→Design source code:-

```
`timescale 1ns / 1ps


module uart_top
#(
parameter clk_freq = 1000000,  //Mhz default frequency
parameter baud_rate = 9600
)
(
 input clk,rst,  //global signals
 input rx,
 input [7:0] dintx,
 input send,
 output tx,
 output [7:0] doutrx,
 output donetx,
 output donerx
   );

uarttx
#(clk_freq, baud_rate)
utx   //name of instance of module uarttx
(clk, rst, send, dintx, tx, donetx);   //connecting ports

uartrx
#(clk_freq, baud_rate)
rtx //name of instance of module uartrx
(clk, rst, rx, donerx, doutrx); //connecting ports

endmodule
```

```
//////////////////////////////////////////////////////////////////

module uarttx
#(
parameter clk_freq = 1000000,  //MHz default frequency
parameter baud_rate = 9600
)
(
input clk,rst,   // Global signals
input send,      // Signifies that user wants to send data when it becomes 1(high)
input [7:0] tx_data,  // Data we want to send on tx pin is given at this tx_data pinv.i.z 8 bit data pin
output reg tx,  //output port on which data is send serially during transmission of data
output reg donetx  //when transmission of data completes this signals becomes high indicating that transmission is completed
);

  localparam clkcount = (clk_freq/baud_rate); ///x helps in generation of clock signal for the specified baud rate

integer count = 0;
integer counts = 0;

reg uclk = 0; //uclk is kept in same state until and unless our count reaches clkcount/2 and
// after that it is inverted this is how we generate clock for specified baud rate v.i.z denoted by uclk

enum bit[1:0] {idle = 2'b00, start = 2'b01, transfer = 2'b10, done = 2'b11} state;

 ///////////uart_clock_generation
  always@(posedge clk) //sensitive to onboard clock
    begin
      if(count < clkcount/2)
```

```
      count <= count + 1;
    else begin
      count <= 0;
      uclk <= ~uclk;
    end
  end
 reg [7:0] din;
 ////////////////////Reset decoder
 always@(posedge uclk)//sensitive to uclk v.i.z clock signal required for requested baud rate
  begin
    if(rst)
    begin
      state <= idle; //deafault state is idle
    end
   else
   begin
   case(state)
     idle:    //idle is default state v.iz maitained until and unless data is being send or recieved
       begin
         counts <= 0;     //default value
         tx <= 1'b1;      //default value
         donetx <= 1'b0;  //default value

         if(send) //checking for send to get high which mean that we want to send data
         begin
           state <= transfer; //then at that point state changes to transfer
           din <= tx_data;  //we're registering data on din as we don't want to see temporary transitions
on tx
           tx <= 1'b0; //start condition in the next clock tick
         end
         else
```

```
          state <= idle;
        end

     transfer: begin
       if(counts <= 7) begin   // count here is given of 7 from 0 to 7 and with each clock cycle of uclk count
       //and din changes and gets transferred to tx serially with indices and as soon as count exceeds 7 we go to else statement
         counts <= counts + 1;
         tx <= din[counts];//at first uclk we send din[0] at second uclk tick we send din[1] and so on upto din[7]
         state <= transfer;
       end
       else //when count=8 means transmission is over
       begin
         counts <= 0;  //default value
         tx <= 1'b1;   //default value
         state <= idle; //default state
        donetx <= 1'b1;  //high donetx signifies that whole data is being transferred
       end
     end

     default : state <= idle;
   endcase
  end
end

endmodule

//////////////////////////////////////////////////////////////////

module uartrx
```

