

The background of the slide features a subtle, light-colored circuit board pattern on a dark blue background. The pattern consists of numerous thin, white lines that form a complex, interconnected network of paths, resembling a printed circuit board (PCB) layout. These lines are more densely packed in the upper half of the slide and become sparser towards the bottom.

UART Design and Implementation using Verilog HDL

Modular Design, Verification, and Integration
with FIFOs

Prepared by: Mohamed Essam

Date: August 2025

Contents

1. Introduction	3
2. Design Methodology	3
2.1 Baud Rate Generator	3
2.2 Transmitter	3
2.3 Receiver	4
2.4 FIFO Buffers	5
3. System Integration	6
4. Verification	7
4.1 Testbench Setup	8
4.2 Waveform Analysis	8
5. Synthesis	12
5.1 Synthesis	12
5.2 Resource Utilization	12
5.3 Timing Report	13
6. Results and Discussion	13
7. Conclusion	13

1. Introduction

In this project, a Universal Asynchronous Receiver/Transmitter (UART) was designed and implemented using Verilog HDL. The design follows a modular approach, where each block (Transmitter, Receiver, Baud Rate Generator, and FIFO Buffers) was implemented, tested individually with testbenches, and then integrated into a top-level UART system.

2. Design Methodology

The UART design process was divided into smaller modules. Each module was designed, verified using testbenches, and then connected together to form the complete UART system.

2.1 Baud Rate Generator

Function: Generates sampling ticks (s_tick) for synchronization.

Inputs: clk, rst, FINAL_VALUE

Outputs: done (s_tick)

```
module baud_rate_generator #(parameter BITS=10)(
    input clk ,rst,en,
    input [BITS-1:0]FINAL_VALUE,
    output done
);

    reg [BITS-1:0] Q_reg,Q_next;
    always @(posedge clk or posedge rst) begin
        if(rst)
            Q_reg<={BITS{1'b0}};
        else if(en)
            Q_reg<=Q_next;
        end
    always @(*) begin
        Q_next=done?{BITS{1'b0}}: Q_reg+1;
    end
    assign done =( Q_reg==FINAL_VALUE);

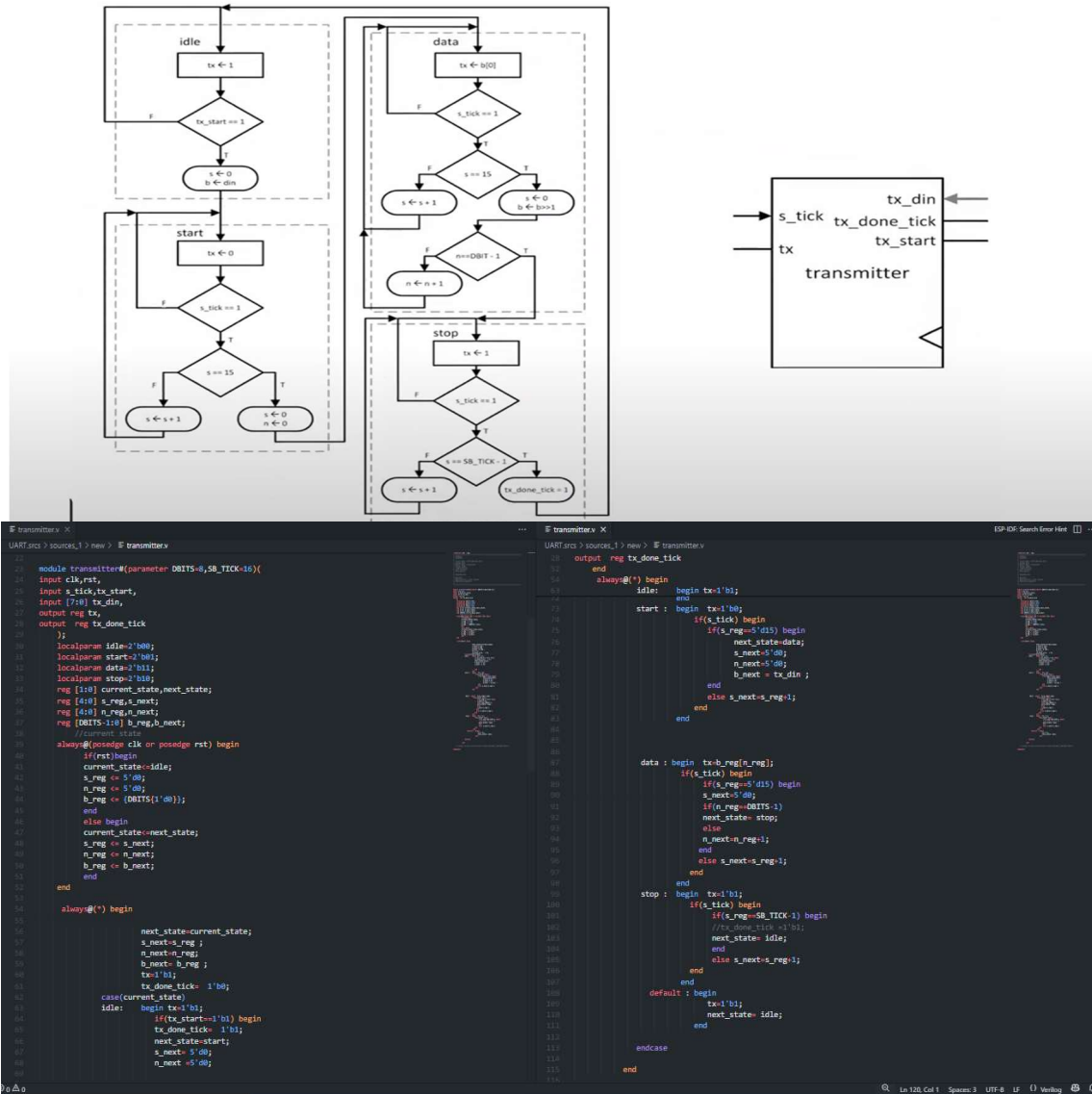
endmodule
```

