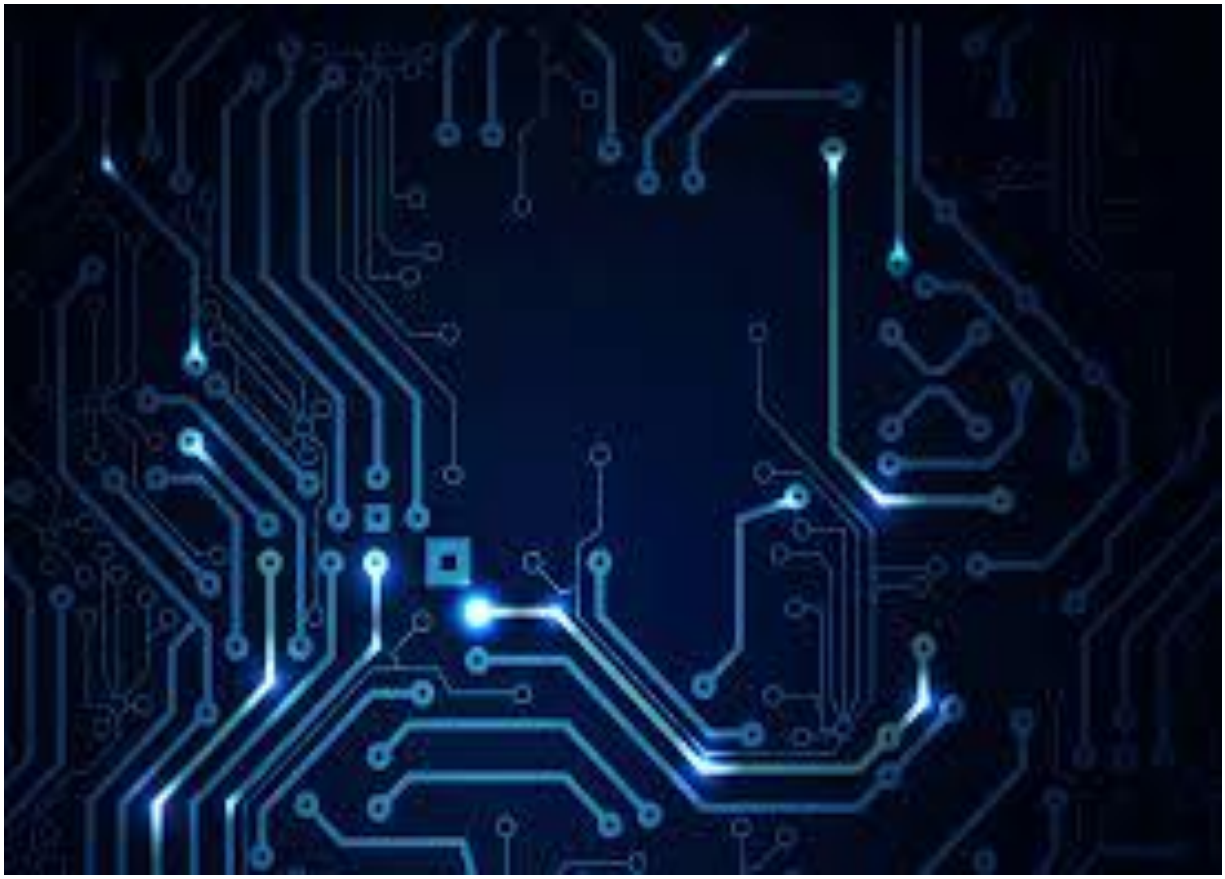
UART Protocol

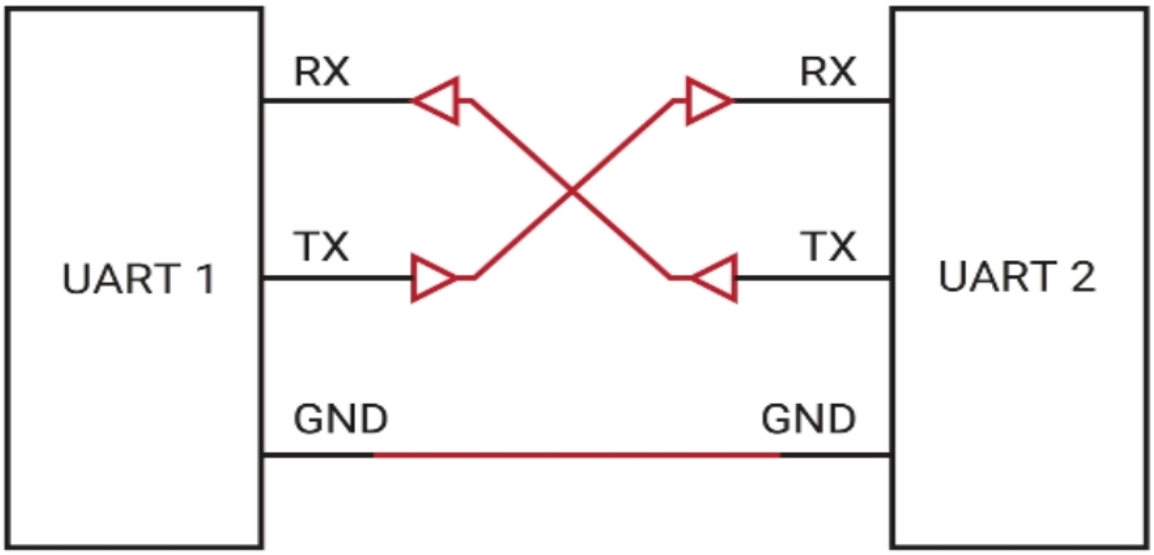


Prepared by

Ali Abd El Aleem

Overview:

Universal Asynchronous Receiver/Transmitter (UART) is a widely used serial communication protocol that enables asynchronous, full-duplex data transfer between two devices using only two main lines: TX (transmit) and RX (receive). Unlike synchronous protocols, UART does not use a clock signal; instead, both devices must operate at the same baud rate (e.g., 9600 or 115200 bps). Data is transmitted in structured frames consisting of a start bit, 5–9 data bits (commonly 8), an optional parity bit for error checking, and one or more stop bits. The line remains high during the idle state and transitions low to indicate the start of a frame. UART is simple, low-cost, and reliable for short-distance, point-to-point communication, making it popular in microcontrollers, PCs, GPS, Bluetooth modules, and other embedded systems. However, it is limited by distance, speed.



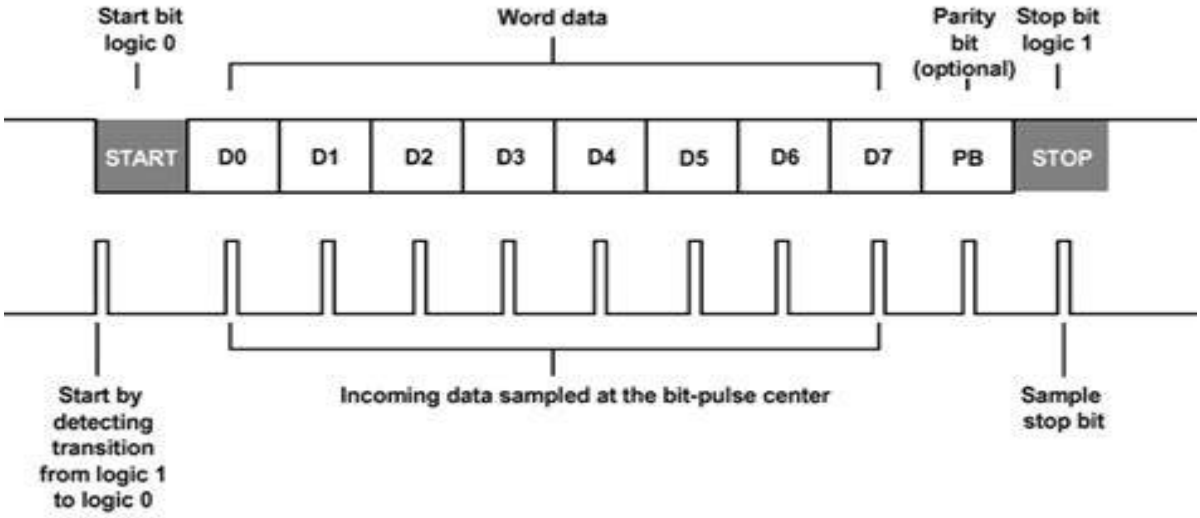
Why UART:

UART is widely used because it provides a simple, reliable, and cost-effective way to transfer data between two devices. It requires only two data lines (TX and RX), making hardware design straightforward. Since it is asynchronous, no extra clock line is needed devices just need to share the same baud rate. UART is built into most microcontrollers, computers, and peripheral modules, so no special hardware is required. It is ideal for short-distance, point-to-point communication in applications like debugging, programming, sensor interfacing, GPS, and Bluetooth modules.