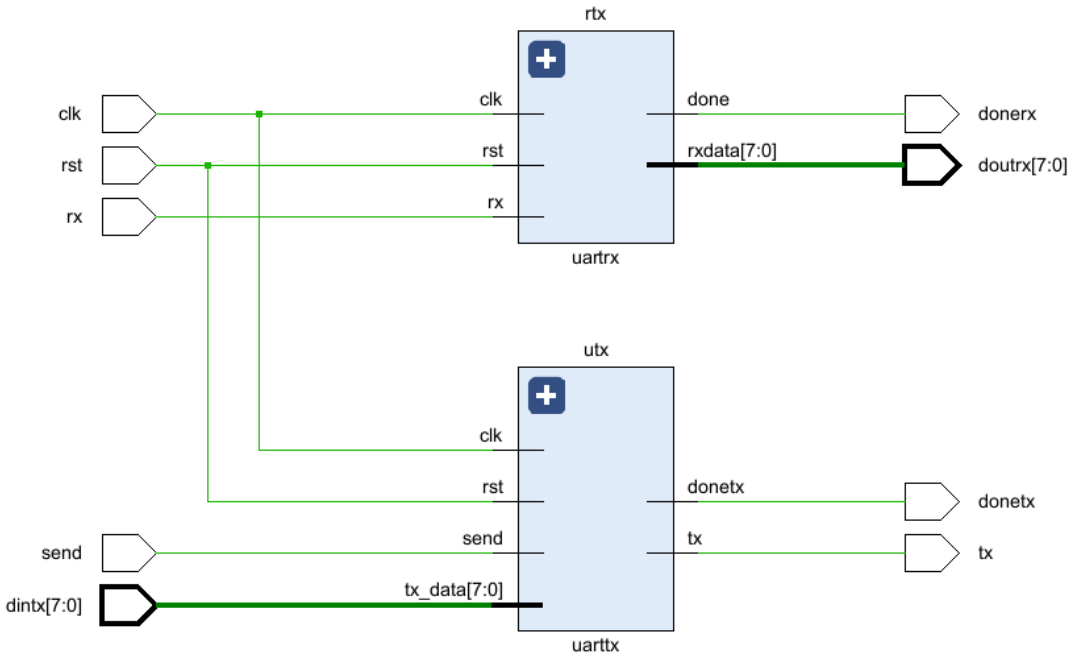
```
#(
parameter clk_freq = 1000000, //MHz  by default clock frequency
parameter baud_rate = 9600      // default output baud rate
   )
 (
input clk, //global signal
input rst, //global signal
input rx, //utilizing rx pin we will be receiving data from a peripheral
output reg done, //when we complete recieving all the data done will become high which indicates data is received
output reg [7:0] rxdata //the data we have received will be send on rxdata for display
);

localparam clkcount = (clk_freq/baud_rate);

integer count = 0;
integer counts = 0;

reg uclk = 0;

enum bit[1:0] {idle = 2'b00, start = 2'b01} state;

 ///////////uart_clock_generation similar to UARTtx
  always@(posedge clk)
    begin
      if(count < clkcount/2)
        count <= count + 1;
      else begin
        count <= 0;
        uclk <= ~uclk;
      end
```

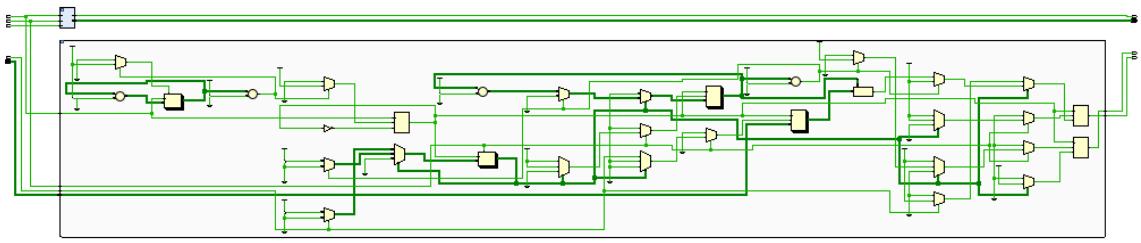
```
  end

//Main FSM of our system uartrx
 always@(posedge uclk)
  begin
   if(rst)
   begin
  rxdata <= 8'h00; //initialized to default value
  counts <= 0;  //initialized to default value
  done <= 1'b0;  //initialized to default value
   end
  else
  begin
  case(state)

  idle : //as rst signal gets low we jump to idle state
  begin
  rxdata <= 8'h00;
  counts <= 0;
  done <= 1'b0;

  if(rx == 1'b0) //As default value of rx is 1 so to start data reception we must make rx=0 so in this line we're checking
  //if rx==0 i.e wheather to start reception

    state <= start;//as soon as rx becomes 0 we jump to start state
  else
    state <= idle;//if rx is not 0 then remain in the idle stae
  end

  start: //reception of data starts with this state
```

```
    begin
     if(counts <= 7)
     begin
    counts <= counts + 1;  //incementing count
    rxdata <= {rx, rxdata[7:1]}; //implementing Right shift register{ rx is being added to MSB and rest bits gets shifted rightwards with current
    //LSB getting discarded with each clock tick of uclk}
    end
    else //as count reaches 8 we execute else block
    begin
    counts <= 0; //default value
    done <= 1'b1; //high done bit indicated that reception of data is completed
    state <= idle; //state restores itself to default
    end
    end
  default : state <= idle;
  endcase
end
end
endmodule
//////////////////////////////////////////////////////////////////////


interface uart_if;
  logic clk;
  logic uclktx;
  logic uclkrx;
  logic rst;
  logic rx;
  logic [7:0] dintx;
  logic send;
  logic tx;
```

```
 logic [7:0] doutrx;
 logic donetx;
 logic donerx;

endinterface
```