2.2 Transmitter

Function: Serializes parallel data and transmits with start and stop bits.

Inputs: clk, rst, tx_din, tx_start, s_tick

Outputs: tx, tx_done_tick

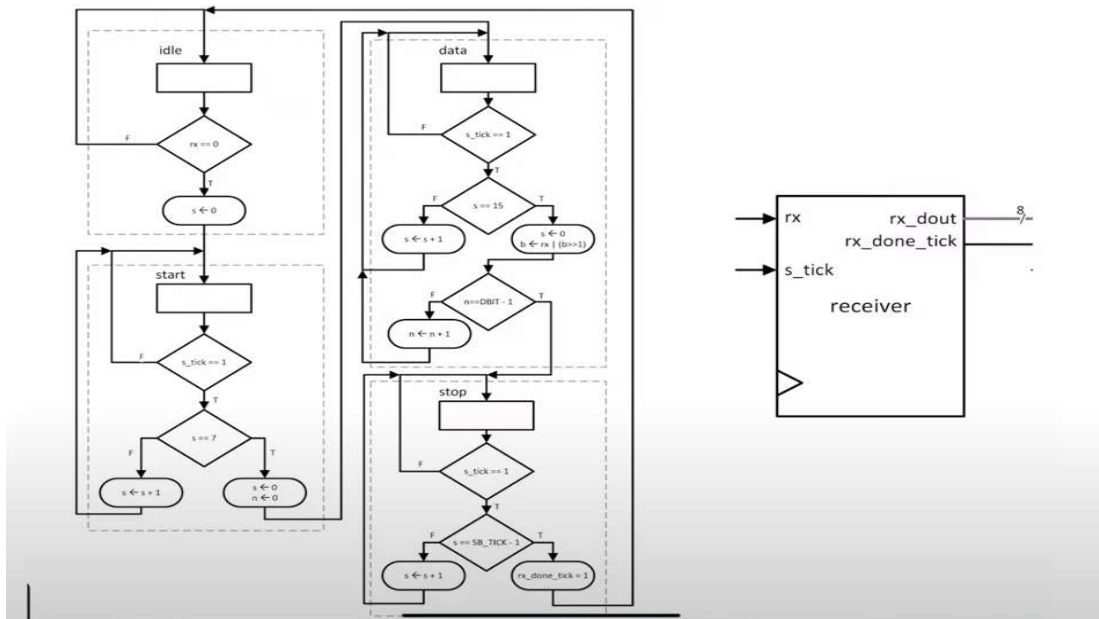


2.3 Receiver

Function: Deserializes incoming serial data into parallel format.