Comparison of UART, SPI, and I²C:

feature	UART	SPI	I²C
Communication type	Asynchronous, serial, point-to-point	Synchronous, serial, master–slave	Synchronous, serial, multi-master, multi-slave
Wires required	2 (TX, RX)	4 (MOSI, MISO, SCLK, SS)	2 (SDA, SCL)
Speed	Moderate (up to a few Mbps)	Very high (tens of Mbps)	Moderate (typically up to a few Mbps)
No. of devices	2 devices only (1:1)	1 master, multiple slaves(needs SS line per slave)	Multiple masters and slaves on same 2 lines
Clock line	No clock (asynchronous)	Yes (SCLK)	Yes (SCL)

How does it work?



First of all, the Tx is on Idle state where no communication occurs, then the sense a start bit of logic zero means start communication so it goes to Start state, then start to serially feed the data out of it (parallel to serial conversion) for (5-8) cycles depending how many data bits within' the frame and this state called Data state, and optionally send a parity bit (not implemented here).

Eventually, goes to Stop for the stop bit (can be a bit and a half or even 2 bits).

On the other hand, the Rx is on Idle state waiting there, then the sense of start bit of logic zero means start communication so it goes to Start state, then start to do the exact opposite of the TX as it collects the serially transmitted data and grouping them together through Data state.

Baud generator:

The system clock of the Tx and the Rx may not be the same as there is no common clock between them, so we need to generate a common clock or tick to synchronize the transmitting and receiving operations.

The baud generator is the block responsible for the generation if the baud clock to synchronize the Tx and the Rx with the same speed.

Be generating a divisor to divide the system clock frequency and make it slower based on specific baud rate for both the Tx and the Rx.

The divisor can be calculated as:

Divisor = system clock / (oversampling ration * baud rate)

And the oversampling ratio is the number of times we are going to read the transmitted bit until we sample it at the middle of the sample to make sure that the signal is stable at this time and ovoid any misalignment between the transmitted and the received bit.

The oversampling rotation can be any number but in common they are *13 or *16 in this design it is *16.

Note that:

In this design we treat the baud rate as a modules counter, with input called final value, we determined it.

Code for baud rate:

ner_input.v

```
module timer_input
    #(parameter BITS = 4)(
    input clk,
    input reset_n,
    input enable,
    input [BITS - 1:0] FINAL_VALUE,
//    output [BITS - 1:0] Q,
    output done
    );

    reg [BITS - 1:0] Q_reg, Q_next;

    always @(posedge clk, negedge reset_n)
    begin
        if (~reset_n)
            Q_reg <= 'b0;
        else if(enable)
            Q_reg <= Q_next;
        else
            Q_reg <= Q_reg;
    end

    // Next state logic
    assign done = Q_reg == FINAL_VALUE;

    always @(*)
        Q_next = done? 'b0: Q_reg + 1;


endmodule
```

Ln 10

Tx interface:

UART is parallel to output transmitter means that the input data is parallel bus which will be transmitted bit by bit after each other.

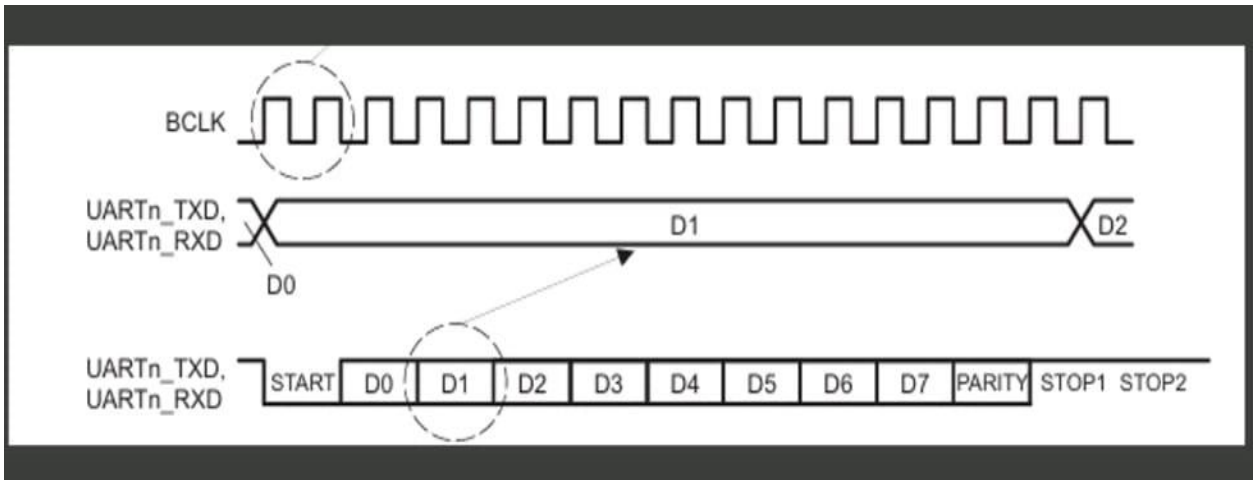
The output clock “ticks“ generated from the baud rate generator is the oversampling clock means that data will be transmitted for 16 BCLK’s positive edge before start transmitting the following bit of data.

Both the start and the stop bits will also last 16 positive edge .

Before sending any frame, the Tx will be in Idle state where the Tx_out signal will be always 1 and will continuously sample the empty flag of the FIFO and rise the read_enable of the FIFO, so, when the Tx is in Idle state and the FIFO is not empty the Tx will read the signal from the FIFO and start transmitting the signal at the following clock cycle by lowering the Tx_out for 16 posedge “the start bit” then 8 bits for the signal then the stop bit/s then go to the idle state and repeat.

The frame usually consists of:

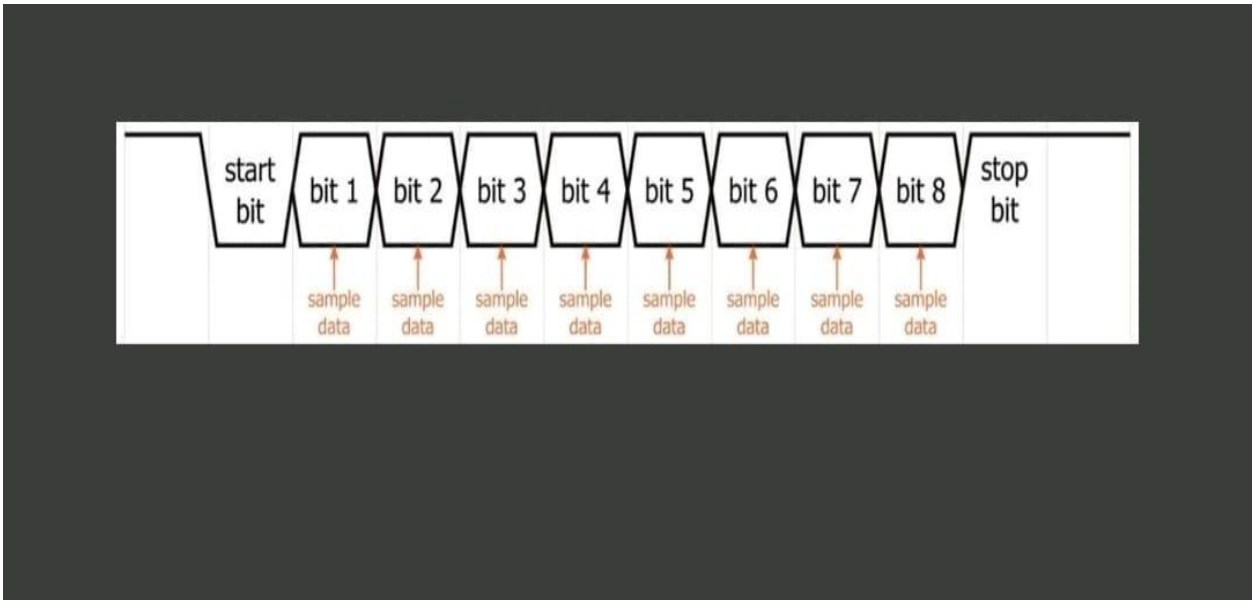
1. Start bit “low bit”.
2. 5 or 6 or 7 data bits.
3. Parity bit ”optional”.
4. 1 or 1.5 or 2 stop bit/s “high bit/s”