→Schematic:





→Simulation source code(testbench):

```
//half duplex
```

```
class transaction;//primary purpose of transaction class is to include variable for all the input and output ports which are present in a design


 typedef enum bit [1:0] {write = 2'b00 , read = 2'b01} oper_type;//enum type allows us to detect type of operation it has two values 0=write and 1=read

  //write means we want to write data to a peripheral and read means we want to read data from a peripheral


  randc oper_type oper;//oper is variable of type oper_type declared above.oper is randomize by putting rand modifier so it may be assigned 0 or 1 after

  // we call randomize() global signals clk and rst won't be added in transaction class instead they'll be added in testbench top and driver respectively


  bit rx;//rx is data we receive from peripheral as rx may be 0 or 1 so it is declared of bit type


  //we may add rand modifier for other input signals namely rx,tx,send but number of iterations will exceed and the in oper_type we have to

  //add 3rd state v.i.z randomized sate so inorder to avoid further complexity

  rand bit [7:0] dintx;


  bit send;

  bit tx;


  bit [7:0] doutrx;

  bit donetx;

  bit donerx;




  function void display (input string tag);// tag represents class name sending data

    $display("[%0s] : oper : %0s send : %0b TX_DATA : %b RX_IN : %0b TX_OUT : %0b RX_OUT : %b DONE_TX : %0b DONE_RX : %0b", tag, oper.name(), send, dintx, rx, tx, doutrx, donetx, donerx);

  endfunction// %0s is for string v.i.z for tag {class name sending data}
```

```
  function transaction copy();// performing deep copy of transaction
    copy = new();
    copy.rx = this.rx;
    copy.dintx = this.dintx;
    copy.send = this.send;
    copy.tx = this.tx;
    copy.doutrx = this.doutrx;
    copy.donetx = this.donetx;
    copy.donerx = this.donerx;
    copy.oper = this.oper;
  endfunction

endclass

class generator; //generator class

 transaction tr; //creating handler for transaction

  mailbox #(transaction) mbx; //mailbox is required for sending data from generator to a driver

  //initializing events
  event done;  //done is used to convey information to our testbench top that we have completed sending requested number of transactions

  int count = 0;//stores the number of iterations user requests

  event drvnext;//used to sense wheather driver have completed it's operation of triggering an interface
  event sconext;//used to get information that wheather scoreboard has completed it's objective

  //custom constructer for our generator class
```

```
  function new(mailbox #(transaction) mbx);
    this.mbx = mbx;
    tr = new();
  endfunction

  //main task
  task run();

    repeat(count) begin
    //as we've added no constraint in transaction hence the line below will give no output values on tcl console as randomization will never fail
      assert(tr.randomize) else $error("[GEN] :Randomization Failed");//tr.randomize generate random values for the variables for which modifiers are added
      mbx.put(tr.copy); //data is being sent to drive through mbx.put(tr.copy) here deep copy of transaction is being sent
      tr.display("GEN");
      @(drvnext); //waiting for an event from driver
      @(sconext); //waiting for an event from a scoreboard as scoreboard completes an operation so does monitor
    end

    -> done;//informs other classes that generator has done it's job of putting requested number of transactions for driver
  endtask



endclass

////////////////////////////////////////////////////////

class driver;
```

virtual uart_if vif;// accessing interface in class using virtual keyword interface name is uart_if and it's variable is vif so,for accessing interface in