Inputs: clk, rst, rx, s_tick

Outputs: rx_dout, rx_done_tick



```

// receiver.v
module receiver(parameter DBITS=8, SB_TICK=16)(
    input clk, rst, rx_tick,
    output [7:0] rx_dout,
    output reg rx_done_tick
);
    localparam idle=2'b00;
    localparam start=2'b01;
    localparam data=2'b11;
    localparam stop=2'b10;
    reg [1:0] current_state,next_state;
    reg [4:0] s_reg,s_next;
    reg [4:0] n_reg,n_next;
    reg [DBITS-1:0] b_reg,b_next;
    //current state
    always@(posedge clk or posedge rst) begin
        if(rst)begin
            current_state=idle;
        end
        else begin
            current_state=next_state;
            s_reg <= s_next;
            n_reg <= n_next;
            b_reg <= b_next;
        end
    end
    //next state
    always@(*) begin
        next_state=current_state;
        s_next=s_reg;
        n_next=n_reg;
        b_next=b_reg;
        rx_done_tick=1'b0;
        case(current_state)
            idle: begin
                if(rx==1'b0) begin
                    next_state=start;
                    s_next=4'd0;
                    n_next=4'd0;
                    b_next={DBITS{1'b0}};
                end
            end
            start: begin
                if(s_tick) begin
                    if(s_reg==4'd7) begin
                        next_state=data;
                        s_next=4'd0;
                        n_next=4'd0;
                    end
                    else s_next=s_reg+1;
                end
            end
            data: begin
                if(s_tick) begin
                    if(s_reg==4'd15) begin
                        s_next=4'd0;
                        b_next={rx,b_reg[DBITS-1:1]};
                        if(n_reg==DBITS-1)
                            next_state=stop;
                        else
                            n_next=n_reg+1;
                    end
                    else s_next=s_reg+1;
                end
            end
            stop: begin
                if(s_tick) begin
                    if(s_reg==SB_TICK-1) begin
                        next_state=idle;
                        rx_done_tick=1'b1;
                        else s_next=s_reg+1;
                    end
                end
            end
            default: next_state=idle;
        endcase
    end
    assign rx_dout=b_reg;
    // assign rx_done_tick=(current_state==stop)&&(s_reg==SB_TICK-1);
endmodule

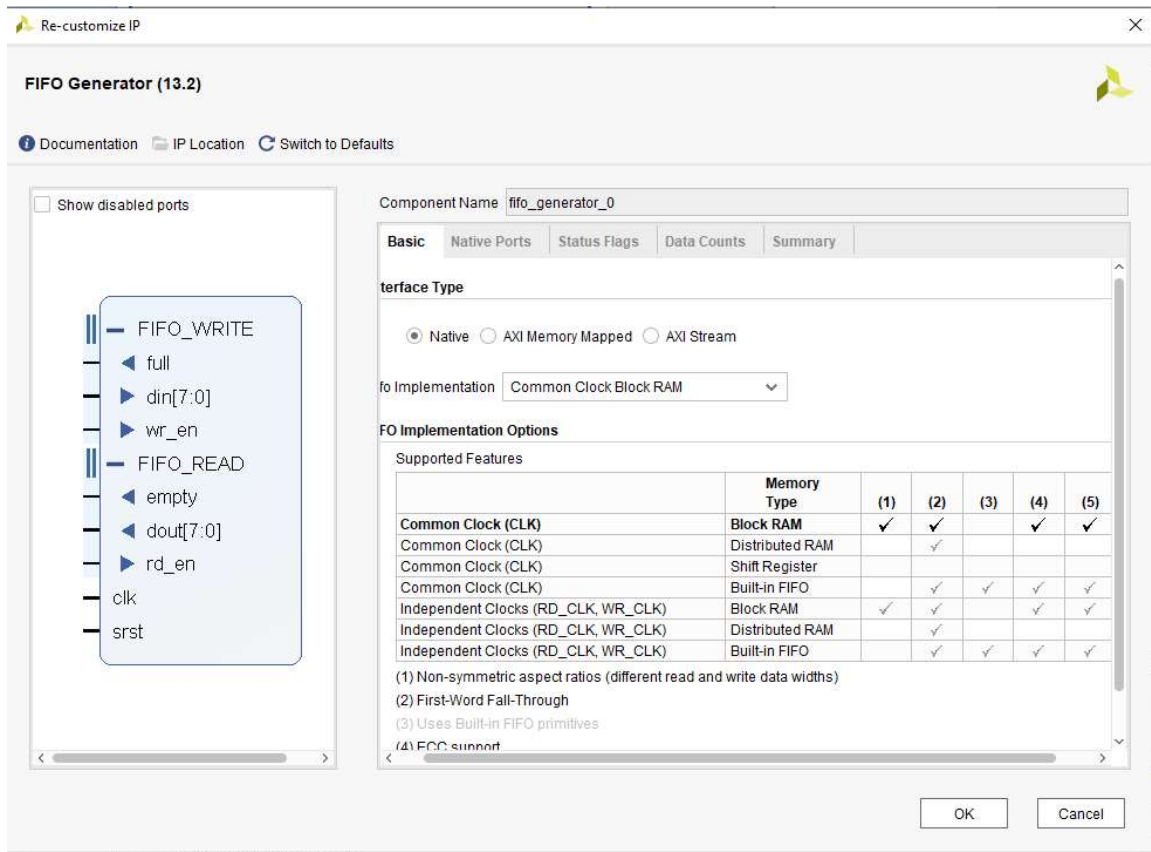
```