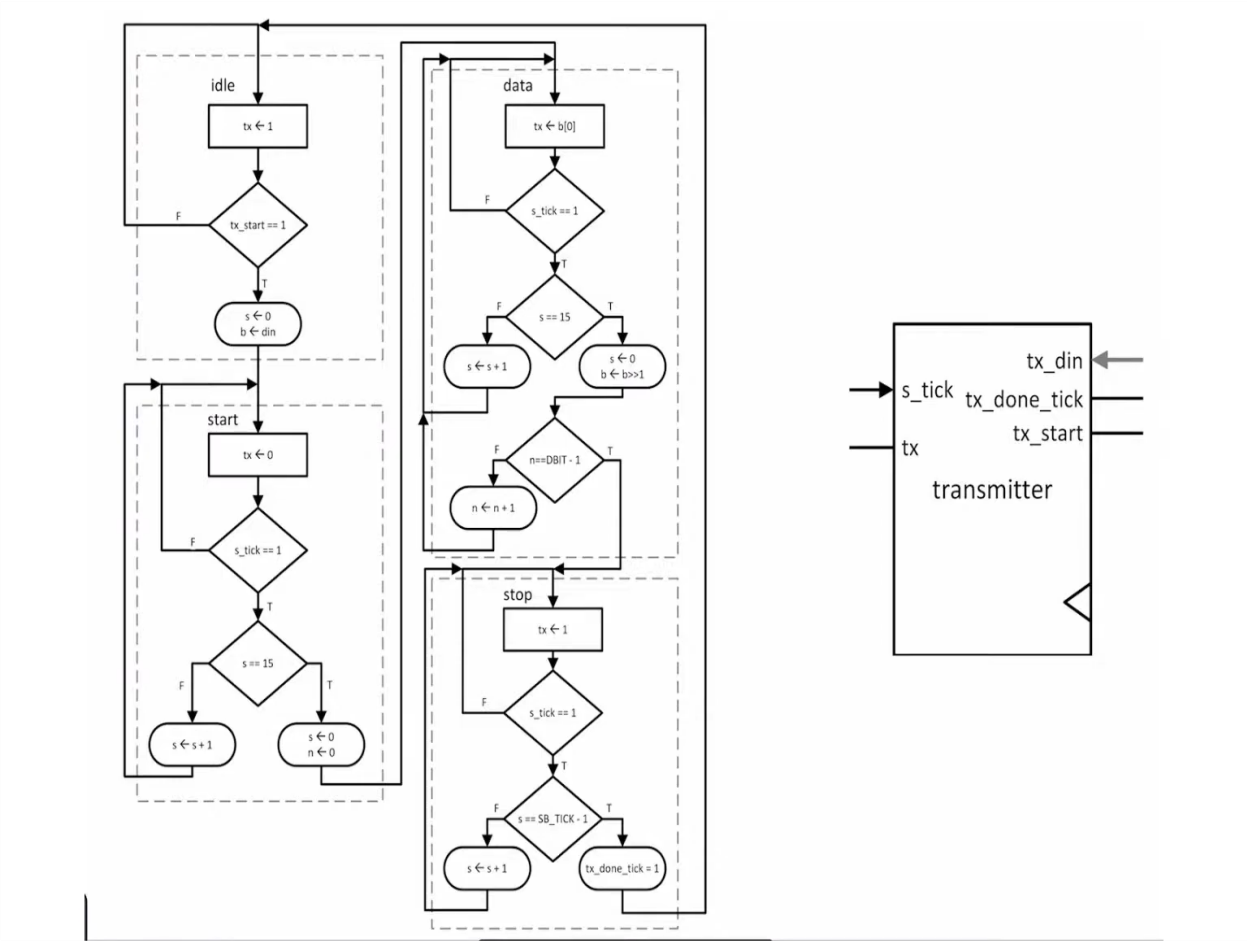
Rx interface:

The Rx receive the transmitted signal from the Tx and oversample it to sample the data at the middle of the bit period “usually after 7 bits”.

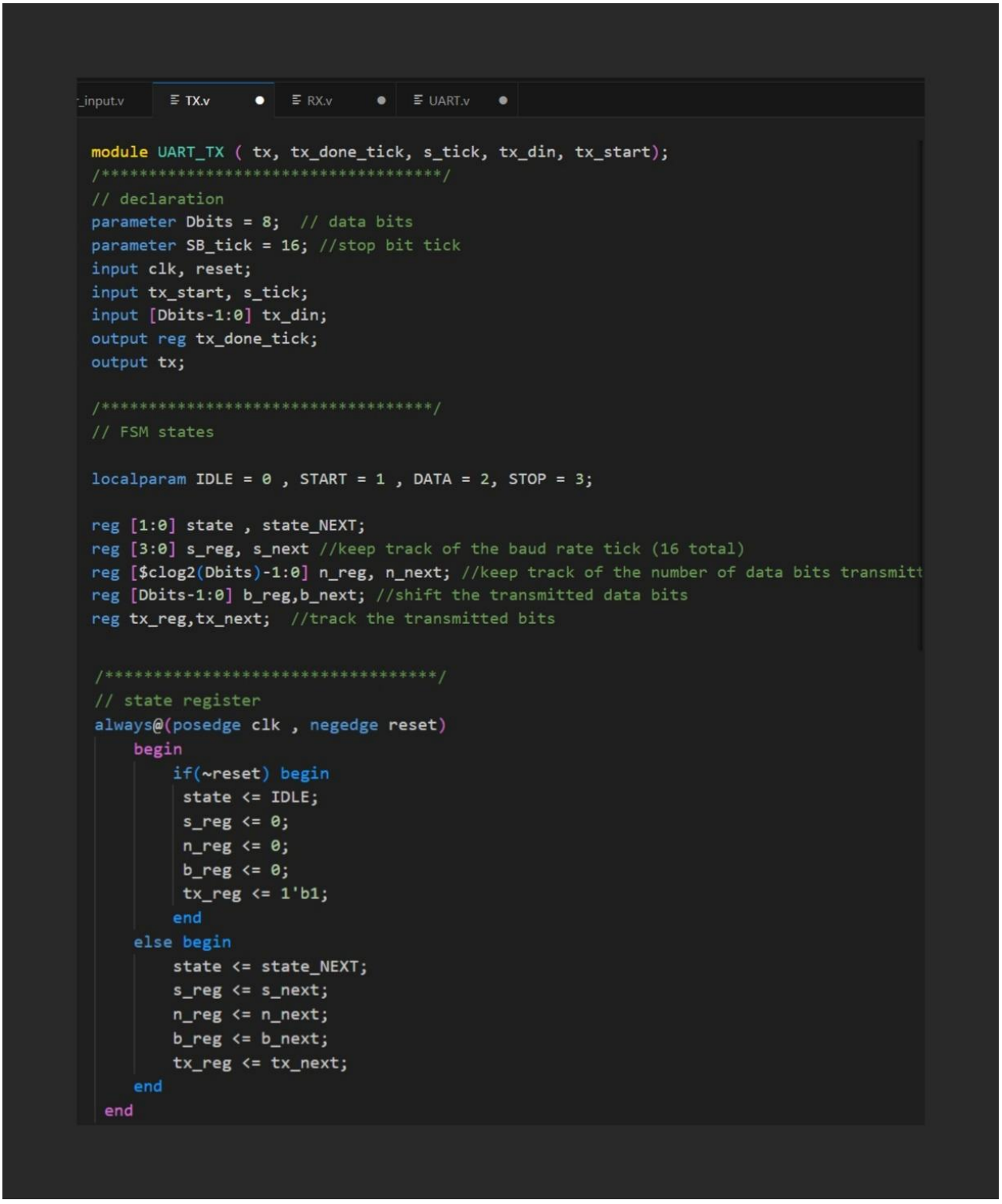
When the Tx_out go down “the start bit” the Rx will start sampling for 7 positive edges to be at the middle of the start bit then start sampling for 16 bits to sample each data bit at the middle and after doing that for 8 times for data bit it will do that again one more time for the stop bit and where it rises the write_enable of the FIFO and writes the received data into the FIFO then became idle again waiting the Tx_out to be low.



FSM for TX interface:



Code for TX interface:



```
module UART_TX ( tx, tx_done_tick, s_tick, tx_din, tx_start);
/***********************************/
// declaration
parameter Dbits = 8;  // data bits
parameter SB_tick = 16; //stop bit tick
input clk, reset;
input tx_start, s_tick;
input [Dbits-1:0] tx_din;
output reg tx_done_tick;
output tx;

/***********************************/
// FSM states

localparam IDLE = 0 , START = 1 , DATA = 2, STOP = 3;

reg [1:0] state , state_NEXT;
reg [3:0] s_reg, s_next //keep track of the baud rate tick (16 total)
reg [$clog2(Dbits)-1:0] n_reg, n_next; //keep track of the number of data bits transmitt
reg [Dbits-1:0] b_reg,b_next; //shift the transmitted data bits
reg tx_reg,tx_next;  //track the transmitted bits

/***********************************/
// state register
always@(posedge clk , negedge reset)
    begin
        if(~reset) begin
         state <= IDLE;
         s_reg <= 0;
         n_reg <= 0;
         b_reg <= 0;
         tx_reg <= 1'b1;
        end
    else begin
        state <= state_NEXT;
        s_reg <= s_next;
        n_reg <= n_next;
        b_reg <= b_next;
        tx_reg <= tx_next;
    end
 end
```