//driver class we use keyword vif

transaction tr;//here tr is a data container used to access data send by generator using mailbox

mailbox #(transaction) mbx; //this mailbox is used to receive data from the generator hence parameter transaction is added

mailbox #(bit [7:0]) mbxds;//this mailbox is used to send 8 bit data to scoreboard whose job is to compare data on input line dintx[7:0] with output

//serial line tx when send=1

event drvnext;

bit [7:0] din;

bit wr = 0; ///random operation read / write

bit [7:0] datarx; ///data rcvd during read

//custom constructor for mailbox

function new(mailbox #(bit [7:0]) mbxds, mailbox #(transaction) mbx);

this.mbx = mbx;

this.mbxds = mbxds;

endfunction

```
  //this task is used to reset the peripheral
  task reset();
    vif.rst <= 1'b1;//reset is active high by default
    vif.dintx <= 0; //default values
    vif.send <= 0;  //default values
    vif.rx <= 1'b1;  //default values
    vif.doutrx <= 0;  //default values
    vif.tx <= 1'b1;  //default values
    vif.donerx <= 0; //default values
    vif.donetx <= 0; //default values
    repeat(5) @(posedge vif.uclktx); //waiting for 5 clock ticks of uclktx (a slower clock) and then apply
0 to vif.rst to it
    vif.rst <= 1'b0;
    @(posedge vif.uclktx);
    $display("[DRV] : RESET DONE");
  endtask



  //in this task we decode the type of operations and apply it to a signal
  task run();


    forever begin
      mbx.get(tr); //receiving transaction from a generator
      //decoding type of operation
      if(tr.oper == 2'b00)  ////data transmission ,performing write operation
          begin
          // w=at a time we can either write data to peripheral or read data from peripheral
            @(posedge vif.uclktx);//waiting for posetive edge of uclktx
            vif.rst <= 1'b0;
            vif.send <= 1'b1;  ///start data sending op
            vif.rx <= 1'b1;
```

```
        vif.dintx = tr.dintx;// data send by generator tr.dintx is being applied to dintx
        @(posedge vif.uclktx);//after data is received we make send signal low
        vif.send <= 1'b0;
         ////wait for completion
        //repeat(9) @(posedge vif.uclktx);
        mbxds.put(tr.dintx);//same data on dintx is being send to the scoreboard to be compared with data on output serial port tx
        $display("[DRV]: Data Sent : %0d", tr.dintx);
         wait(vif.donetx == 1'b1);  //we'll hold our simulation till we get donetx=1
         ->drvnext;  //this even trigger signifies that driver has completed it's job of triggering an interface
       end

    else if (tr.oper == 2'b01) //data reception,read operation
          begin

           @(posedge vif.uclkrx);//waiting for posetive edge of uclkrx both clks uclktxand uclkrx are working at same frequency
            vif.rst <= 1'b0;
            vif.rx <= 1'b0; //start condition for read operation
            vif.send <= 1'b0;//as we're receiving data from peripheral hence send=0
            @(posedge vif.uclkrx);

           for(int i=0; i<=7; i++)
           begin
              @(posedge vif.uclkrx);
              vif.rx <= $urandom;//entering values in rx using urandom() which generates a 32 bit random unsigned integer
              datarx[i] = vif.rx; // the same data we're sending on rx is getting stores in datarx which helps us to compare data that we recieve
                                  // serially on rx and on doutrx[7:0]
           end
```

```
            mbxds.put(datarx);// sending data to scoreboard

            $display("[DRV]: Data RCVD : %0d", datarx); //displaying msg on console and display the data we've recieved
            wait(vif.donerx == 1'b1);//haulting simulation till donerx becomes 1
             vif.rx <= 1'b1; //default value of rx when no reception takes place
                                    ->drvnext;


          end



   end

 endtask

endclass

class monitor; //primary objective of monitor is to recieve a response update the data member of a transaction and then send it to scoreboard for comparison
          // with golden data and also handles which data to be send to scoreboard

 transaction tr;

 mailbox #(bit [7:0]) mbx;// here mailbox is used to communicate data to the scoreboard

 bit [7:0] srx; //////the data we have on tx pin will be stored here (send)
 bit [7:0] rrx; ///// the we read during read operation dintx(recieve)
 //here we'll be sampling data on doutrx bus v.i.z 8 bit instead of sampling bit by bit
```

```
 virtual uart_if vif;//accessing interface