2.4 FIFO Buffers

Function: Buffering mechanism for transmit and receive data.

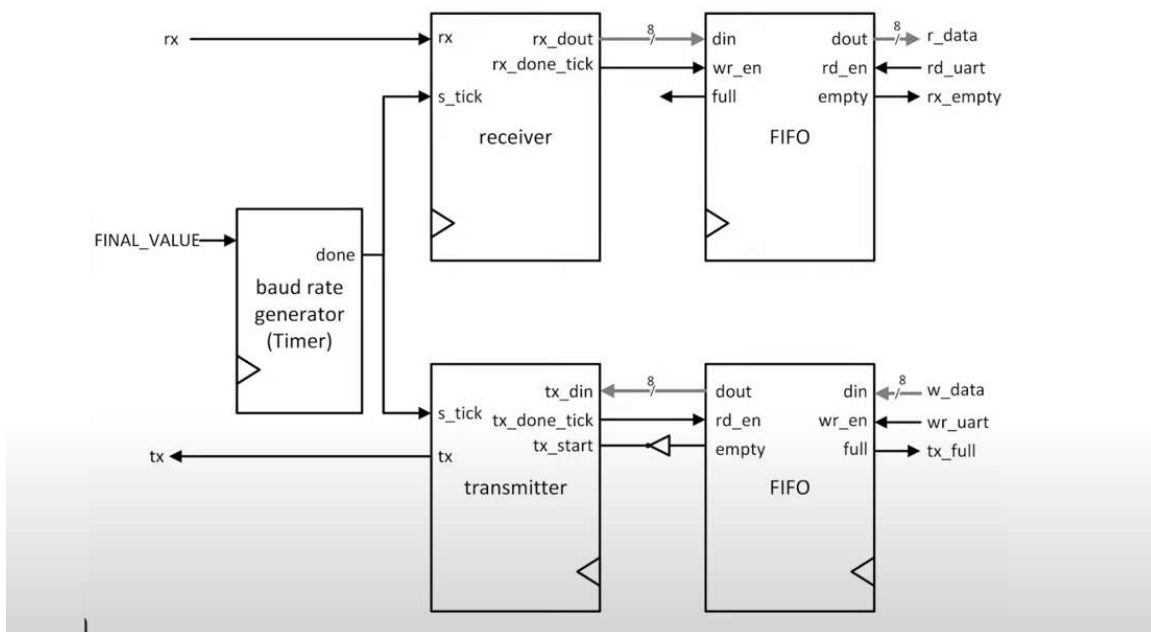
Inputs: din, wr_en, rd_en

Outputs: dout, full, empty



3. System Integration

The transmitter, receiver, baud rate generator, and FIFO buffers were integrated into a top-level UART design. The final UART module allows bidirectional data transfer with buffering and configurable baud rate.



```

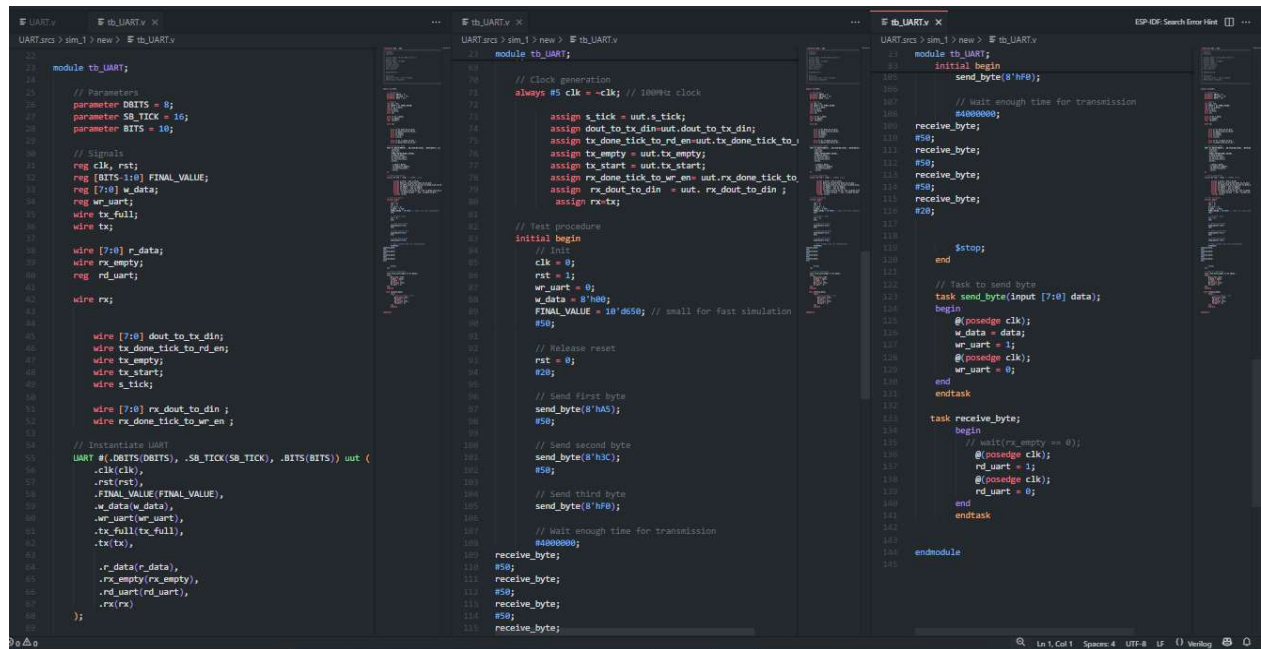
1  module UART #(parameter DBITS=8, SB_TICK=16, BITS=16){
2    input clk, rst,
3    input [BITS-1:0] FINAL_VALUE,
4    input [7:0] w_data,
5    input wr_uart,
6    output tx_full, tx,
7    output [7:0] r_data,
8    output rx_empty,
9    input rd_uart,
10   input rx;
11
12   wire [7:0] dout_to_tx_din;
13   wire tx_done_tick_to_rd_en;
14   wire tx_empty;
15   wire tx_start;
16   wire s_tick;
17
18   wire [7:0] rx_dout_to_din;
19   wire rx_done_tick_to_wr_en;
20
21   fifo_generator_0_tx_fifo (
22     .clk(clk), // input wire clk
23     .rst(rst), // input wire rst
24     .din(w_data), // input wire [7 : 0] din
25     .wr_en(wr_uart), // input wire wr_en
26     .rd_en(tx_done_tick_to_rd_en), // input wire rd_en
27     .dout(dout_to_tx_din), // output wire [7 : 0] dout
28     .full(tx_full), // output wire full
29     .empty(tx_empty) // output wire empty
30   );
31
32   assign tx_start=tx_empty;
33