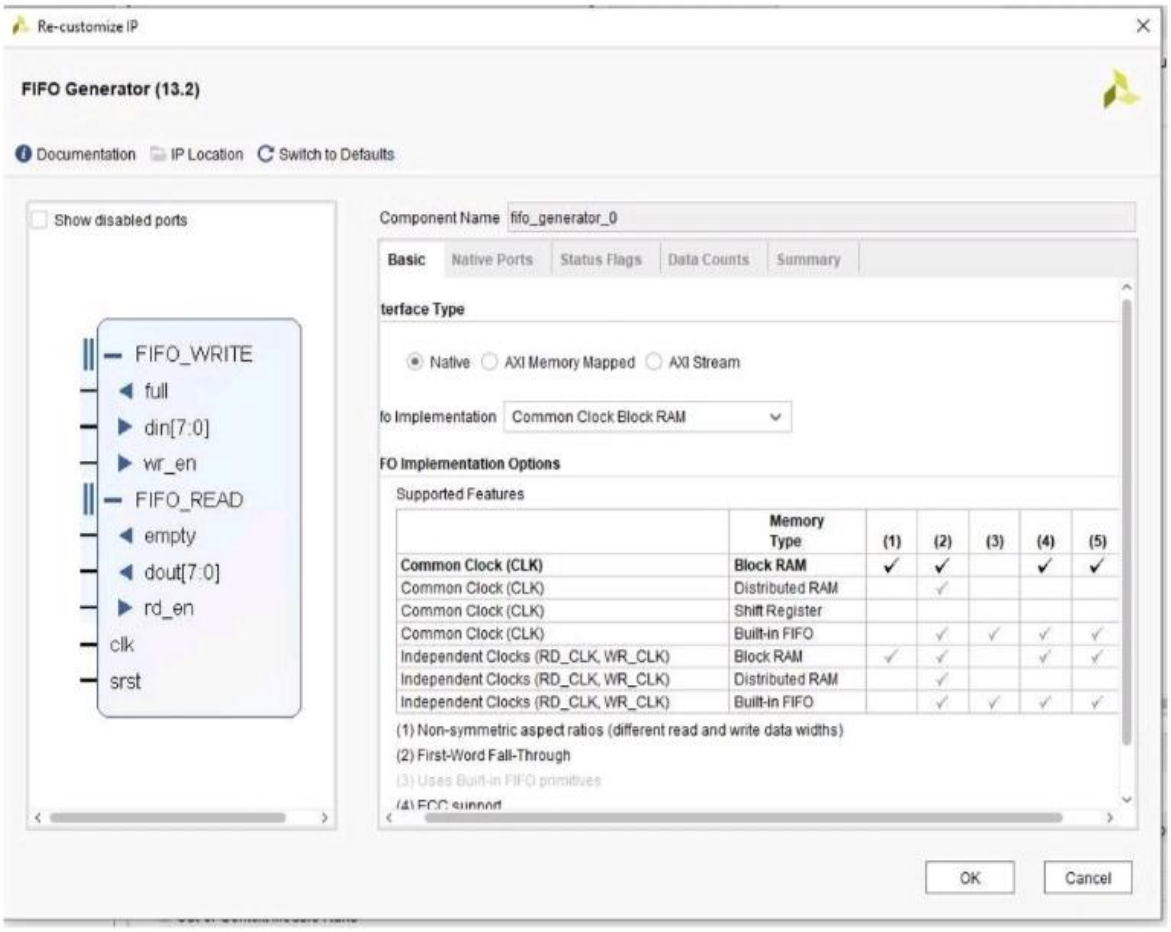
```
/*********************************/
// FSM and next state logic
always@(*) begin
    state_NEXT <= state;
    s_reg <= s_next;
    n_reg <= n_next;
    b_reg <= b_next;
    tx_done_tick <= 1'b0;
    case(state)
        IDLE:begin
            tx_next <= 1'b01
            if(tx_start) begin
                s_next <= 0;
                b_next <= tx_din;
                state_NEXT <= START;
            end
        end
        START: begin
            tx_next <= 0;
            if(s_tick)
                if(s_reg == 15) begin
                    s_next <= 0;
                    n_next <= 0;
                    state_NEXT <= DATA;
                end
                else
                    s_next <= s_reg +1;

        end
        DATA:begin
            tx_next <= b_reg[0];
            if(s_tick)
            if(s_reg == 15) begin
                s_next <= 0;
                b_next <= {1'b0,b_reg[Dbits-1:1]}; //shift right
            if(n_reg == (Dbits-1))
            state_NEXT <= STOP;
            else
             s_next <= s_reg+1;
            end
        end
        STOP:begin
            if(s_tick)
            if(s_reg == (SB_tick-1)) begin
                tx_done_tick <= 1;
                state_NEXT <= IDLE;
            end
            else
                s_next <= s_reg +1;
        end
    endcase
end
endmodule
```

Code for Rx interface:

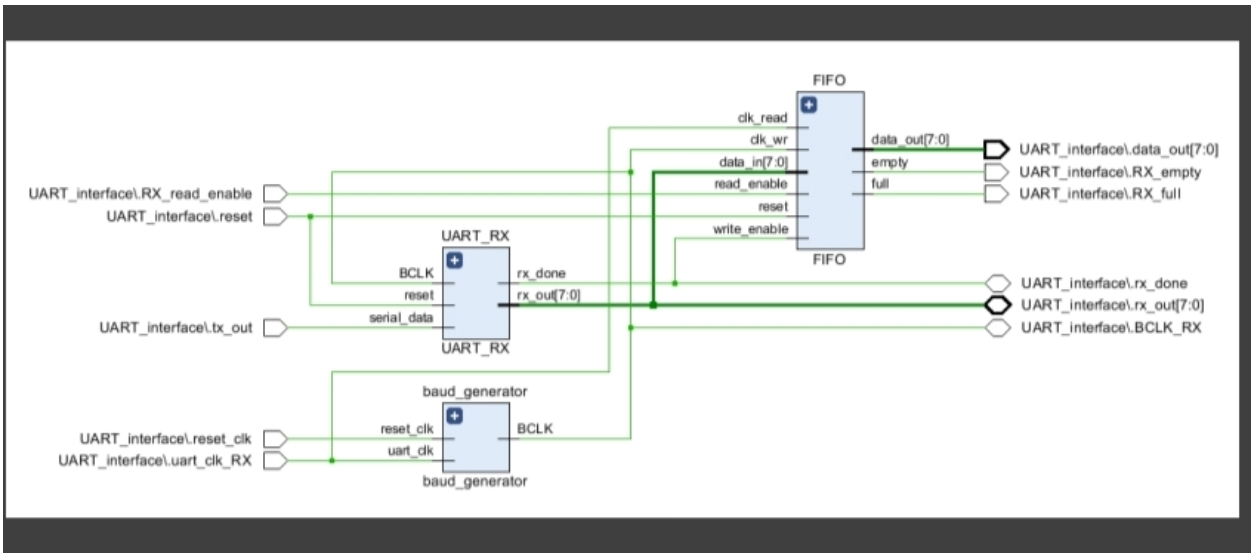
```
module UART_RX( rx_done_tick, rx_dout, rx, s_tick, clk, reset);

/*****************************/
// declaration
parameter Dbits = 8;
parameter SB_tick = 16;
input clk, reset;
input rx, s_tick;
output reg rx_done_tick;
output [Dbits-1:0] rx_dout;

/*****************************/
// FSM parameter
localparam IDLE = 0, START = 1, DATA = 2, STOP = 3;
reg [1:0] state, state_NEXT;
reg [3:0] s_reg, s_next;    //keep track of the baud rate tick (16 total)
reg[$clog2(Dbits-1):0] n_reg, n_next;   //keep track of the number of data bits received
reg [Dbits-1:0] b_reg, b_next;   //shift the received data bits

/*****************************/
// state register
always@(posedge clk , negedge reset)
    begin
        if(~reset) begin
         state <= IDLE;
         s_reg <= 0;
         n_reg <= 0;
         b_reg <= 0;
        end
    else begin
        state <= state_NEXT;
        s_reg <= s_next;
        n_reg <= n_next;
        b_reg <= b_next;
    end
 end
```

```
/*****************************/
// next state logic
always@(*) begin
   state_NEXT <= state;
   s_reg <= s_next;
   n_reg <= n_next;
   b_reg <= b_next;
   rx_done_tick <= 1'b0;
   case(state)
       IDLE:begin
           if(~rx) begin
               s_next <= 0;
               state_NEXT <= START;
           end
       end
       START: begin
           if(s_tick)
               if(s_reg == 7) begin
                   s_next <= 0;
                   n_next <= 0;
                   state_NEXT <= DATA;
               end
               else
                   s_next <= s_reg +1;

       end
          DATA:begin
              if(s_tick)
              if(s_reg == 15) begin
                  s_next <= 0;
                  b_next <= {rx,b_reg[Dbits-1:1]}; //shift right
              if(n_reg == (Dbits-1))
              state_NEXT <= STOP;
              else
               n_next <= n_reg+1;
              end
          end
          STOP:begin
              if(s_tick)
              if(s_reg == (SB_tick-1)) begin
                  rx_done_tick <= 1;
                  state_NEXT <= IDLE;
              end
              else
                  s_next <= s_reg +1;
          end
      endcase
end
endmodule
```

