Custom APB UART IP Design

Prepared by

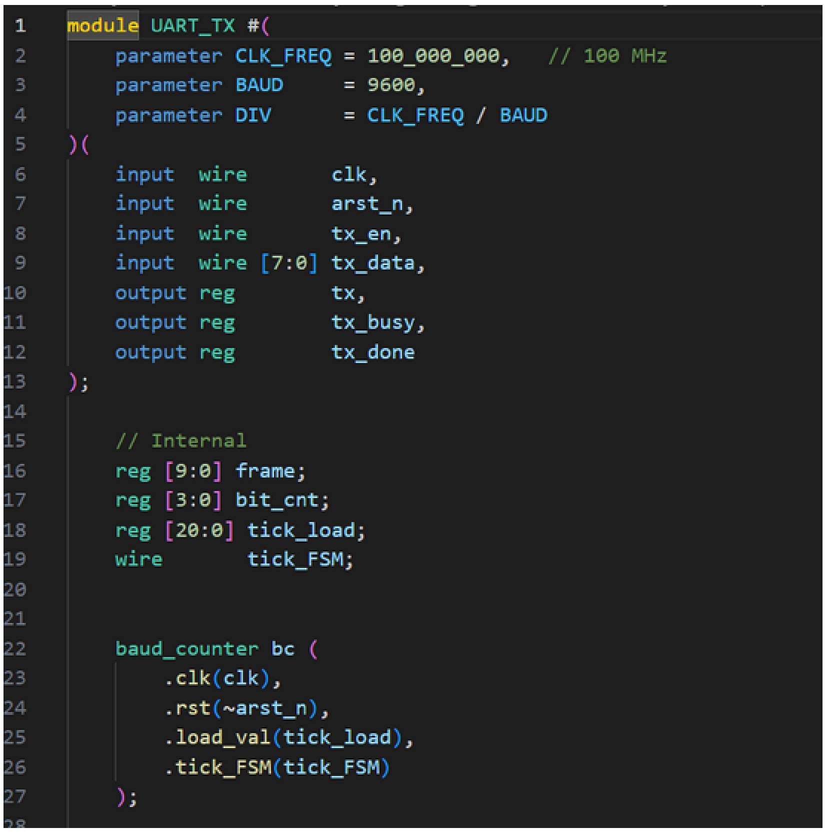
Aly Mohammed Hussein 22p0018

1.0 Introduction

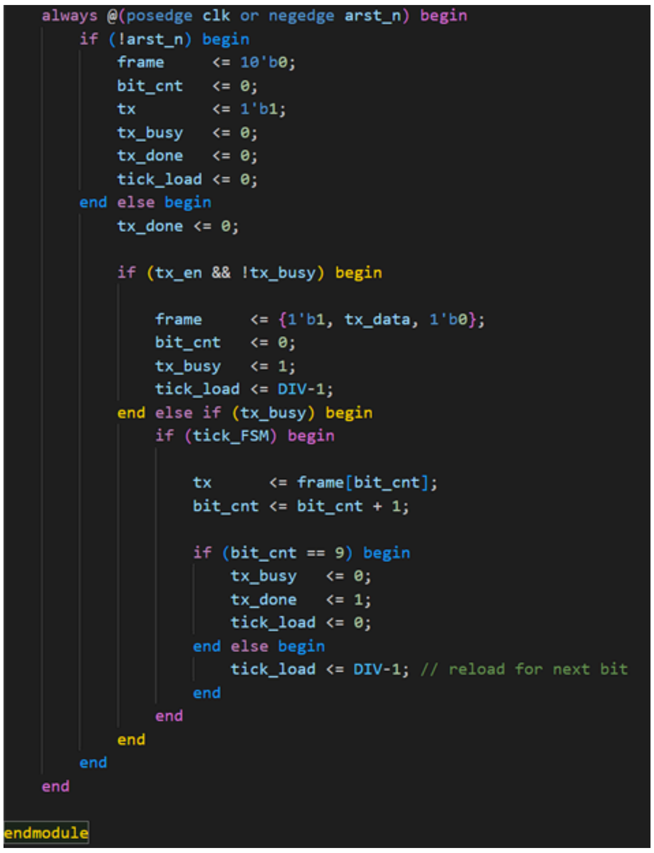
This project focuses on the design of a Universal Asynchronous Receiver/Transmitter (UART) wrapped with an AMBA APB slave interface.
The UART provides reliable serial communication, while the APB wrapper enables memory-mapped register access from a processor in an SoC.
Key objectives:
Implement UART transmitter (TX) and receiver (RX) in Verilog.
Implement a baud rate generator for precise timing.
Implement an APB slave logic that exposes UART control/status/data registers to a system bus.
Verify operation using testbenches for TX, RX, and APB wrapper.

2.0 Design Analysis

UART Transmitter (TX)



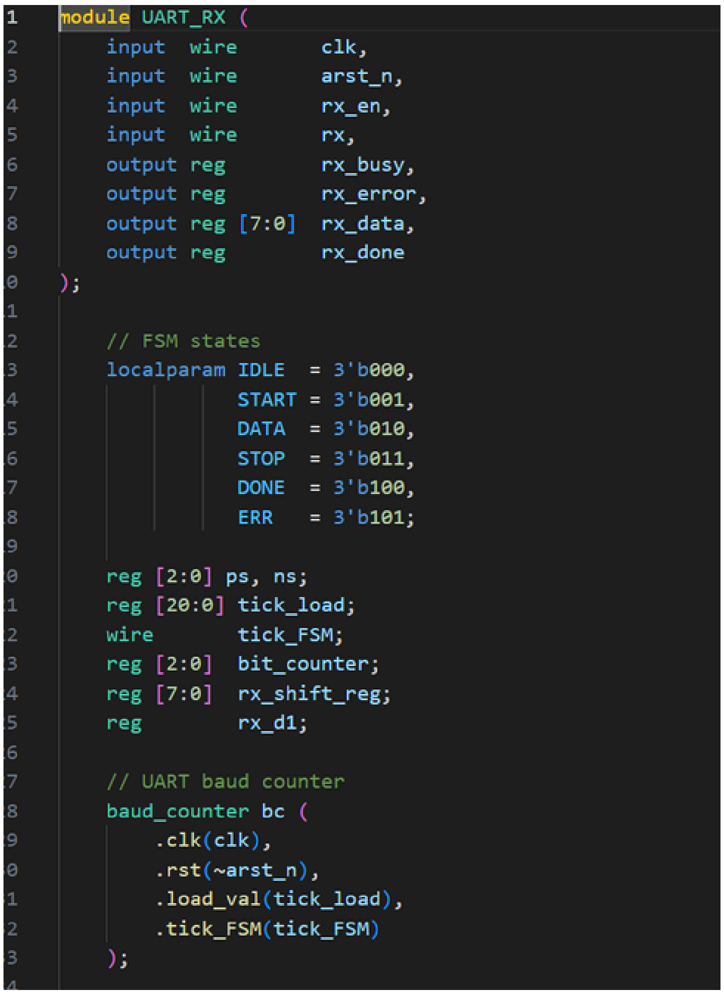
```verilog
module UART_TX #(
    parameter CLK_FREQ = 100_000_000,   // 100 MHz
    parameter BAUD     = 9600,
    parameter DIV      = CLK_FREQ / BAUD
)(
    input  wire       clk,
    input  wire       arst_n,
    input  wire       tx_en,
    input  wire [7:0] tx_data,
    output reg        tx,
    output reg        tx_busy,
    output reg        tx_done
);

    // Internal
    reg [9:0] frame;
    reg [3:0] bit_cnt;
    reg [20:0] tick_load;
    wire        tick_FSM;


    baud_counter bc (
        .clk(clk),
        .rst(~arst_n),
        .load_val(tick_load),
        .tick_FSM(tick_FSM)
    );
```



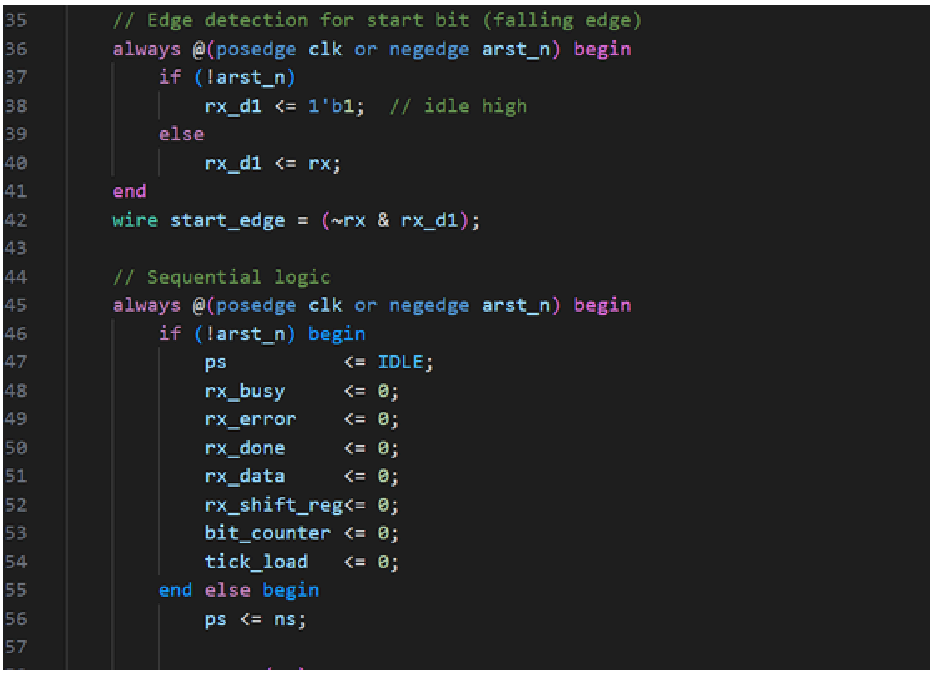
```verilog
    always @(posedge clk or negedge arst_n) begin
        if (!arst_n) begin
            frame     <= 10'b0;
            bit_cnt   <= 0;
            tx        <= 1'b1;
            tx_busy   <= 0;
            tx_done   <= 0;
            tick_load <= 0;
        end else begin
            tx_done <= 0;

            if (tx_en && !tx_busy) begin

                frame     <= {1'b1, tx_data, 1'b0};
                bit_cnt   <= 0;
                tx_busy   <= 1;
                tick_load <= DIV-1;
            end else if (tx_busy) begin
                if (tick_FSM) begin

                    tx      <= frame[bit_cnt];
                    bit_cnt <= bit_cnt + 1;

                    if (bit_cnt == 9) begin
                        tx_busy   <= 0;
                        tx_done   <= 1;
                        tick_load <= 0;
                    end else begin
                        tick_load <= DIV-1; // reload for next bit
                    end
                end
            end
        end
    end

endmodule
```