34   transmitter #(DBITS(DBITS), .SB_TICK(SB_TICK)) tx_UART (
35     .clk(clk),
36     .rst(rst),
37     .s_tick(s_tick),
38     .tx_start(tx_start),
39     .tx_din(dout_to_tx_din),
40     .tx(tx),
41     .tx_done_tick(tx_done_tick_to_rd_en)
42   );
43
44   baud_rate_generator #(
45     .BITS(DBITS)
46   ) baud_rate (
47     .clk(clk), // input wire clk
48     .rst(rst), // input wire rst
49     .rat(rst),
50     .en(1'b1),
51     .FINAL_VALUE(FINAL_VALUE),
52     .done(s_tick)
53   );
54
55   receiver #(DBITS(8), .SB_TICK(16)) rx_UART (
56     .clk(clk), // input wire clk
57     .rst(rst), // input wire rst
58     .rd_uart(rd_uart),
59     .rx(rx),
60     .s_tick(s_tick),
61     .rx_dout(rx_dout_to_din),
62     .rx_done_tick(rx_done_tick_to_wr_en)
63   );
64
65   fifo_generator_0_rx_fifo (
66     .clk(clk), // input wire clk
67     .rst(rst), // input wire rst
68     .din(rx_dout_to_din), // input wire [7 : 0] din
69     .wr_en(rx_done_tick_to_wr_en), // input wire wr_en
70     .rd_en(rd_uart), // input wire rd_en
71     .dout(r_data), // output wire [7 : 0] dout
72     .full(rx_full), // output wire full
73     .empty(rx_empty) // output wire empty
74   );
75
76 endmodule

```

4. Verification

Each module was verified using individual testbenches. Simulation results confirmed correct serialization, deserialization, and data buffering. Waveforms and FSM diagrams were analyzed for validation.