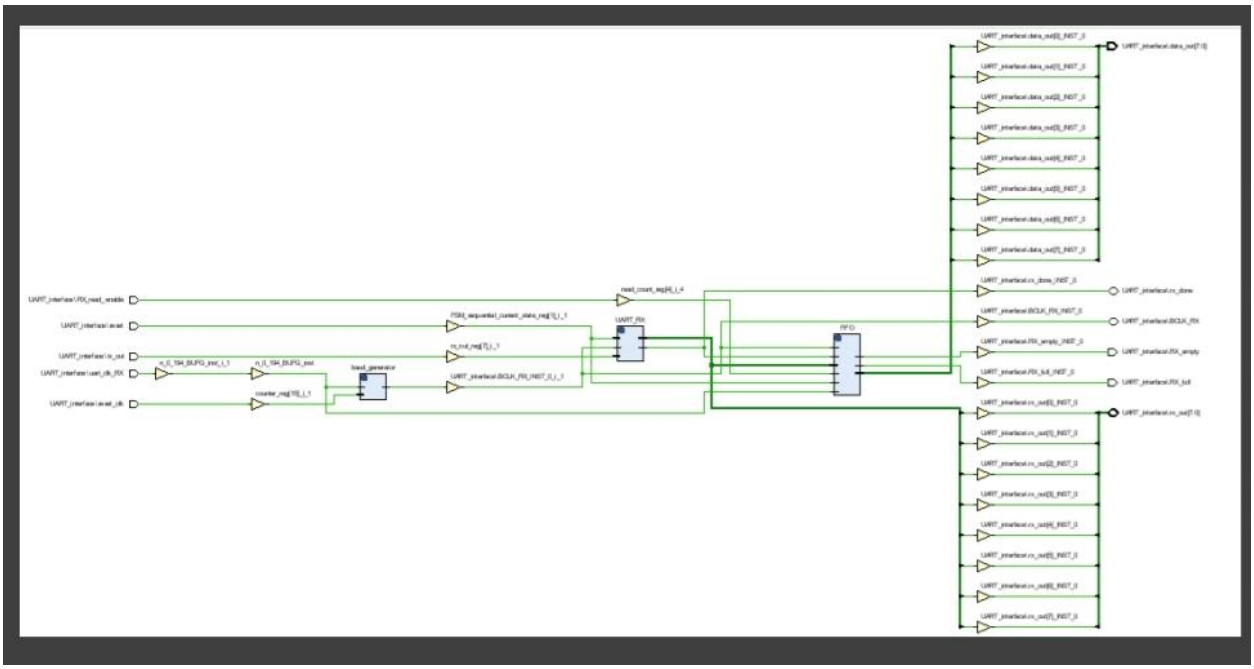
FIFO generator:



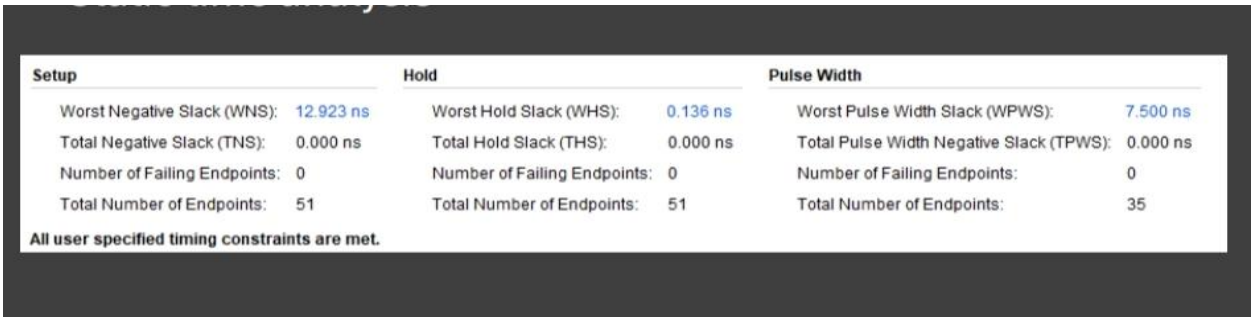
RTL schematic for RX:



Synthesis for RX:



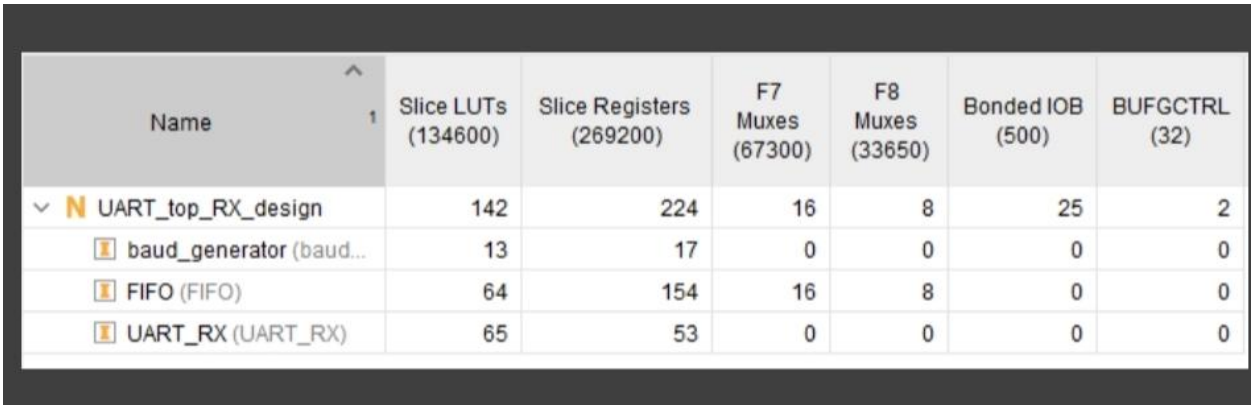
Timing Report for RX:



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	12.923 ns	Worst Hold Slack (WHS):	0.136 ns	Worst Pulse Width Slack (WPWS):	7.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	51	Total Number of Endpoints:	51	Total Number of Endpoints:	35

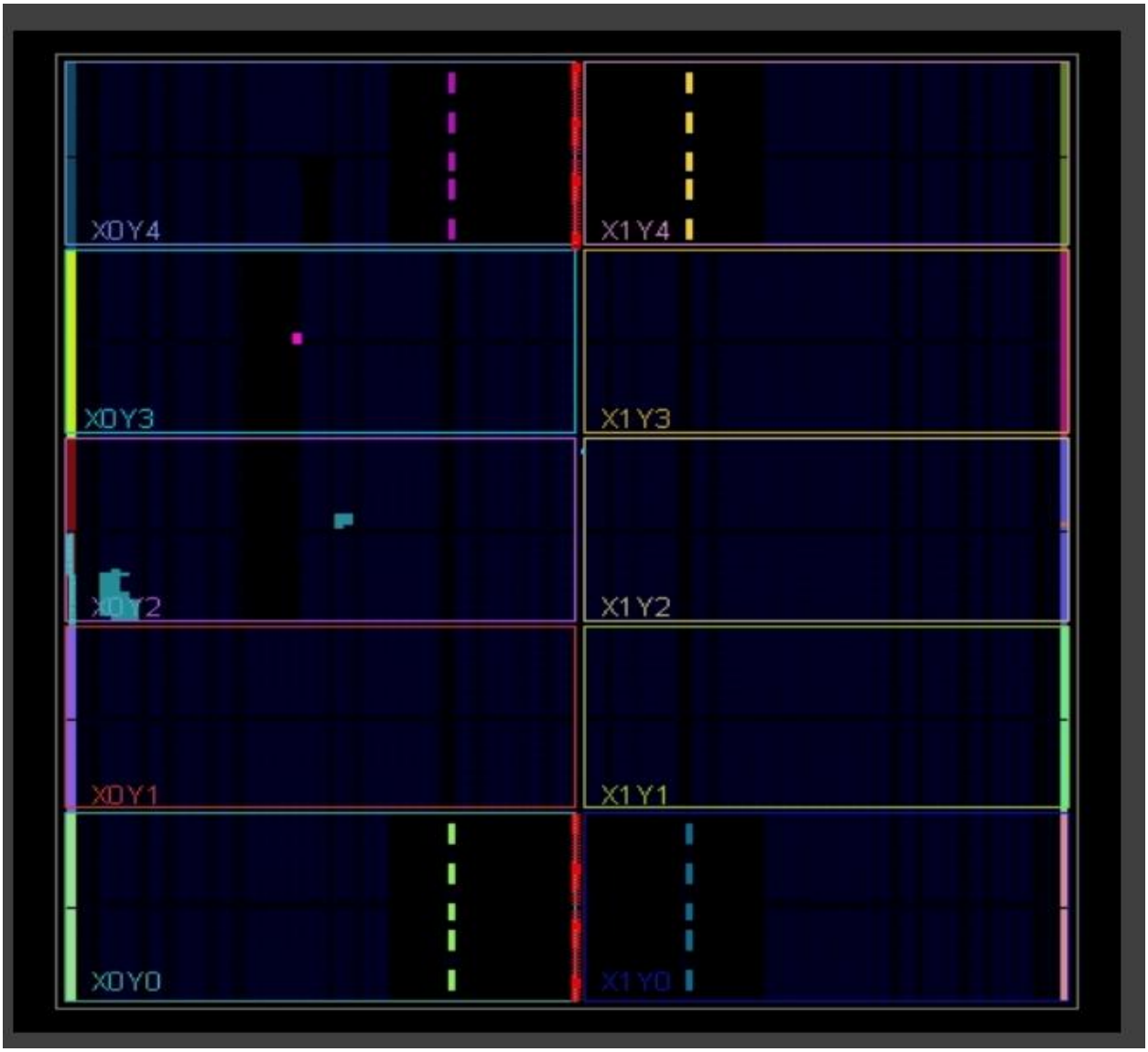
All user specified timing constraints are met.