The transmitter appends a start bit (0), sends 8 data bits (LSB first), then a stop bit (1).
Data is loaded into a 10-bit shift frame: {stop_bit, tx_data[7:0], start_bit}.
Transmission timing is controlled by the baud_counter.
The tx_busy signal remains high during transmission, and tx_done is asserted when complete.

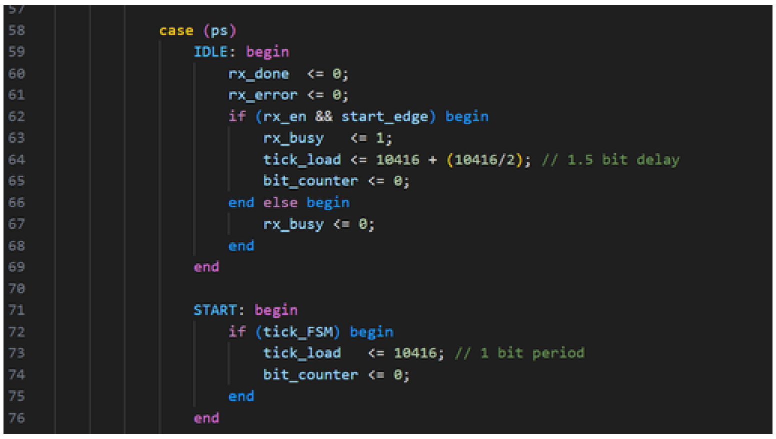
UART Receiver (RX)



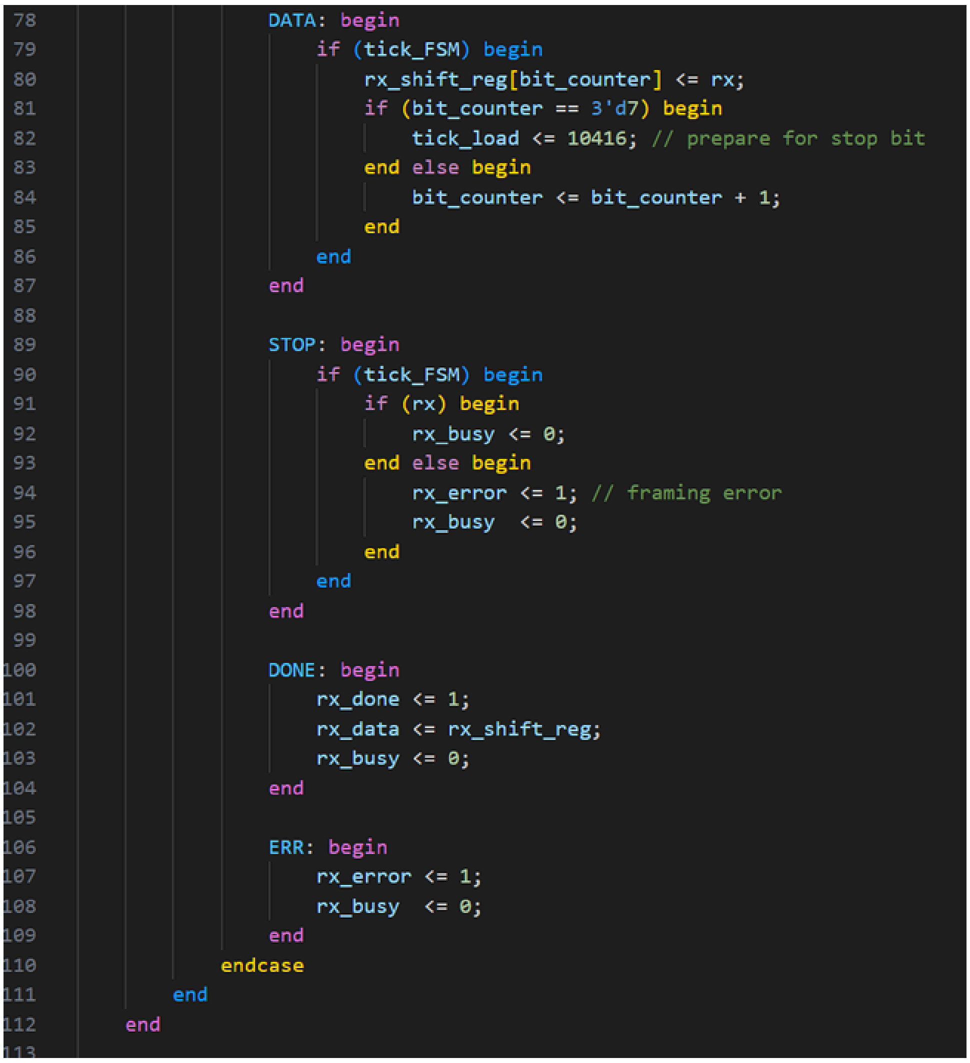
```verilog
module UART_RX (
    input  wire        clk,
    input  wire        arst_n,
    input  wire        rx_en,
    input  wire        rx,
    output reg         rx_busy,
    output reg         rx_error,
    output reg [7:0]   rx_data,
    output reg         rx_done
);

    // FSM states
    localparam IDLE  = 3'b000,
               START = 3'b001,
               DATA  = 3'b010,
               STOP  = 3'b011,
               DONE  = 3'b100,
               ERR   = 3'b101;

    reg [2:0] ps, ns;
    reg [20:0] tick_load;
    wire       tick_FSM;
    reg [2:0]  bit_counter;
    reg [7:0]  rx_shift_reg;
    reg        rx_d1;

    // UART baud counter
    baud_counter bc (
        .clk(clk),
        .rst(~arst_n),
        .load_val(tick_load),
        .tick_FSM(tick_FSM)
    );
```



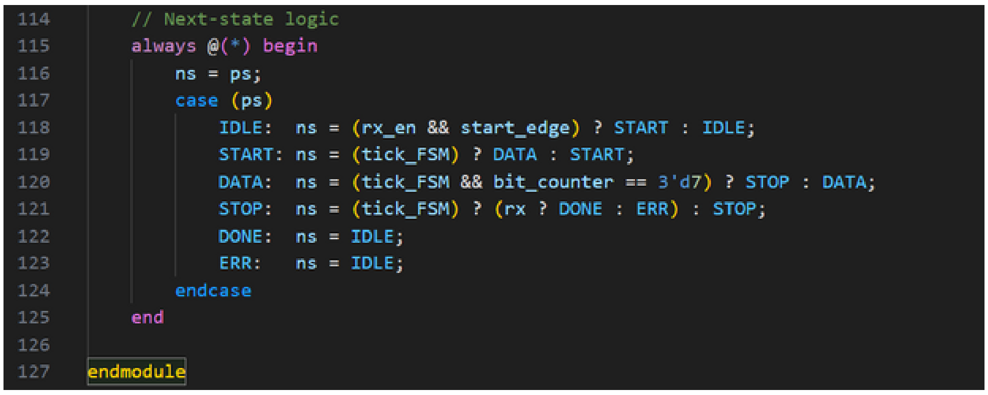
```verilog
    // Edge detection for start bit (falling edge)
    always @(posedge clk or negedge arst_n) begin
        if (!arst_n)
            rx_d1 <= 1'b1;  // idle high
        else
            rx_d1 <= rx;
    end
    wire start_edge = (~rx & rx_d1);

    // Sequential logic
    always @(posedge clk or negedge arst_n) begin
        if (!arst_n) begin
            ps          <= IDLE;
            rx_busy     <= 0;
            rx_error    <= 0;
            rx_done     <= 0;
            rx_data     <= 0;
            rx_shift_reg<= 0;
            bit_counter <= 0;
            tick_load   <= 0;
        end else begin
            ps <= ns;

```



```
            case (ps)
                IDLE: begin
                    rx_done  <= 0;
                    rx_error <= 0;
                    if (rx_en && start_edge) begin
                        rx_busy   <= 1;
                        tick_load <= 10416 + (10416/2); // 1.5 bit delay
                        bit_counter <= 0;
                    end else begin
                        rx_busy <= 0;
                    end
                end

                START: begin
                    if (tick_FSM) begin
                        tick_load   <= 10416; // 1 bit period
                        bit_counter <= 0;
                    end
                end
```