4.1 Testbench Setup



The screenshot shows three separate Verilog testbench files for a UART module. The first file on the left defines the module parameters and signals. The middle file contains the test procedure logic, including clock generation and signal assignments. The third file on the right shows the task definitions for sending and receiving bytes. All three files are part of a project named 'UART.v'.

```
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),
    .rst(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);
    #50;

    // Wait enough time for transmission
    #4000000;

    receive_byte;
    #50;
    receive_byte;
    #50;
    receive_byte;
    #50;
    receive_byte;
    #50;
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(r_x_empty == 0);
    @(posedge clk);
    rd_uart = 1;
    @(posedge clk);
    rd_uart = 0;
end
endtask
endmodule
```

4.2 Waveform Analysis

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

    // Release reset
    rst = 0;
    #20;
```

Baud Rate=9600

tb_uart.v

UART.srcs > sim_1 > new > tb_uart.v

```

23 module tb_uart;
83   initial begin
92     // Release reset
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
108    #4000000;

```

tb_uart.v

UART.srcs > sim_1 > new > tb_uart.v

```

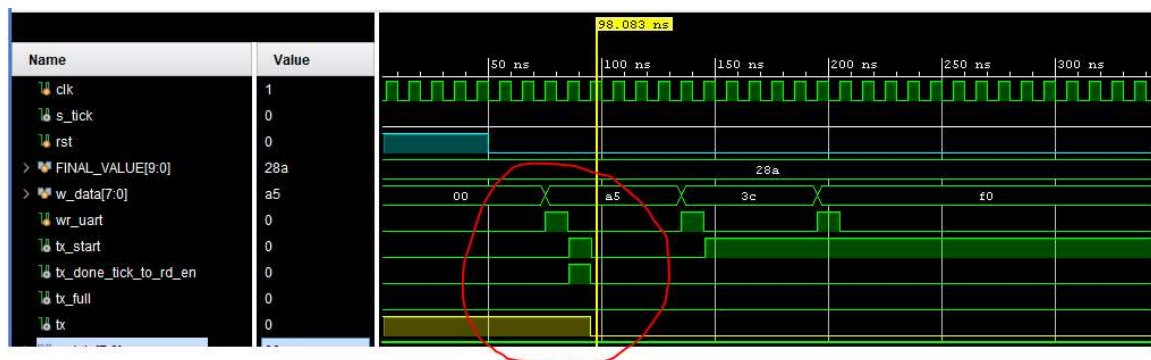
116 #20;
119 $stop;
120 end
121
122 // Task to send byte
123 task send_byte(input [7:0] data);
124 begin
125     @(posedge clk);
126     w_data = data;
127     wr_uart = 1;
128     @(posedge clk);
129     wr_uart = 0;
130 end
131 endtask
132
133 task receive_byte;
134 begin
135     // wait(rx_empty == 0);
136     @(posedge clk);