Utilization Report:

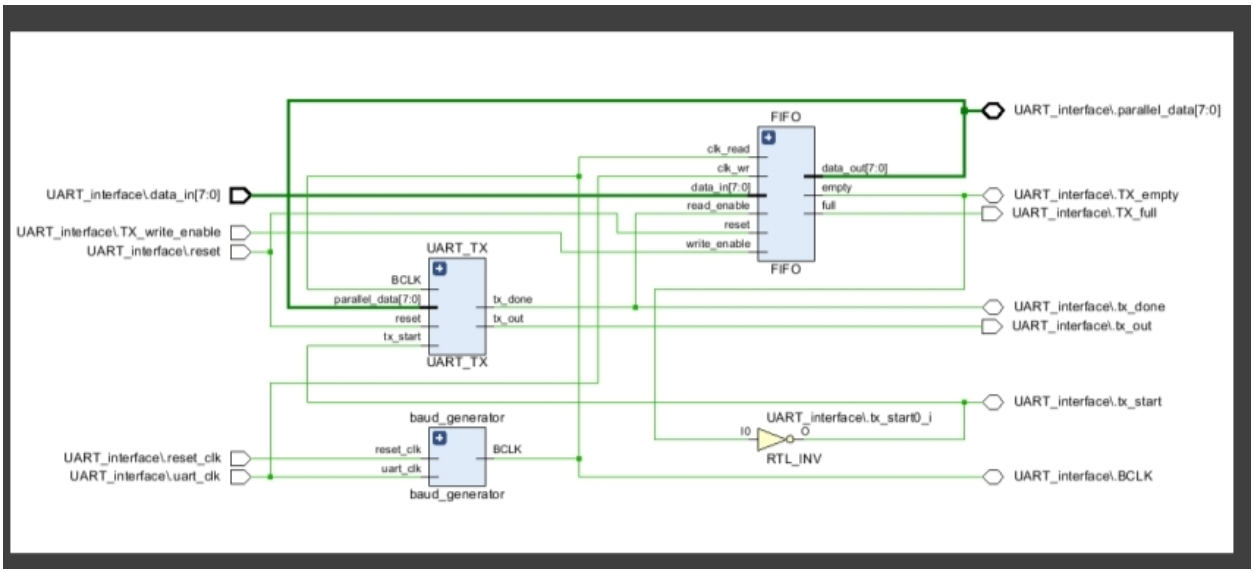


Name	Slice LUTs (134600)	Slice Registers (269200)	F7 Muxes (67300)	F8 Muxes (33650)	Bonded IOB (500)	BUFGCTRL (32)
UART_top_RX_design	142	224	16	8	25	2
baud_generator (baud...	13	17	0	0	0	0
FIFO (FIFO)	64	154	16	8	0	0
UART_RX (UART_RX)	65	53	0	0	0	0

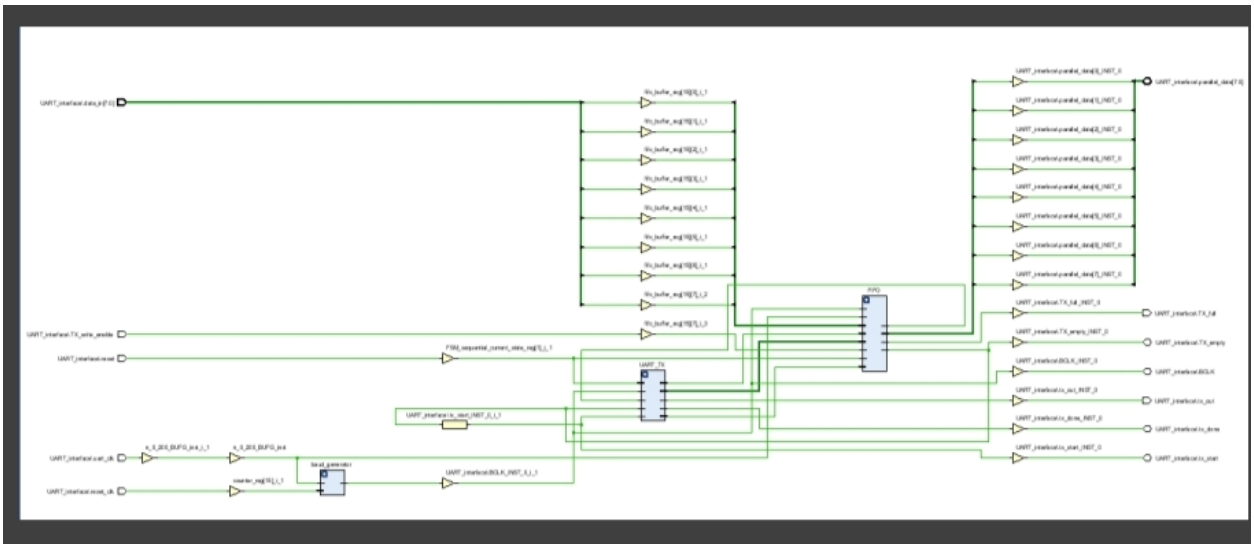
Implementation:



RTL schematic for TX:



Synthesis for TX:



Utilization report for TX:

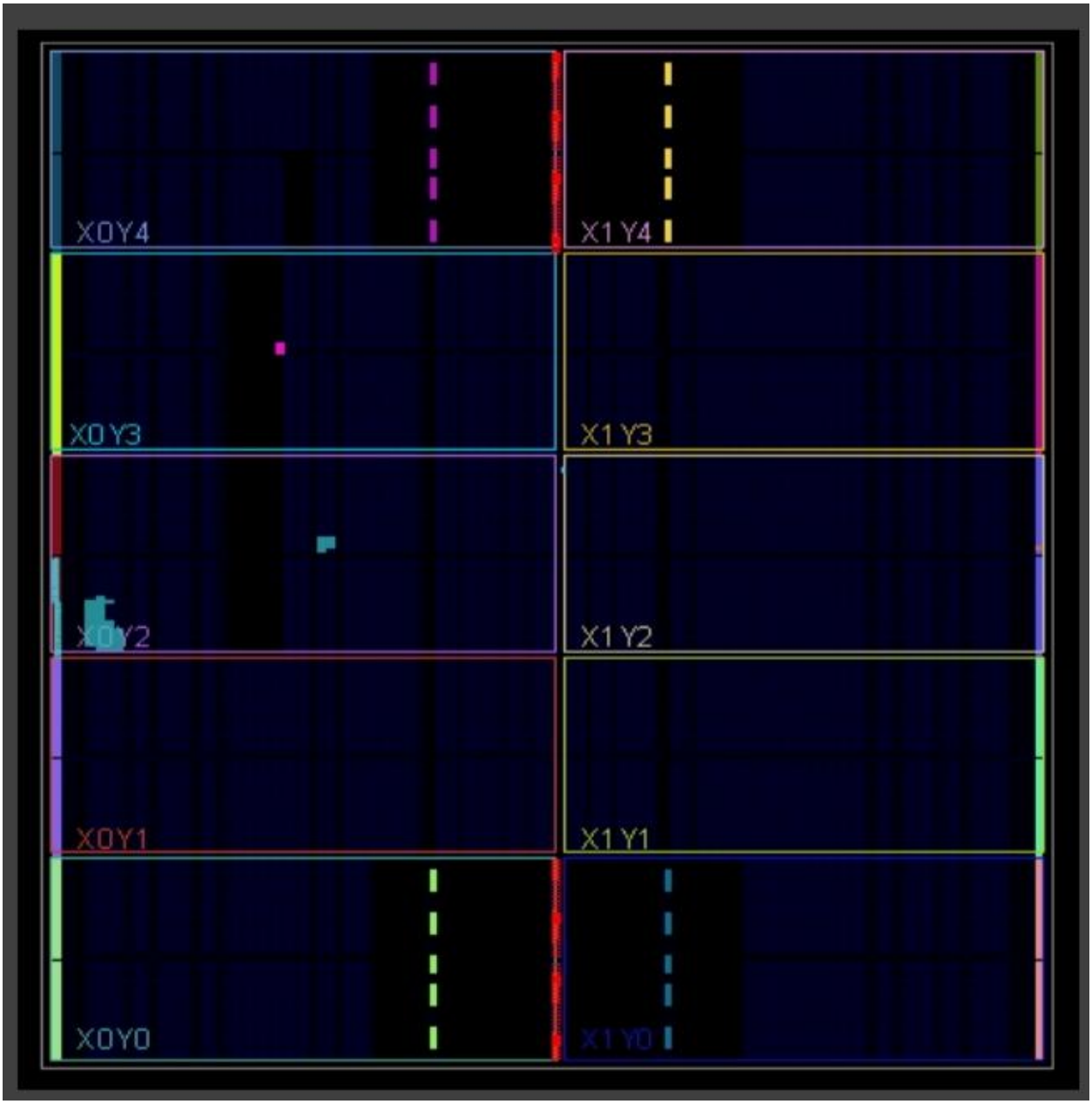
Name	Slice LUTs (134600)	Slice Registers (269200)	F7 Muxes (67300)	F8 Muxes (33650)	Bonded IOB (500)	BUFGCTRL (32)
UART_top_TX_design	152	215	16	8	26	2
baud_generator (baud...	22	17	0	0	0	0
FIFO (FIFO)	73	154	16	8	0	0
UART_TX (UART_TX)	56	44	0	0	0	0

Timing report for TX:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	12.810 ns	Worst Hold Slack (WHS):	0.136 ns	Worst Pulse Width Slack (WPWS):	7.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	163	Total Number of Endpoints:	163	Total Number of Endpoints:	155

All user specified timing constraints are met.