 function new(mailbox #(bit [7:0]) mbx); //constructor to make mailbox to work between monitor and scoreboard
   this.mbx = mbx;
   endfunction


 task run();


   forever begin


     @(posedge vif.uclktx);//waiting for positive edge of a clock
    if ( (vif.send== 1'b1) && (vif.rx == 1'b1) ) //performing write operation  rx=1 means reading data is disabled on transmission is allowed
            begin


              @(posedge vif.uclktx); ////start collecting tx data from next clock tick


           for(int i = 0; i<= 7; i++) //collecting data from tx and storing it in srx
           begin
               @(posedge vif.uclktx);//wait for posetive edge of clock
               srx[i] = vif.tx;


           end



             $display("[MON] : DATA SEND on UART TX %0d", srx);


             //////////wait for done tx before proceeding next transaction
            @(posedge vif.uclktx); //
            mbx.put(srx);
```

```
            end

    else if ((vif.rx == 1'b0) && (vif.send == 1'b0) ) //rx=0 signifies data reception and during reception send=0
      begin
        wait(vif.donerx == 1);//waiting for completion of data reception
        rrx = vif.doutrx;     //data available on 8 bit output bus is stored in rrx
        $display("[MON] : DATA RCVD RX %0d", rrx);
        @(posedge vif.uclktx);
        mbx.put(rrx);//data is put in mbx so that it can be recieved by scoreboard
     end
  end
endtask



endclass

//////////////////////////////////////////////////////////////////////////////

class scoreboard;
  mailbox #(bit [7:0]) mbxds, mbxms;//scoreboard needs two mailbox one to recieve data from monitor and other to recieve data from driver

  bit [7:0] ds; //data container
  bit [7:0] ms; //data container

   event sconext;//we need and event because before generator starting to send next transaction it should recieve trigger from sconext

  function new(mailbox #(bit [7:0]) mbxds, mailbox #(bit [7:0]) mbxms);//constructor having 2 parameters from driver and scoreboard
    this.mbxds = mbxds;
```

```
    this.mbxms = mbxms;
  endfunction

  task run();
    forever begin

      mbxds.get(ds);//storing data from driver in ds
      mbxms.get(ms);//storing data from monitor in ms

      $display("[SCO] : DRV : %0d MON : %0d", ds, ms);
      if(ds == ms)
        $display("DATA MATCHED");
      else
        $display("DATA MISMATCHED");

     ->sconext; //triggering an event so that generator could start it's next transaction
    end
  endtask



endclass

////////////////////////////////

class environment;//Environment act as data container connecting all mailbox and event together
and also correctly calling different task

    generator gen;//creating instance of all the classes
    driver drv;
    monitor mon;
    scoreboard sco;
```

```
  event nextgd; ///gen -> drv

  event nextgs;  /// gen -> sco

mailbox #(transaction) mbxgd; ///gen - drv

mailbox #(bit [7:0]) mbxds; /// drv - sco


mailbox #(bit [7:0]) mbxms;  /// mon - sco

  virtual uart_if vif;


function new(virtual uart_if vif);//custom constructor having argument as an interface

  mbxgd = new();
  mbxms = new();
  mbxds = new();

  gen = new(mbxgd);
  drv = new(mbxds,mbxgd);



  mon = new(mbxms);
  sco = new(mbxds, mbxms);
```

```
  this.vif = vif;
  drv.vif = this.vif;//same interface in driver
  mon.vif = this.vif;//same interface in monitor

  gen.sconext = nextgs;//event merging
  sco.sconext = nextgs;

  gen.drvnext = nextgd;
  drv.drvnext = nextgd;

endfunction

task pre_test();
  drv.reset();
endtask

task test();
fork
  gen.run();
  drv.run();
  mon.run();
  sco.run();
join_any
endtask

task post_test();
  wait(gen.done.triggered);
  $finish();
endtask

task run();
```

```
    pre_test();
    test();
    post_test();
  endtask



endclass

//////////////////////////////////////////

module tb;

  uart_if vif();//declaring an interface

  uart_top #(1000000, 9600) dut (vif.clk,vif.rst,vif.rx,vif.dintx,vif.send,vif.tx,vif.doutrx,vif.donetx,
vif.donerx);
  //connection of interface to ports of design specifing frequency to 1MHZ and Baud rate to be 9600



    initial begin
      vif.clk <= 0;//initializing clock signal to zero
    end

    always #10 vif.clk <= ~vif.clk;

    environment env; //instance name of environment
```

```
 initial begin
   env = new(vif); //a s environment requires interface as argument hen putting vif as an argument
   env.gen.count = 5;//transaction count to be 5
   env.run();
 end

/*
 initial begin
   $dumpfile("dump.vcd");
   $dumpvars;
 end
*/
assign vif.uclktx = dut.utx.uclk;
assign vif.uclkrx = dut.rtx.uclk;

endmodule
```

