verification of the total UART testbench.

From the output, we can see that for the first case, the transmitter does not take din as wr_en is kept low. *tx_busy* remains low indicating that the transmitter is not processing the input. After that, for the second case, wr_en is made high, however, txclken is made low. So, the input is taken into the transmitter but the trasnmitter does not process the value as clock pin is not enabled. The waveform from receiver output validates this statement as it remains 0000 0000.

Next, both of them are made high, but the rxclken pin is kept low. As a result, the receiver does not process the value being received by the transmitter. After a while, the receiver clock pin is enabled. But the result comes erronous because few bits have already been transmitted. Even if we follow with a correct arrangement, similar to the functionality check, the output is erronous.

To solve this, the data is reset by making all the pins low. Then after receiver is done processing, the inputs are made high again. In this scenario, the inputs are properly processed and and the receiver output waveform validates this statement.

APPENDIX A

TX DESIGN MODULE

```
`timescale 1ns / 10ps


module transmitter(input wire [7:0] din,    //input data
                   input wire wr_en,         //Wire enable pin (allows data input)
                   input wire txclk,         //A 1GHz Clock
                   input wire txclken,               //Clock enable pin
                   output reg tx,            //Transmission wire
                   output wire tx_busy);     //Transmission busy
initial begin
        tx = 1'b1;    //Line remains high during idle for fault detection
end
parameter STATE_IDLE  = 2'b00;       //default state
parameter STATE_START = 2'b01;       //start bit adding state
parameter STATE_DATA  = 2'b10;       //data transmission state
parameter STATE_STOP  = 2'b11;       //stop bit adding state
reg [7:0] data = 8'h00;
reg [2:0] bitpos = 3'h0;     //bit position index
reg [1:0] state = STATE_IDLE;//default state initialized with idle
always @(posedge txclk) begin       //positive edge triggered transmitter
       case (state)
       /*
       * The first state. Which is idle and goes to start when gets data from din.
       * Bit position is initialized with 0.
       */
```

```
        STATE_IDLE: begin
                if (wr_en) begin             //when the wire is enabled it takes the data as
input
                        state <= STATE_START; //next state
                        data <= din;          //data is taken
                        bitpos <= 3'h0;               //index=0
                end
        end
        /*
        * The second state.
        * start bit is added with 0
        */
        STATE_START: begin
                if (txclken) begin    //clock is enabled for data transmission start
                        tx <= 1'b0;    //start bit=0 (indicating the start of transmission
                        #1;
                        $display("start bit %b",tx);
                        #1200;
                        state <= STATE_DATA;  // next state is data state (where data is
transmitted)
                end
                else   begin                 //if somehow the clock is not enabled it goes
back to the idle state
                        state <=STATE_IDLE;
                        data <= 8'h00;
                        bitpos <= 3'h0;
                end
        end
        STATE_DATA: begin
                if (txclken) begin
                        if (bitpos == 3'h7)                 //when at the last position
                        state <= STATE_STOP;         //the next state is made to stop state
                        else
                        bitpos <= bitpos + 3'h1;     //index position is increased
```

```
                        tx <= data[bitpos];
                        $display("bit number %0d = %b",bitpos,tx);
                        #1100; /* this is outside the if-else condition so gets implemented
always,that's why we can make the index position == 7 and get away with it. */
                        end
                else begin//if somehow the clock is not enabled it goes back to the idle
state
                        state  <=STATE_IDLE;
                        data   <= 8'h00;
                        bitpos <= 3'h0;
                end
        end
        STATE_STOP: begin
                if (txclken) begin
                        tx <= 1'b1;
                        #1;            //stop bit is made 1
                        $display("stop bit %b",tx);
                        #1100;
                        state <= STATE_IDLE;
                        #1000; //goes back to idle state
                end
                else   begin  //if somehow the clock is not enabled it goes back to the idle
state
                        state  <=STATE_IDLE;
                        data   <= 8'h00;
                        bitpos <= 3'h0;
                end
        end
        default: begin
                tx <= 1'b1;
                state <= STATE_IDLE;
        end
        endcase
end
assign tx_busy = (state != STATE_IDLE);     //busy when not idle
endmodule
```