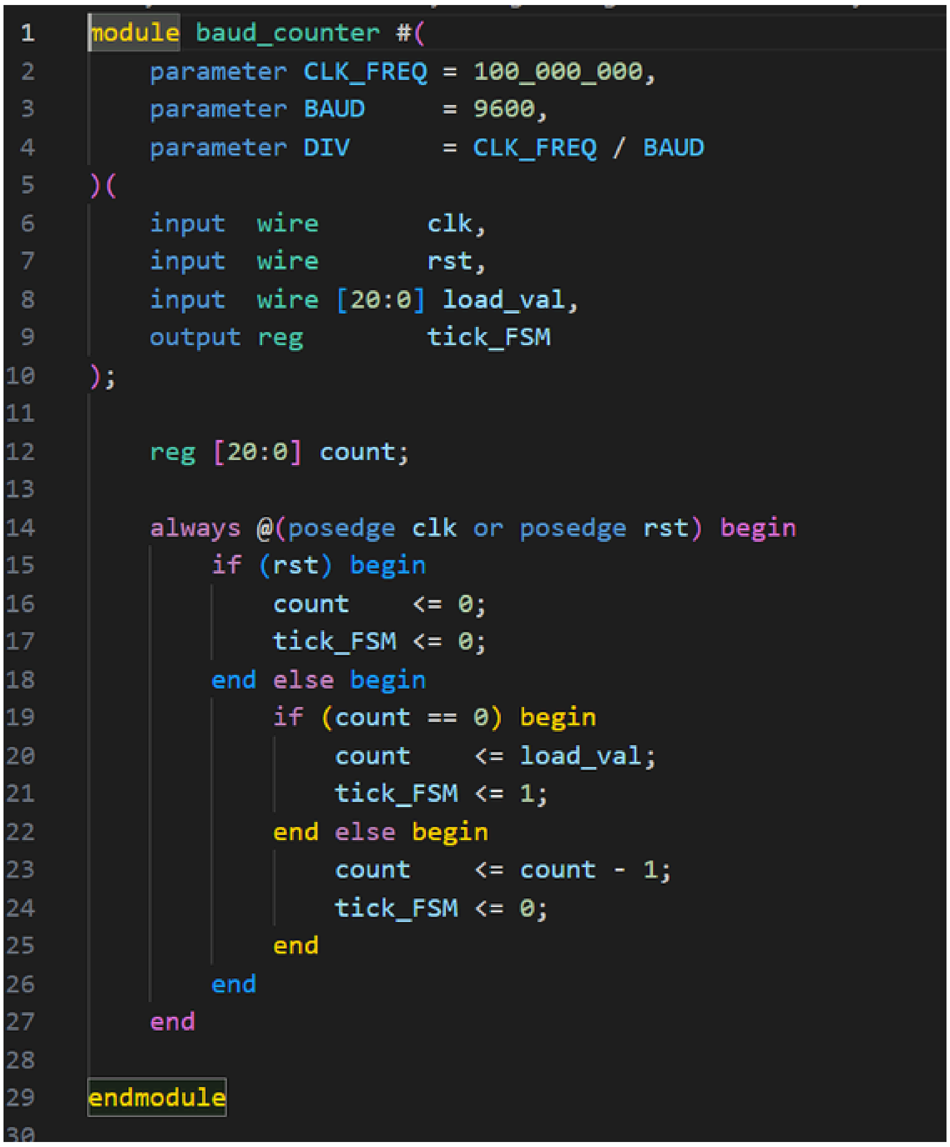
```
                DATA: begin
                    if (tick_FSM) begin
                        rx_shift_reg[bit_counter] <= rx;
                        if (bit_counter == 3'd7) begin
                            tick_load <= 10416; // prepare for stop bit
                        end else begin
                            bit_counter <= bit_counter + 1;
                        end
                    end
                end

                STOP: begin
                    if (tick_FSM) begin
                        if (rx) begin
                            rx_busy <= 0;
                        end else begin
                            rx_error <= 1; // framing error
                            rx_busy  <= 0;
                        end
                    end
                end

                DONE: begin
                    rx_done <= 1;
                    rx_data <= rx_shift_reg;
                    rx_busy <= 0;
                end

                ERR: begin
                    rx_error <= 1;
                    rx_busy  <= 0;
                end
            endcase
        end
    end
```



```
    // Next-state logic
    always @(*) begin
        ns = ps;
        case (ps)
            IDLE:  ns = (rx_en && start_edge) ? START : IDLE;
            START: ns = (tick_FSM) ? DATA : START;
            DATA:  ns = (tick_FSM && bit_counter == 3'd7) ? STOP : DATA;
            STOP:  ns = (tick_FSM) ? (rx ? DONE : ERR) : STOP;
            DONE:  ns = IDLE;
            ERR:   ns = IDLE;
        endcase
    end

endmodule
```

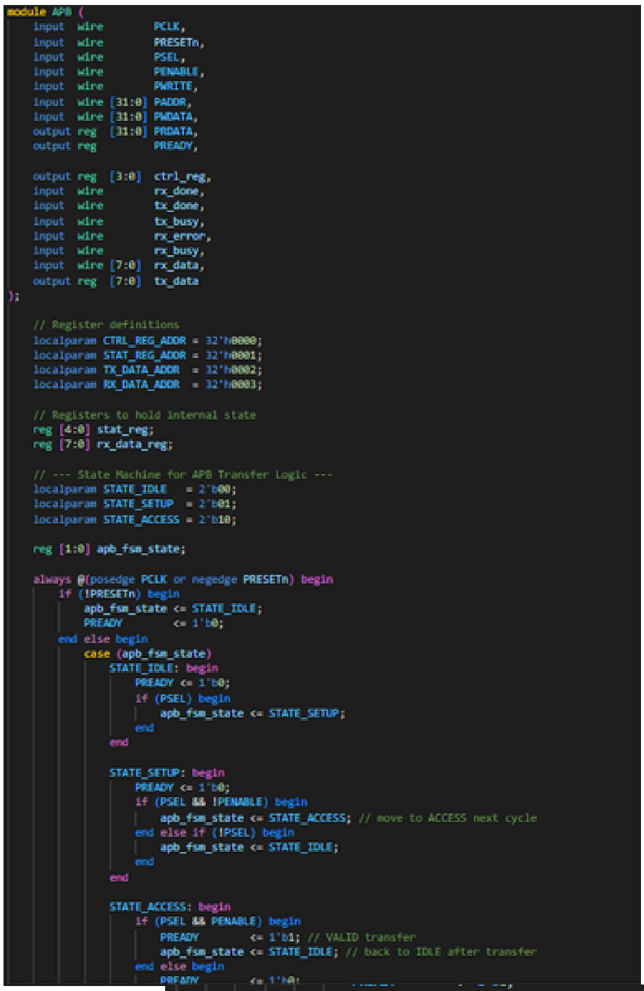
The receiver samples the line at 1.5 bit-time after the falling edge of the start bit.
FSM states: IDLE → START → DATA → STOP → DONE/ERR.
Bits are shifted into rx_shift_reg and latched into rx_data once reception is complete.
If stop bit = 0, a framing error is flagged via rx_error.

Baud Generator

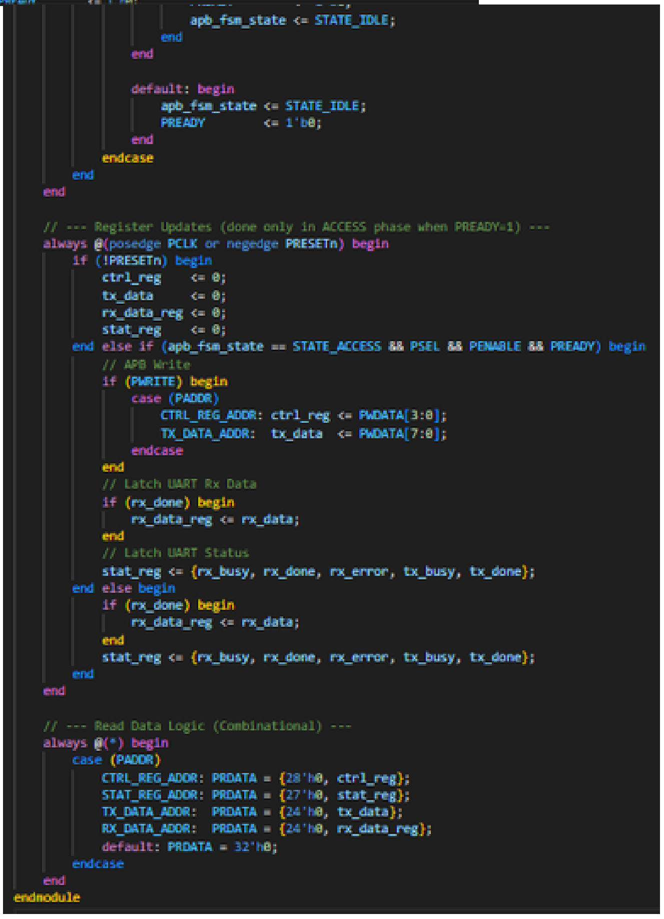

```
module baud_counter #(
    parameter CLK_FREQ = 100_000_000,
    parameter BAUD     = 9600,
    parameter DIV      = CLK_FREQ / BAUD
)(
    input  wire        clk,
    input  wire        rst,
    input  wire [20:0] load_val,
    output reg         tick_FSM
);

    reg [20:0] count;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            count    <= 0;
            tick_FSM <= 0;
        end else begin
            if (count == 0) begin
                count    <= load_val;
                tick_FSM <= 1;
            end else begin
                count    <= count - 1;
                tick_FSM <= 0;
            end
        end
    end

endmodule
```

Generates tick pulses at baud intervals:
Parameterized with CLK_FREQ = 100 MHz and BAUD = 9600.
Produces a tick_FSM pulse when the down-counter reaches zero.
Reloads with DIV = CLK_FREQ / BAUD.
This ensures both TX and RX operate with precise bit-timing.