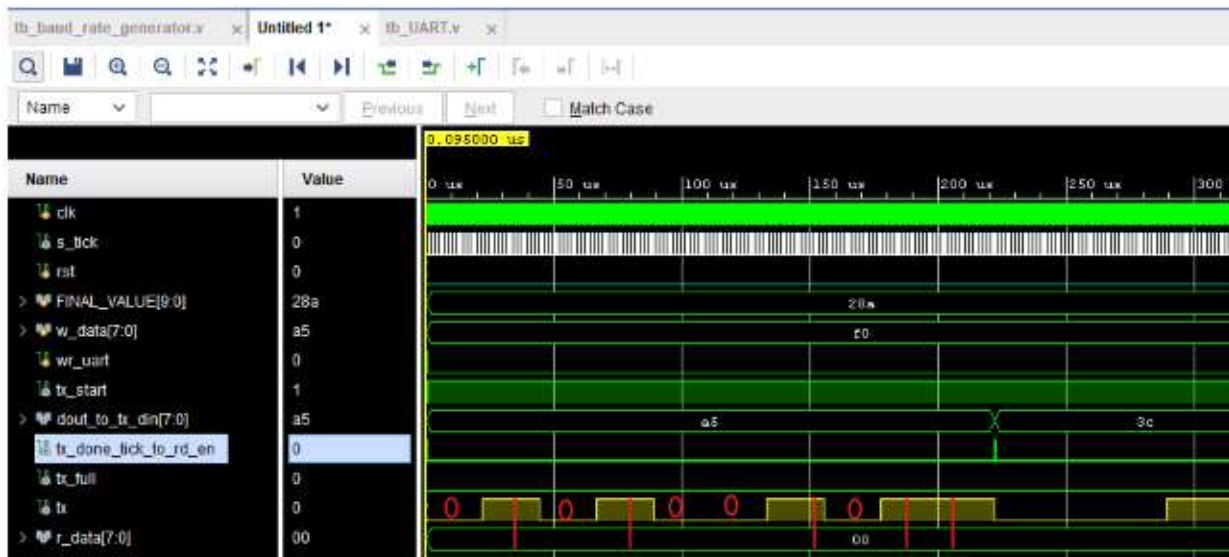
```

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

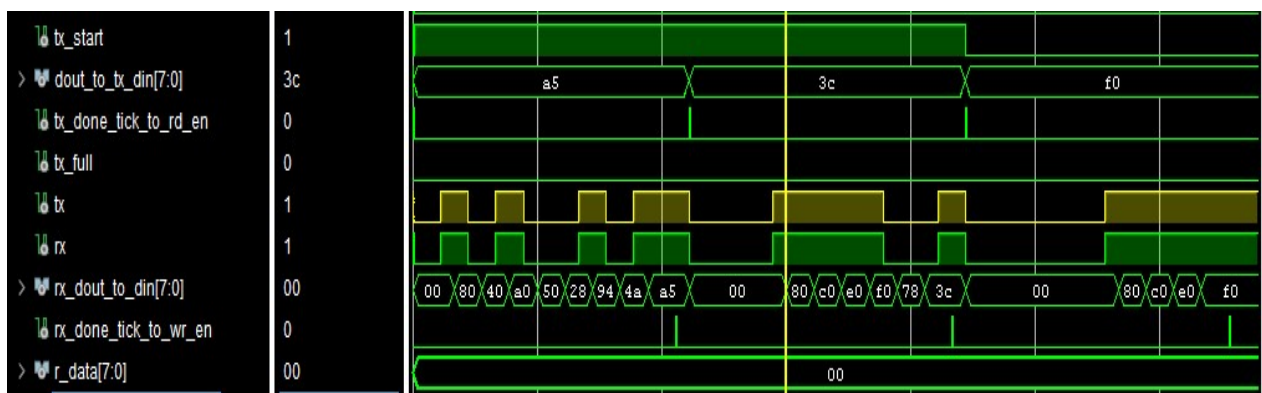




8'hA5 = 8'b10100101

Tx=start_bit(0)+10100101 +stop_bit(1)

At same time rx =tx



Rx receive the data and store it at rx_fifo

```

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

// Wait enough time for transmissio
#4000000;

receive_byte;
#50;
receive_byte;
#50;
receive_byte;
#50;
receive_byte;
#20;

$stop;