RX DESIGN MODULE

```
module receiver(input wire rx,
                  output wire rx_busy,
                  //input wire rdy_clr,
                  input wire rxclk,
                  input wire rxclken,
                  output reg [7:0] dout);
initial begin
       //rdy = 0;
       dout=8'b0;
end
parameter CLKS_PER_BIT              = 1085;               //clock=1GHz ;
baud rate= 115200x8 = 921600; clocks/bit = 1G/921600 = 1085
parameter RX_STATE_IDLE             = 2'b00;
parameter RX_STATE_START     = 2'b01;
parameter RX_STATE_DATA             = 2'b10;
parameter RX_STATE_STOP             = 2'b11;
reg [1:0] state       = RX_STATE_IDLE;
reg [15:0] counter    = 0;
reg [2:0] bitpos      = 0;
always @(posedge rxclk) begin
       case (state)
              RX_STATE_IDLE :       begin
              if (rxclken) begin
                     counter<= 0;                         //counter set to 0
                     bitpos  <= 0;                        //index position
                     if (rx == 1'b0)                      // Start bit detected
                            state <= RX_STATE_START;
                     else
                            state <= RX_STATE_IDLE;
              end
```

```
                else
                        state <= RX_STATE_IDLE;
                end
                RX_STATE_START: begin
                if (rxclken) begin
                                if (counter > (CLKS_PER_BIT-1)/2) begin
                                        if (rx == 1'b0) begin
                                                counter<= 0; // reset counter, found the middle
                                                state   <= RX_STATE_DATA;//move to the next
                                        end
                                        else   //anomaly detected, go back to idle state
                                                state <= RX_STATE_IDLE;
                                end
                                else begin
                                        counter <= counter + 1;      //increase counter
                                        state   <= RX_STATE_START;//stay on the current state
                                end
                        end
                else begin
                        state  <= RX_STATE_IDLE;
                        counter       <= 0;
                        bitpos <= 0;
                end
                end
RX_STATE_DATA : begin
                if (rxclken) begin
                        if (counter < (CLKS_PER_BIT-1)/2) begin //until counter reaches
middle sampling point
                                counter <= counter + 1; //increase counter
                                state   <= RX_STATE_DATA; //stay on the current state
                        end
```

```
                else begin            //when at middle sampling point


                        counter      <= 0;           //reset counter
                        dout[bitpos] <= rx;   //start converting to byte
                        // Check if we have received all bits
                        if (bitpos < 7) begin
                                bitpos <= bitpos + 1;        //increasing index
                                state  <= RX_STATE_DATA;
                        end
                        else begin
                                bitpos <= 0;   //received all dout as byte, set index=0
                                state  <= RX_STATE_STOP;             //go to next state
                        end
                end
        end
        else begin
                state  <= RX_STATE_IDLE;
                counter        <= 0;
                bitpos <= 0;
        end
        end // case: RX_STATE_DATA
        RX_STATE_STOP : begin
        if (rxclken) begin
                if (counter < CLKS_PER_BIT) begin
                        counter <= counter + 1;
                        state   <= RX_STATE_STOP;
                end
                else begin
                        //rdy          <= 1'b1;
                        counter        <= 0;
                        state          <= RX_STATE_IDLE;
                        $display("Output : %b",dout);
                end
        end
```

```
                else begin
                        state   <= RX_STATE_IDLE;
                        counter        <= 0;
                        bitpos  <= 0;



                end
                end



                default: begin
                        state <= RX_STATE_IDLE;
                end
                endcase



end


assign rx_busy = (state != RX_STATE_IDLE);  //busy when not idle


endmodule
```

RUN.DO FILE FOR SIMULATION

A run.do file to be run in transcript of ModelSim.

```
#compilation
vlog tb.sv
#simulation
vsim tb
#add wave
add wave -position insertpoint sim:/tb/*
run -all
```

GITHUB LINK

https://github.com/ishraqtashdid/UART

REFERENCES

1. https://www.ti.com/lit/ug/sprugp1/sprugp1.pdf?ts=1646035996483&ref_url=https%253A%252F%252Fwww.google.com%252F
2. https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter
3. https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html
4. https://www.circuitbasics.com/basics-uart-communication/
5. Serial Communication Protocols and Standards: RS232/485, UART/USART, SPI, USB, INSTEON, Wi-Fi and WiMAX (River Publishers Series in Communications) by Dawoud Shenouda Dawoud (Author), Peter Dawoud (Author).
6. Y. -y. Fang and X. -j. Chen, "Design and Simulation of UART Serial Communication Module Based on VHDL," 2011 3rd International Workshop on Intelligent Systems and Applications, 2011, pp. 1-4, doi: 10.1109/ISA.2011.5873448.
7. https://exploreembedded.com/wiki/LPC1768:_UART_Programming
8. https://www.nandland.com/vhdl/modules/module-uart-serial-port-rs232.html
9. https://medium.com/@chandulanethmal/uart-communication-link-implementation-with-verilog-hdl-on-fpga-b6e405c5cbd8
10. https://www.ijeat.org/wp-content/uploads/papers/v9i5/E1135069520.pdf
11. N. F. Mahat, "Design of a 9-bit UART module based on Verilog HDL," 2012 10th IEEE International Conference on Semiconductor Electronics (ICSE), 2012, pp. 570-573, doi: 10.1109/SMElec.2012.6417210.
12. https://excamera.com/sphinx/fpga-uart.html