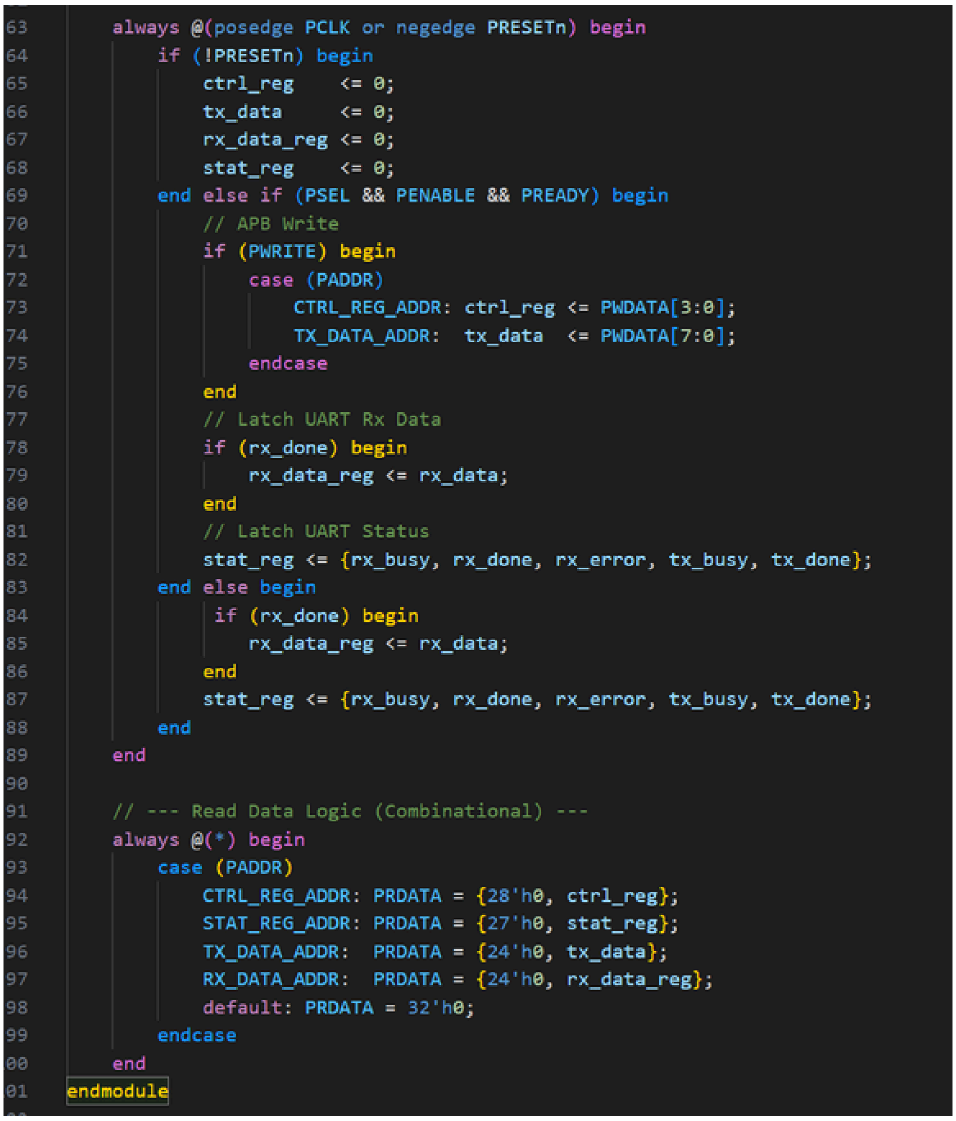
APB Slave Logic



```verilog
module APB (
    input  wire        PCLK,
    input  wire        PRESETn,
    input  wire        PSEL,
    input  wire        PENABLE,
    input  wire        PWRITE,
    input  wire [31:0] PADDR,
    input  wire [31:0] PWDATA,
    output reg  [31:0] PRDATA,
    output reg         PREADY,

    output reg  [3:0]  ctrl_reg,
    input  wire        rx_done,
    input  wire        tx_done,
    input  wire        tx_busy,
    input  wire        rx_error,
    input  wire        rx_busy,
    input  wire [7:0]  rx_data,
    output reg  [7:0]  tx_data
);

    // Register definitions
    localparam CTRL_REG_ADDR = 32'h0000;
    localparam STAT_REG_ADDR = 32'h0001;
    localparam TX_DATA_ADDR  = 32'h0002;
    localparam RX_DATA_ADDR  = 32'h0003;

    // Registers to hold internal state
    reg [4:0] stat_reg;
    reg [7:0] rx_data_reg;

    // --- State Machine for APB Transfer Logic ---
    localparam STATE_IDLE   = 2'b00;
    localparam STATE_SETUP  = 2'b01;
    localparam STATE_ACCESS = 2'b10;

    reg [1:0] apb_fsm_state;

    always @(posedge PCLK or negedge PRESETn) begin
        if (!PRESETn) begin
            apb_fsm_state <= STATE_IDLE;
            PREADY        <= 1'b0;
        end else begin
            case (apb_fsm_state)
                STATE_IDLE: begin
                    PREADY <= 1'b0;
                    if (PSEL) begin
                        apb_fsm_state <= STATE_SETUP;
                    end
                end

                STATE_SETUP: begin
                    PREADY <= 1'b0;
                    if (PSEL && !PENABLE) begin
                        apb_fsm_state <= STATE_ACCESS; // move to ACCESS next cycle
                    end else if (!PSEL) begin
                        apb_fsm_state <= STATE_IDLE;
                    end
                end

                STATE_ACCESS: begin
                    if (PSEL && PENABLE) begin
                        PREADY        <= 1'b1; // VALID transfer
                        apb_fsm_state <= STATE_IDLE; // back to IDLE after transfer
                    end else begin
                        PREADY        <= 1'b0;
```



```verilog
                        apb_fsm_state <= STATE_IDLE;
                    end
                end

                default: begin
                    apb_fsm_state <= STATE_IDLE;
                    PREADY        <= 1'b0;
                end
            endcase
        end
    end

    // --- Register Updates (done only in ACCESS phase when PREADY=1) ---
    always @(posedge PCLK or negedge PRESETn) begin
        if (!PRESETn) begin
            ctrl_reg    <= 0;
            tx_data     <= 0;
            rx_data_reg <= 0;
            stat_reg    <= 0;
        end else if (apb_fsm_state == STATE_ACCESS && PSEL && PENABLE && PREADY) begin
            // APB Write
            if (PWRITE) begin
                case (PADDR)
                    CTRL_REG_ADDR: ctrl_reg <= PWDATA[3:0];
                    TX_DATA_ADDR:  tx_data  <= PWDATA[7:0];
                endcase
            end
            // Latch UART Rx Data
            if (rx_done) begin
                rx_data_reg <= rx_data;
            end
            // Latch UART Status
            stat_reg <= {rx_busy, rx_done, rx_error, tx_busy, tx_done};
        end else begin
            if (rx_done) begin
                rx_data_reg <= rx_data;
            end
            stat_reg <= {rx_busy, rx_done, rx_error, tx_busy, tx_done};
        end
    end

    // --- Read Data Logic (Combinational) ---
    always @(*) begin
        case (PADDR)
            CTRL_REG_ADDR: PRDATA = {28'h0, ctrl_reg};
            STAT_REG_ADDR: PRDATA = {27'h0, stat_reg};
            TX_DATA_ADDR:  PRDATA = {24'h0, tx_data};
            RX_DATA_ADDR:  PRDATA = {24'h0, rx_data_reg};
            default: PRDATA = 32'h0;
        endcase
    end
endmodule
```