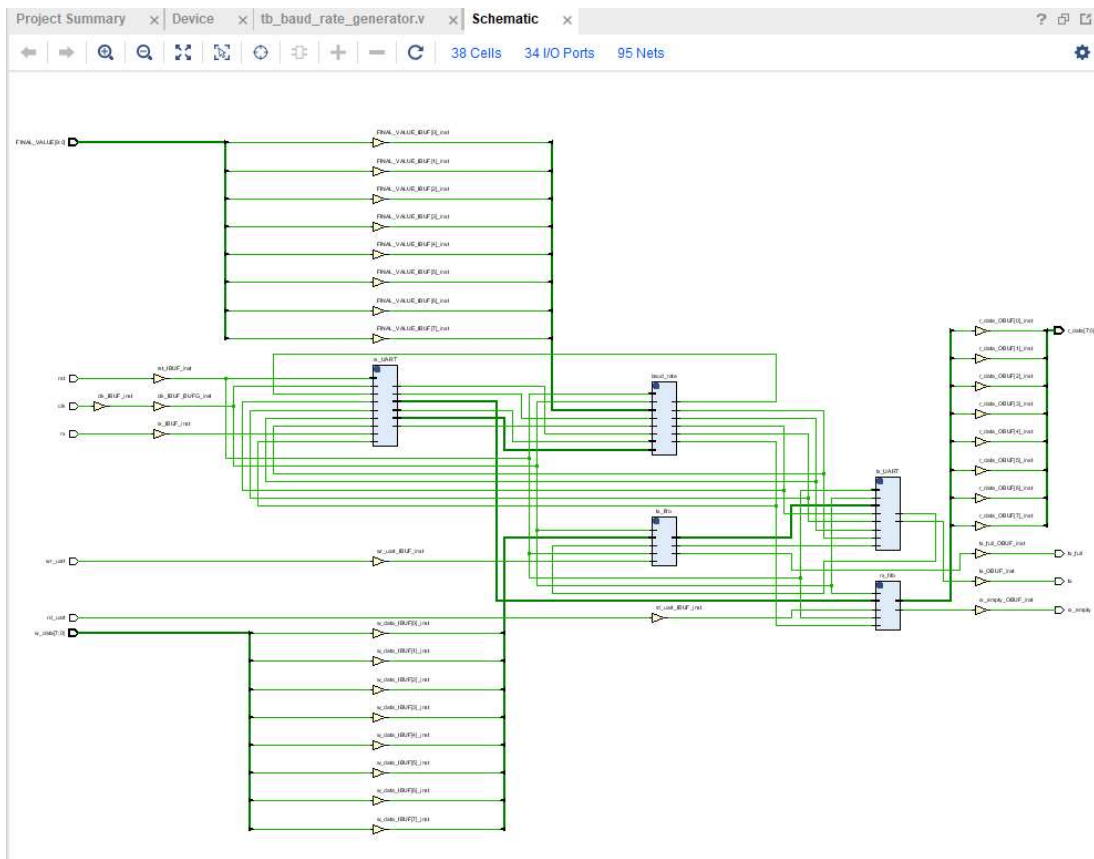
129 wr_uart = 0;
130 end
131 endtask
132
133 task receive_byte;
134 begin
135     // wait(rx_empty == 0);
136     @(posedge clk);
137     rd_uart = 1;
138     @(posedge clk);
139     rd_uart = 0;
140 end
141 endtask
142
143
144 endmodule
145

```



5. Synthesis

5.1 Synthesis



5.2 Resource Utilization

Tcl Console	Messages	Log	Reports	Design Runs	Utilization	Timing																																										
Hierarchy																																																
<table><tr><th>Name</th><th>Slice LUTs (134600)</th><th>Slice Registers (269200)</th><th>Block RAM Tile (365)</th><th>Bonded IOB (500)</th><th>BUFGCTRL (32)</th></tr><tr><td>UART</td><td>136</td><td>120</td><td>1</td><td>32</td><td>1</td></tr><tr><td> baud_rate (baud_rate_...</td><td>14</td><td>8</td><td>0</td><td>0</td><td>0</td></tr><tr><td> rx_fifo (fifo_generator_0)</td><td>34</td><td>36</td><td>0.5</td><td>0</td><td>0</td></tr><tr><td> rx_UART (receiver)</td><td>30</td><td>20</td><td>0</td><td>0</td><td>0</td></tr><tr><td> tx_fifo (fifo_generator_0)</td><td>34</td><td>36</td><td>0.5</td><td>0</td><td>0</td></tr><tr><td> tx_UART (transmitter)</td><td>24</td><td>20</td><td>0</td><td>0</td><td>0</td></tr></table>							Name	Slice LUTs (134600)	Slice Registers (269200)	Block RAM Tile (365)	Bonded IOB (500)	BUFGCTRL (32)	UART	136	120	1	32	1	baud_rate (baud_rate_...	14	8	0	0	0	rx_fifo (fifo_generator_0)	34	36	0.5	0	0	rx_UART (receiver)	30	20	0	0	0	tx_fifo (fifo_generator_0)	34	36	0.5	0	0	tx_UART (transmitter)	24	20	0	0	0
Name	Slice LUTs (134600)	Slice Registers (269200)	Block RAM Tile (365)	Bonded IOB (500)	BUFGCTRL (32)																																											
UART	136	120	1	32	1																																											
baud_rate (baud_rate_...	14	8	0	0	0																																											
rx_fifo (fifo_generator_0)	34	36	0.5	0	0																																											
rx_UART (receiver)	30	20	0	0	0																																											
tx_fifo (fifo_generator_0)	34	36	0.5	0	0																																											
tx_UART (transmitter)	24	20	0	0	0																																											

5.3 Timing Report

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.979 ns	Worst Hold Slack (WHS):	0.130 ns	Worst Pulse Width Slack (WPWS):	4.500 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:	271	Total Number of Endpoints:	271	Total Number of Endpoints:	125

All user specified timing constraints are met.

6. Results and Discussion

The UART system successfully transmitted and received data at different baud rates. FIFO ensured smooth data flow without data loss. The modular verification approach simplified debugging and improved reliability.

7. Conclusion

This project demonstrated a modular UART implementation in Verilog HDL. Future work may include parity bit support, error detection, and extension for multi-channel UARTs.