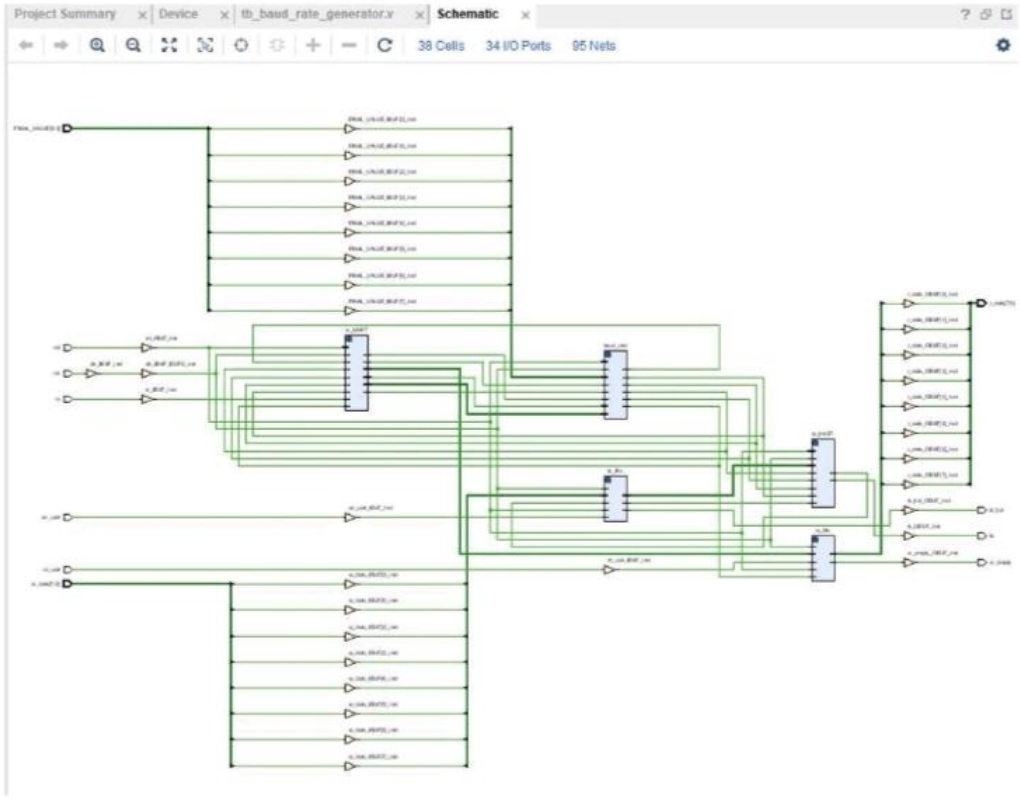
Implementation for TX:



UART wrapper:

```
module uart();
parameter Dbits = 8 //data bits
parameter SB_tick = 16 // stop bit tick
input clk, reset;

// receiver port
output [Dbits-1:0] r_data;
input rd_uart;
output rx_empty;
input rx;

// transmitter port
input [Dbits-1:0] w_data;
input wr_uart;
output tx_full;
output tx;

// baud rate generator
input [10:0] timer_final_value;


// timer as baud rate generator
wire tick;
timer_input #(.BITS(11)) baud_rate_generator(
    .clk(clk),
    .reset_n(reset_n),
    .enable(1'b1),
        .FINAL_VALUE(TIMER_FINAL_VALUE),
        .done(tick)
    );

   // Receiver
   wire rx_done_tick;
   wire [DBIT - 1: 0] rx_dout;
   uart_rx #(.DBIT(DBIT), .SB_TICK(SB_TICK)) receiver(
        .clk(clk),
        .reset_n(reset_n),
        .rx(rx),
        .s_tick(tick),
        .rx_done_tick(rx_done_tick),
        .rx_dout(rx_dout)
    );

   fifo_generator_0 rx_FIFO (
        .clk(clk),              // input wire clk
        .srst(~reset_n),        // input wire srst
        .din(rx_dout),          // input wire [7 : 0] din
        .wr_en(rx_done_tick),   // input wire wr_en
        .rd_en(rd_uart),        // input wire rd_en
        .dout(r_data),          // output wire [7 : 0] dout
        .full(),                // output wire full
        .empty(rx_empty)        // output wire empty
    );

    // Transmitter
    wire tx_fifo_empty, tx_done_tick;
    wire [DBIT - 1: 0] tx_din;
    uart_tx #(.DBIT(DBIT), .SB_TICK(SB_TICK)) transmitter(
        .clk(clk),
        .reset_n(reset_n),
        .tx_start(~tx_fifo_empty),
        .s_tick(tick),
        .tx_din(tx_din),
        .tx_done_tick(tx_done_tick),
        .tx(tx)
    );

    fifo_generator_0 tx_FIFO (
        .clk(clk),              // input wire clk
        .srst(~reset_n),        // input wire srst
        .din(w_data),           // input wire [7 : 0] din
        .wr_en(wr_uart),        // input wire wr_en
        .rd_en(tx_done_tick),   // input wire rd_en
        .dout(tx_din),          // output wire [7 : 0] dout
        .full(tx_full),         // output wire full
        .empty(tx_fifo_empty)   // output wire empty
    );
endmodule
```

RTL for UART wrapper:

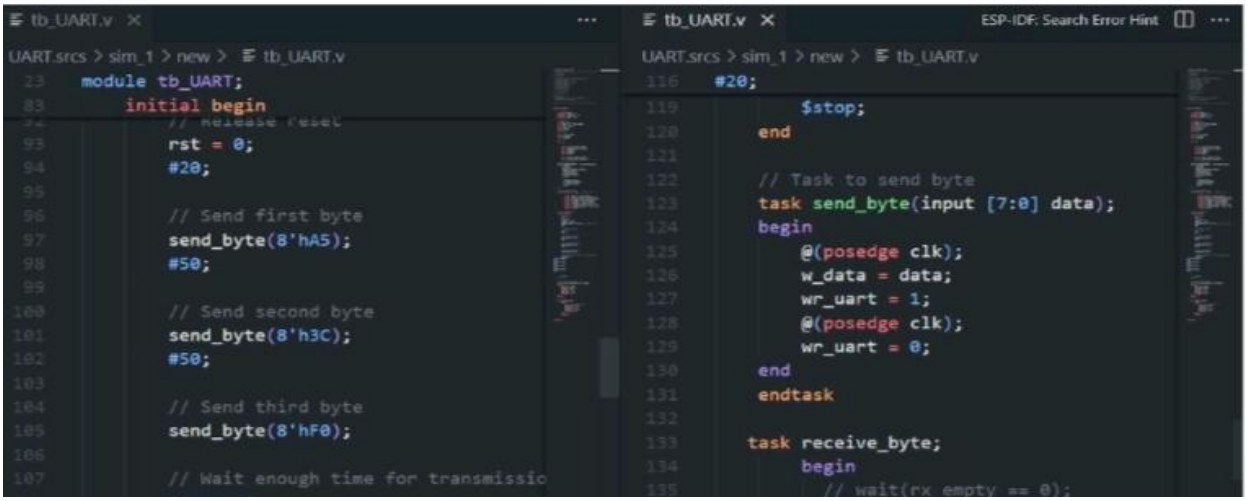


Testbench:

```
module tb_UART;

    // Parameters
    parameter DBITS = 8;
    parameter SB_TICK = 16;
    parameter BITS = 10;

    // Signals
    reg clk, rst;
    reg [BITS-1:0] FINAL_VALUE;
    reg [7:0] w_data;
    reg wr_uart;
    wire tx_full;
    wire tx;
    wire [7:0] r_data;
    wire rx_empty;
    reg  rd_uart;
    wire rx;
    wire [7:0] dout_to_tx_din;
    wire tx_done_tick_to_rd_en;
    wire tx_empty;
    wire tx_start;
    wire s_tick;
    wire [7:0] rx_dout_to_din ;
    wire rx_done_tick_to_wr_en ;

    // Instantiate UART
    UART #(.DBITS(DBITS), .SB_TICK(SB_TICK), .BITS(BITS)) uut (
        .clk(clk),
        .rst(rst),
        .FINAL_VALUE(FINAL_VALUE),
        .w_data(w_data),
        .wr_uart(wr_uart),
        .tx_full(tx_full),
        .tx(tx),

         .r_data(r_data),
         .rx_empty(rx_empty),
         .rd_uart(rd_uart),
         .rx(rx)
    );

    // Clock generation
    always #5 clk = ~clk; // 100MHz clock

        assign s_tick = uut.s_tick;
        assign dout_to_tx_din=uut.dout_to_tx_din;
        assign tx_done_tick_to_rd_en=uut.tx_done_tick_to_rd_en;
        assign tx_empty = uut.tx_empty;
        assign tx_start = uut.tx_start;
        assign rx_done_tick_to_wr_en= uut.rx_done_tick_to_wr_en;
        assign  rx_dout_to_din  = uut. rx_dout_to_din ;
        assign rx=tx;
```

```
    // Test procedure
    initial begin
        // Init
        clk = 0;
        rst = 1;
        wr_uart = 0;
        w_data = 8'h00;
        FINAL_VALUE = 10'd650; // small for fast simulation
        #50;

        // Release reset
        rst = 0;
        #20;

        // Send first byte
        send_byte(8'hA5);
        #50;

        // Send second byte
        send_byte(8'h3C);
        #50;

        // Send third byte
        send_byte(8'hF0);

        // Wait enough time for transmission
        #4000000;
receive_byte;
#50;
receive_byte;
#50;
receive_byte;
#50;
receive_byte;
#20;


        $stop;
    end

    // Task to send byte
    task send_byte(input [7:0] data);
    begin
        @(posedge clk);
        w_data = data;
        wr_uart = 1;
        @(posedge clk);
        wr_uart = 0;
    end
    endtask

     task receive_byte;
          begin
            // wait(rx_empty == 0);
              @(posedge clk);
              rd_uart = 1;
              @(posedge clk);
              rd_uart = 0;
          end
          endtask


  endmodule
```