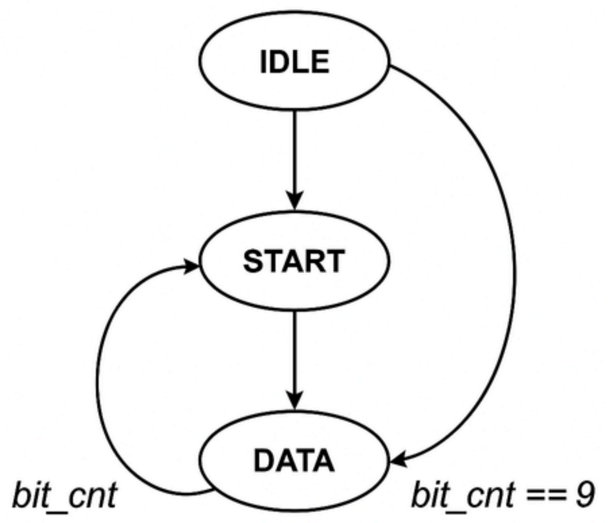


```verilog
    always @(posedge PCLK or negedge PRESETn) begin
        if (!PRESETn) begin
            ctrl_reg    <= 0;
            tx_data     <= 0;
            rx_data_reg <= 0;
            stat_reg    <= 0;
        end else if (PSEL && PENABLE && PREADY) begin
            // APB Write
            if (PWRITE) begin
                case (PADDR)
                    CTRL_REG_ADDR: ctrl_reg <= PWDATA[3:0];
                    TX_DATA_ADDR:  tx_data  <= PWDATA[7:0];
                endcase
            end
            // Latch UART Rx Data
            if (rx_done) begin
                rx_data_reg <= rx_data;
            end
            // Latch UART Status
            stat_reg <= {rx_busy, rx_done, rx_error, tx_busy, tx_done};
        end else begin
             if (rx_done) begin
                rx_data_reg <= rx_data;
            end
            stat_reg <= {rx_busy, rx_done, rx_error, tx_busy, tx_done};
        end
    end

    // --- Read Data Logic (Combinational) ---
    always @(*) begin
        case (PADDR)
            CTRL_REG_ADDR: PRDATA = {28'h0, ctrl_reg};
            STAT_REG_ADDR: PRDATA = {27'h0, stat_reg};
            TX_DATA_ADDR:  PRDATA = {24'h0, tx_data};
            RX_DATA_ADDR:  PRDATA = {24'h0, rx_data_reg};
            default: PRDATA = 32'h0;
        endcase
    end
endmodule
```

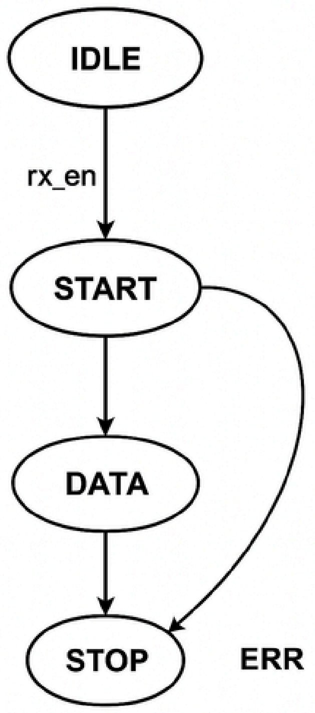
Implements an AMBA APB slave interface to expose UART registers:
CTRL_REG (0x0000) → control bits (tx_en, rx_en, tx_rst, rx_rst).
STAT_REG (0x0001) → status (rx_busy, rx_done, rx_error, tx_busy, tx_done).
TX_DATA (0x0002) → data to transmit.
RX_DATA (0x0003) → received data.
FSM for APB handshake:
IDLE: Wait for PSEL.
SETUP: Assert PREADY for 1 cycle when PENABLE is high.

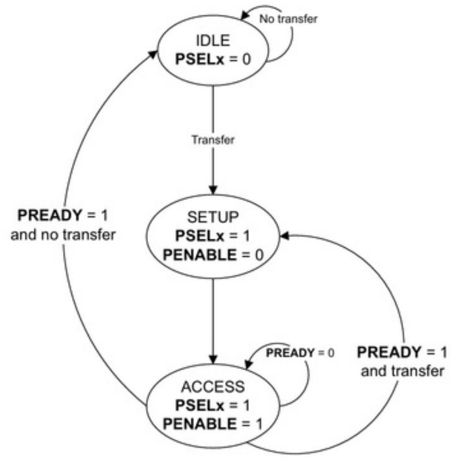
3.0 State Diagrams

UART Transmitter FSM



UART Receiver FSM





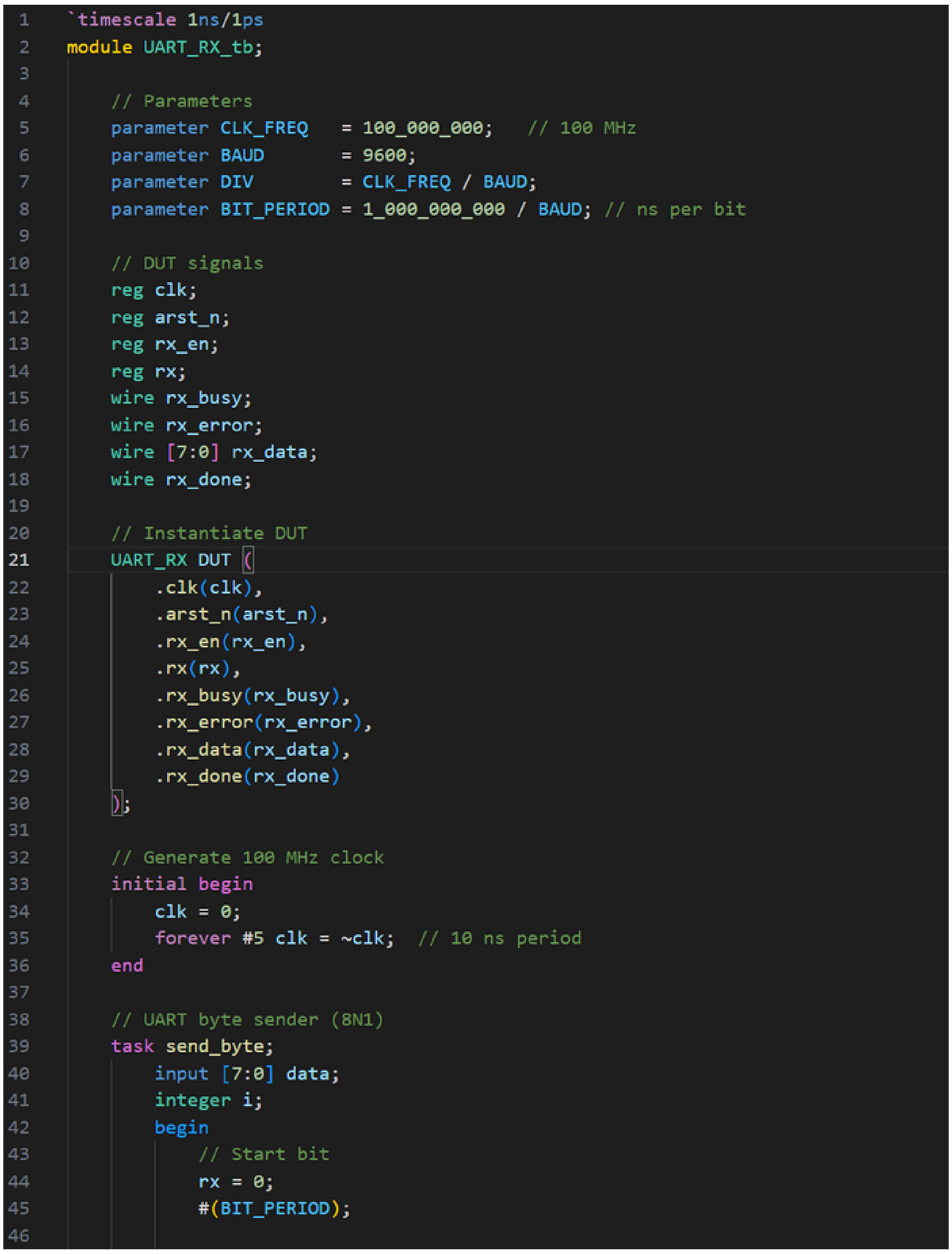
4.0 Design Decisions

8-N-1 format chosen (common and simple).
APB protocol used instead of AXI (simpler for register-mapped peripherals).
Synchronous logic with async reset for clean reset.
Status and data registers latched to avoid glitches on reads.