//

→Console output:

Log Share

```
# KERNEL: SLP simulation initialization done - time: 0.0 [s].
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 5599 kB (elbread=459 elab2=4975 kernel=164 sdf=0)
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: [DRV] : RESET DONE
# KERNEL: [GEN] : oper : write send : 0 TX_DATA : 01011000 RX_IN : 0 TX_OUT : 0 RX_OUT : 00000000 DONE_TX : 0 DONE_RX : 0
# KERNEL: [DRV]: Data Sent : 88
# KERNEL: [MON] : DATA SEND on UART TX 88
# KERNEL: [SCO] : DRV : 88 MON : 88
# KERNEL: DATA MATCHED
# KERNEL: [GEN] : oper : read send : 0 TX_DATA : 00001010 RX_IN : 0 TX_OUT : 0 RX_OUT : 00000000 DONE_TX : 0 DONE_RX : 0
# KERNEL: [DRV]: Data RCVD : 98
# KERNEL: [MON] : DATA RCVD RX 98
# KERNEL: [SCO] : DRV : 98 MON : 98
# KERNEL: DATA MATCHED
# KERNEL: [GEN] : oper : write send : 0 TX_DATA : 11001110 RX_IN : 0 TX_OUT : 0 RX_OUT : 00000000 DONE_TX : 0 DONE_RX : 0
# KERNEL: [DRV]: Data Sent : 206
# KERNEL: [MON] : DATA SEND on UART TX 206
# KERNEL: [SCO] : DRV : 206 MON : 206
# KERNEL: DATA MATCHED
# KERNEL: [GEN] : oper : read send : 0 TX_DATA : 00100111 RX_IN : 0 TX_OUT : 0 RX_OUT : 00000000 DONE_TX : 0 DONE_RX : 0
# KERNEL: [DRV]: Data RCVD : 224
# KERNEL: [MON] : DATA RCVD RX 224
# KERNEL: [SCO] : DRV : 224 MON : 224
# KERNEL: DATA MATCHED
# KERNEL: [GEN] : oper : write send : 0 TX_DATA : 11111110 RX_IN : 0 TX_OUT : 0 RX_OUT : 00000000 DONE_TX : 0 DONE_RX : 0
# KERNEL: [DRV]: Data Sent : 254
# KERNEL: [MON] : DATA SEND on UART TX 254
# KERNEL: [SCO] : DRV : 254 MON : 254
# KERNEL: DATA MATCHED
# RUNTIME: Info: RUNTIME_0068 testbench.sv (350): $finish called.
# KERNEL: Time: 138850 ns,  Iteration: 1,  Instance: /tb,  Process: @INITIAL#384_2@.
# KERNEL: stopped at time: 138850 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
Finding VCD file...
./dump.vcd
[2023-07-18 00:02:48 UTC] Opening EPWave...
Done
```