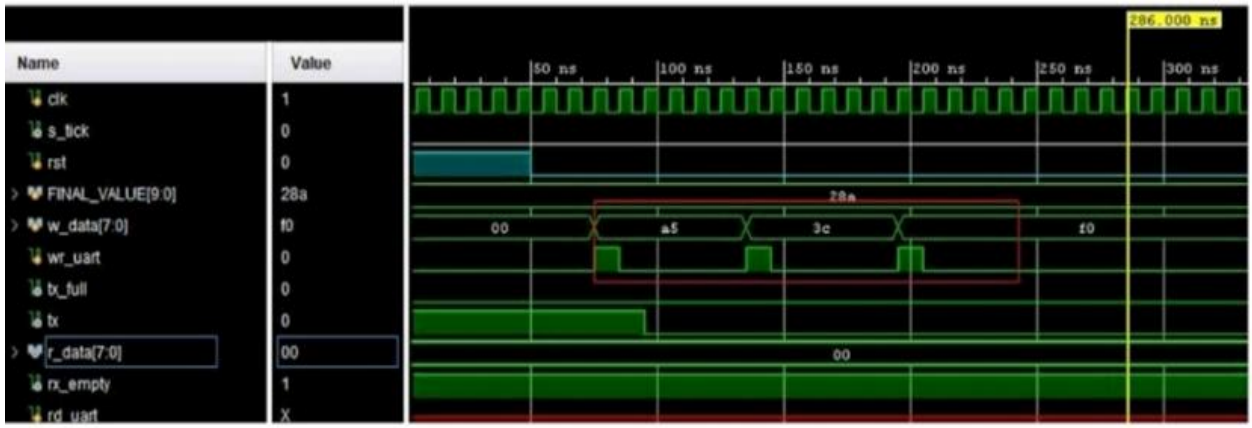
Waveform:



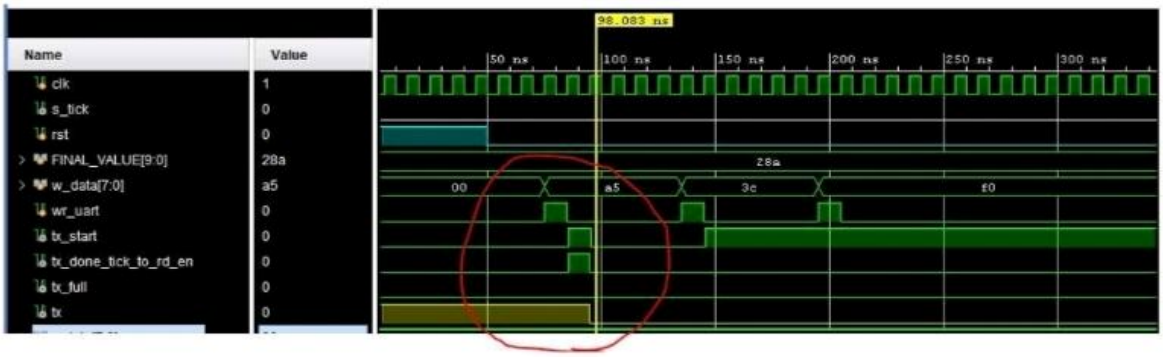
```
tb_UART.v
UART.srcs > sim_1 > new > tb_UART.v
23    module tb_UART;
83        initial begin
93            rst = 0;
94            #20;
95
96            // Send first byte
97            send_byte(8'hA5);
98            #50;
99
100           // Send second byte
101           send_byte(8'h3C);
102           #50;
103
104           // Send third byte
105           send_byte(8'hF0);
106
107           // Wait enough time for transmissio
```

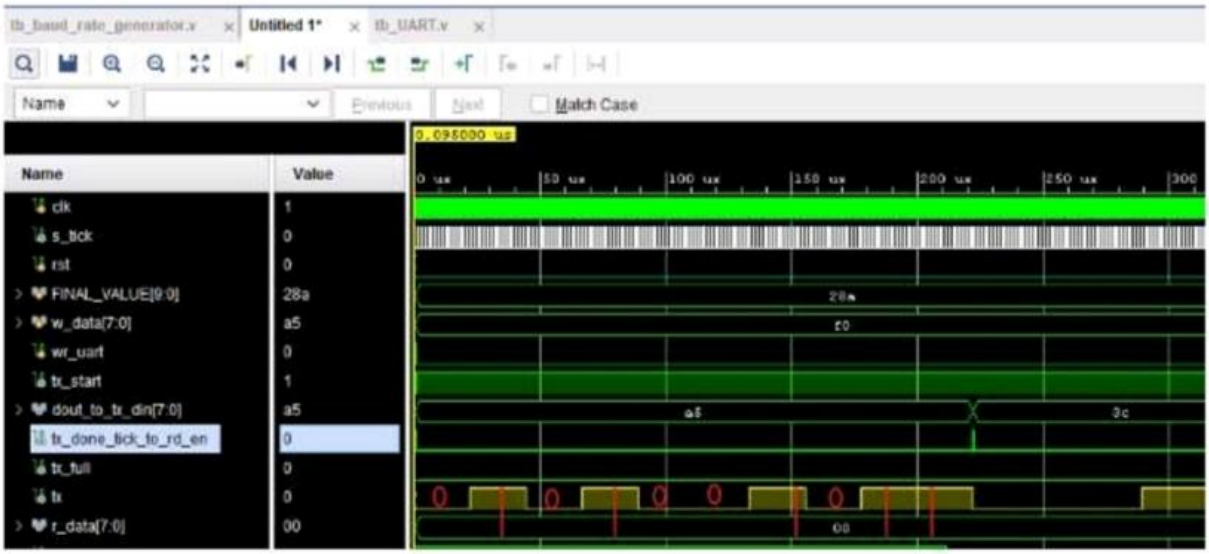
```
tb_UART.v                                   ESP-IDF: Search Error Hint
UART.srcs > sim_1 > new > tb_UART.v
116   #20;
119           $stop;
120       end
121
122       // Task to send byte
123       task send_byte(input [7:0] data);
124       begin
125           @(posedge clk);
126           w_data = data;
127           wr_uart = 1;
128           @(posedge clk);
129           wr_uart = 0;
130       end
131       endtask
132
133       task receive_byte;
134           begin
135               // wait(rx_empty == 0);
```

- Send to tx_FIFO: 5A, 3C, F0.



- once tx_FIFO not empty Tx_start take action and dout _to_Tx_din[7:0] and send data over Tx.

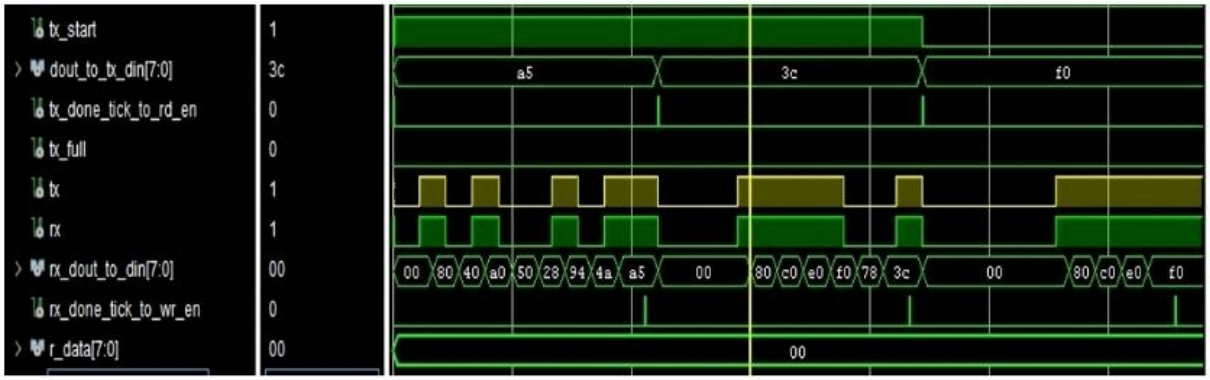




- 8'hA5 = 8'B10100101.

- Tx = start_bit(0) + 10100101 + stop bit(1).

- At same time Rx = Tx.



- Rx receives the data and stores it at rx_FIFO.