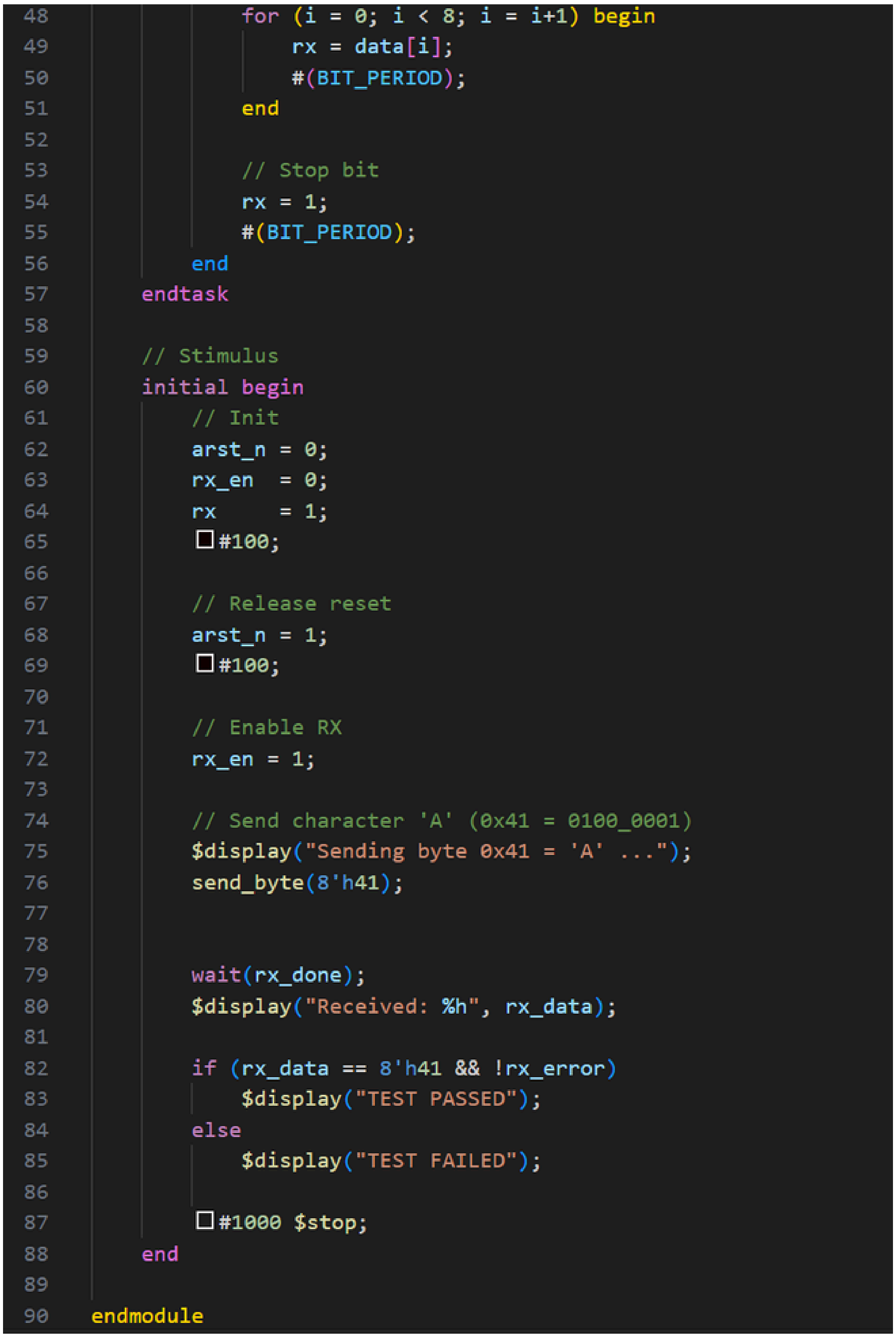
5.0 Verification Strategy

Test Environment: Custom APB testbench (APB_tb) simulates master transactions.
Loopback test: TX output connected to RX input (tx_serial → rx).
APB tasks:
apb_write(addr, data) – write to APB register.
apb_read(addr) – read from APB register, display result.
Test Sequence:
Reset system.
Enable UART by writing to CTRL_REG.
Send data byte (0x51) by writing to TX_DATA.
Wait for TX to complete.
Read RX_DATA → should return 0x51.
Read STAT_REG → should confirm tx_done=1, rx_done=1, rx_error=0.
The testbench provides self-checking capability via $display messages.

UART RX Testbench

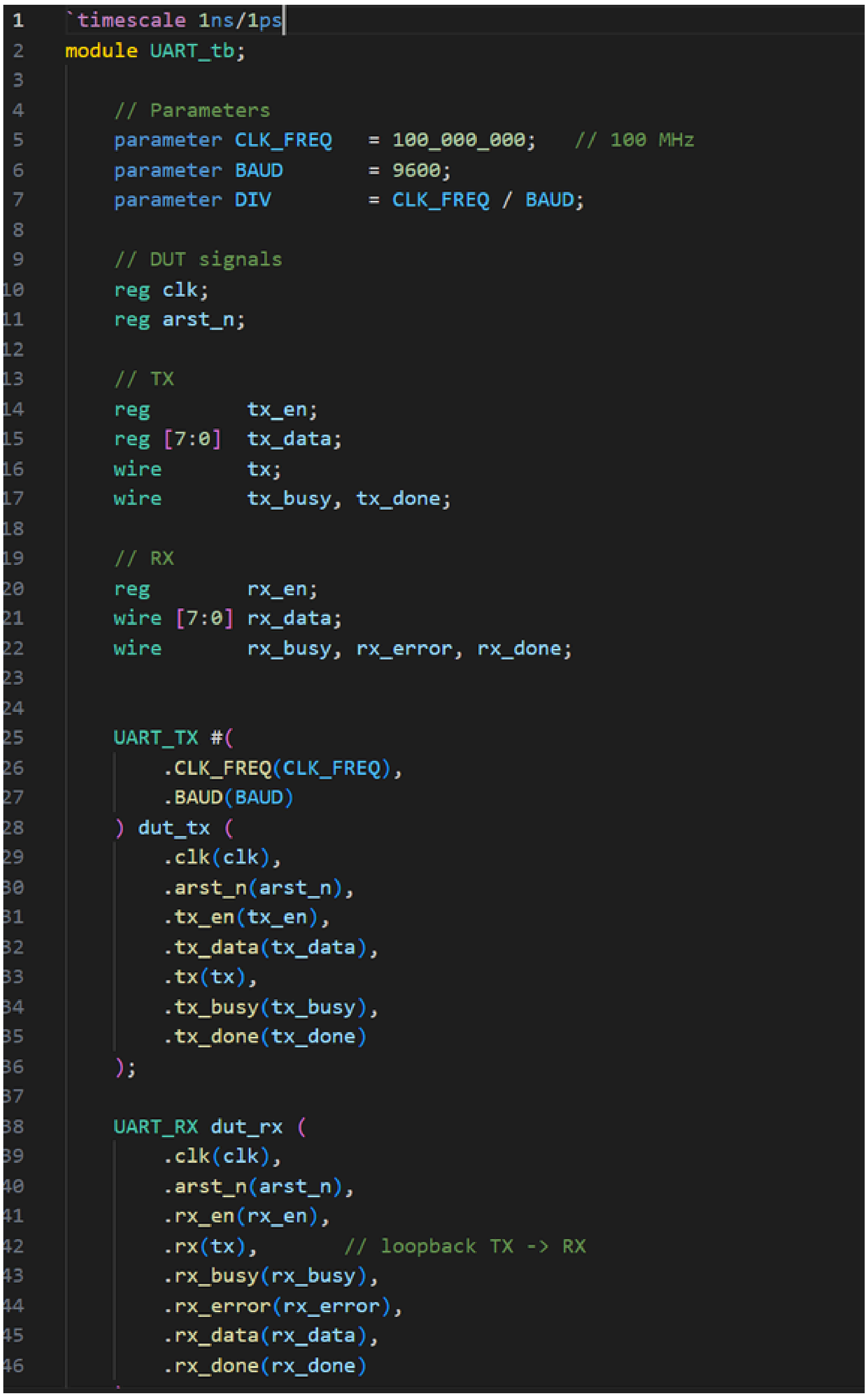


```
`timescale 1ns/1ps
module UART_RX_tb;

    // Parameters
    parameter CLK_FREQ    = 100_000_000;    // 100 MHz
    parameter BAUD        = 9600;
    parameter DIV         = CLK_FREQ / BAUD;
    parameter BIT_PERIOD = 1_000_000_000 / BAUD; // ns per bit

    // DUT signals
    reg clk;
    reg arst_n;
    reg rx_en;
    reg rx;
    wire rx_busy;
    wire rx_error;
    wire [7:0] rx_data;
    wire rx_done;

    // Instantiate DUT
    UART_RX DUT (
        .clk(clk),
        .arst_n(arst_n),
        .rx_en(rx_en),
        .rx(rx),
        .rx_busy(rx_busy),
        .rx_error(rx_error),
        .rx_data(rx_data),
        .rx_done(rx_done)
    );

    // Generate 100 MHz clock
    initial begin
        clk = 0;
        forever #5 clk = ~clk;  // 10 ns period
    end

    // UART byte sender (8N1)
    task send_byte;
        input [7:0] data;
        integer i;
        begin
            // Start bit
            rx = 0;
            #(BIT_PERIOD);

```

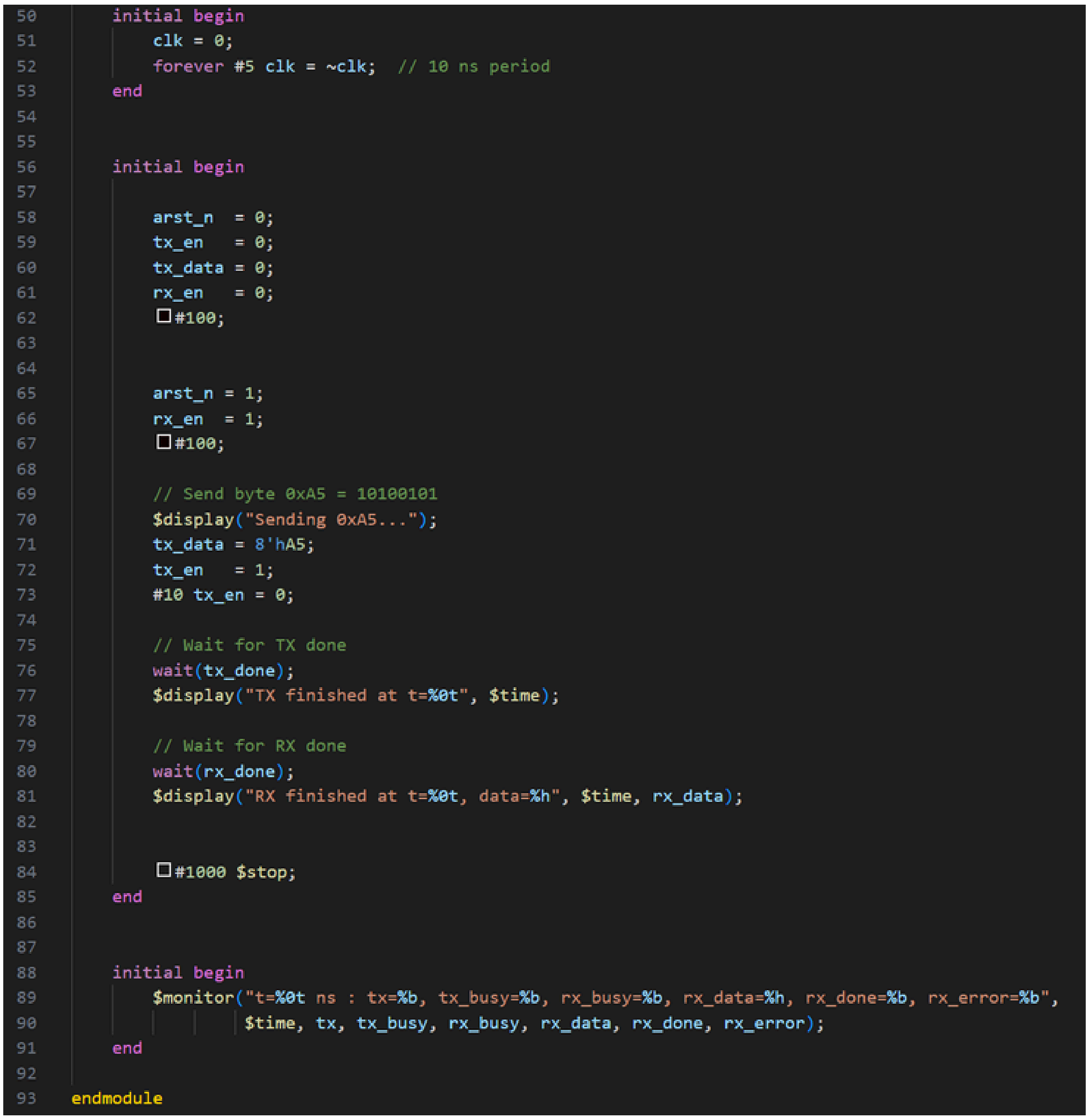


```
            for (i = 0; i < 8; i = i+1) begin
                rx = data[i];
                #(BIT_PERIOD);
            end

            // Stop bit
            rx = 1;
            #(BIT_PERIOD);
        end
    endtask

    // Stimulus
    initial begin
        // Init
        arst_n = 0;
        rx_en  = 0;
        rx     = 1;
        □#100;

        // Release reset
        arst_n = 1;
        □#100;

        // Enable RX
        rx_en = 1;

        // Send character 'A' (0x41 = 0100_0001)
        $display("Sending byte 0x41 = 'A' ...");
        send_byte(8'h41);


        wait(rx_done);
        $display("Received: %h", rx_data);

        if (rx_data == 8'h41 && !rx_error)
            $display("TEST PASSED");
        else
            $display("TEST FAILED");

        □#1000 $stop;
    end

endmodule
```

UART TX Testbench

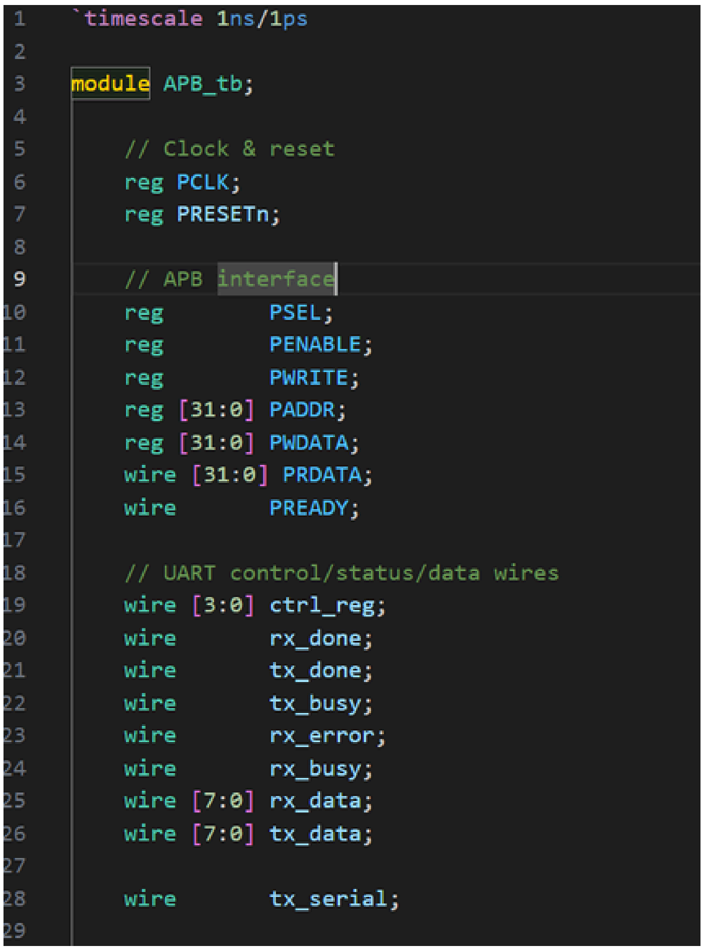


```
`timescale 1ns/1ps
module UART_tb;

    // Parameters
    parameter CLK_FREQ   = 100_000_000;    // 100 MHz
    parameter BAUD       = 9600;
    parameter DIV        = CLK_FREQ / BAUD;

    // DUT signals
    reg clk;
    reg arst_n;

    // TX
    reg         tx_en;
    reg [7:0]   tx_data;
    wire        tx;
    wire        tx_busy, tx_done;

    // RX
    reg         rx_en;
    wire [7:0]  rx_data;
    wire        rx_busy, rx_error, rx_done;


    UART_TX #(
        .CLK_FREQ(CLK_FREQ),
        .BAUD(BAUD)
    ) dut_tx (
        .clk(clk),
        .arst_n(arst_n),
        .tx_en(tx_en),
        .tx_data(tx_data),
        .tx(tx),
        .tx_busy(tx_busy),
        .tx_done(tx_done)
    );

    UART_RX dut_rx (
        .clk(clk),
        .arst_n(arst_n),
        .rx_en(rx_en),
        .rx(tx),       // loopback TX -> RX
        .rx_busy(rx_busy),
        .rx_error(rx_error),
        .rx_data(rx_data),
        .rx_done(rx_done)
```

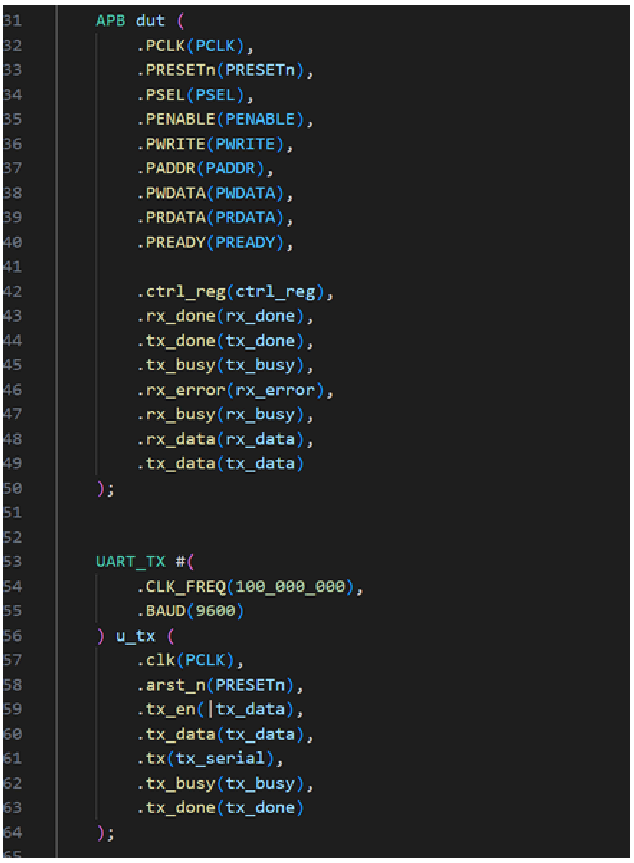


```verilog
    initial begin
        clk = 0;
        forever #5 clk = ~clk;  // 10 ns period
    end


    initial begin

        arst_n  = 0;
        tx_en   = 0;
        tx_data = 0;
        rx_en   = 0;
        □#100;


        arst_n = 1;
        rx_en  = 1;
        □#100;

        // Send byte 0xA5 = 10100101
        $display("Sending 0xA5...");
        tx_data = 8'hA5;
        tx_en   = 1;
        #10 tx_en = 0;

        // Wait for TX done
        wait(tx_done);
        $display("TX finished at t=%0t", $time);

        // Wait for RX done
        wait(rx_done);
        $display("RX finished at t=%0t, data=%h", $time, rx_data);


        □#1000 $stop;
    end


    initial begin
        $monitor("t=%0t ns : tx=%b, tx_busy=%b, rx_busy=%b, rx_data=%h, rx_done=%b, rx_error=%b",
                 $time, tx, tx_busy, rx_busy, rx_data, rx_done, rx_error);
    end

endmodule
```

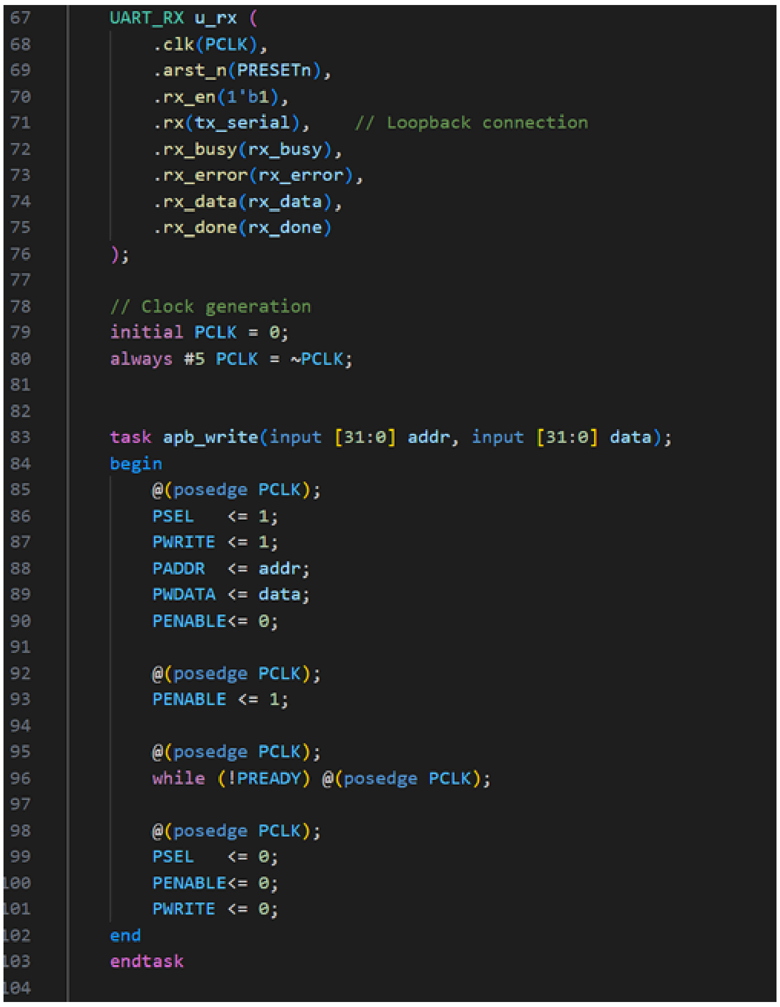
APB Testbench



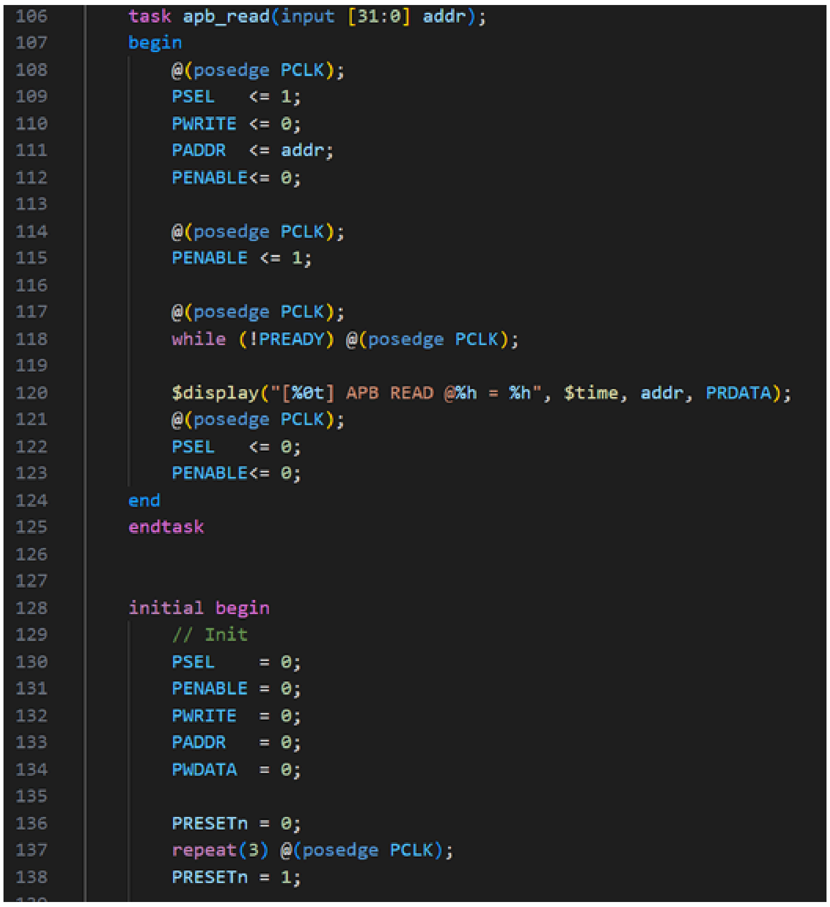
```
`timescale 1ns/1ps

module APB_tb;

    // Clock & reset
    reg PCLK;
    reg PRESETn;

    // APB interface
    reg         PSEL;
    reg         PENABLE;
    reg         PWRITE;
    reg [31:0]  PADDR;
    reg [31:0]  PWDATA;
    wire [31:0] PRDATA;
    wire        PREADY;

    // UART control/status/data wires
    wire [3:0]  ctrl_reg;
    wire        rx_done;
    wire        tx_done;
    wire        tx_busy;
    wire        rx_error;
    wire        rx_busy;
    wire [7:0]  rx_data;
    wire [7:0]  tx_data;

    wire        tx_serial;

```



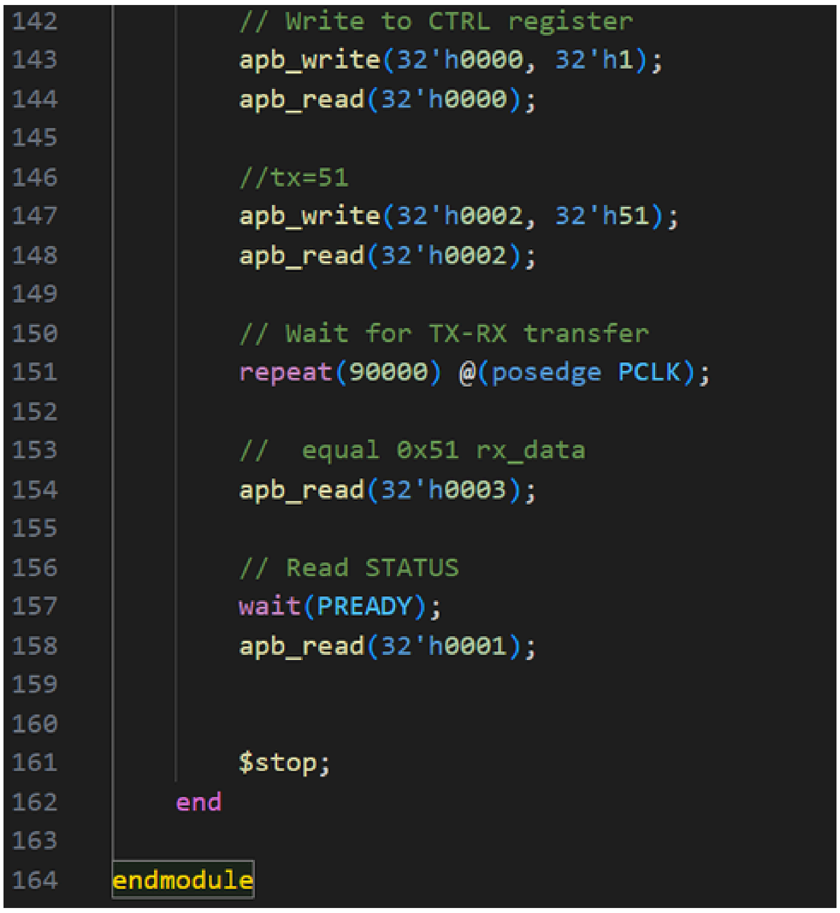
```
    APB dut (
        .PCLK(PCLK),
        .PRESETn(PRESETn),
        .PSEL(PSEL),
        .PENABLE(PENABLE),
        .PWRITE(PWRITE),
        .PADDR(PADDR),
        .PWDATA(PWDATA),
        .PRDATA(PRDATA),
        .PREADY(PREADY),

        .ctrl_reg(ctrl_reg),
        .rx_done(rx_done),
        .tx_done(tx_done),
        .tx_busy(tx_busy),
        .rx_error(rx_error),
        .rx_busy(rx_busy),
        .rx_data(rx_data),
        .tx_data(tx_data)
    );


    UART_TX #(
        .CLK_FREQ(100_000_000),
        .BAUD(9600)
    ) u_tx (
        .clk(PCLK),
        .arst_n(PRESETn),
        .tx_en(|tx_data),
        .tx_data(tx_data),
        .tx(tx_serial),
        .tx_busy(tx_busy),
        .tx_done(tx_done)
    );
```



```
UART_RX u_rx (
    .clk(PCLK),
    .arst_n(PRESETn),
    .rx_en(1'b1),
    .rx(tx_serial),     // Loopback connection
    .rx_busy(rx_busy),
    .rx_error(rx_error),
    .rx_data(rx_data),
    .rx_done(rx_done)
);

// Clock generation
initial PCLK = 0;
always #5 PCLK = ~PCLK;


task apb_write(input [31:0] addr, input [31:0] data);
begin
    @(posedge PCLK);
    PSEL    <= 1;
    PWRITE  <= 1;
    PADDR   <= addr;
    PWDATA  <= data;
    PENABLE<= 0;

    @(posedge PCLK);
    PENABLE <= 1;

    @(posedge PCLK);
    while (!PREADY) @(posedge PCLK);

    @(posedge PCLK);
    PSEL    <= 0;
    PENABLE<= 0;
    PWRITE  <= 0;
end
endtask

```

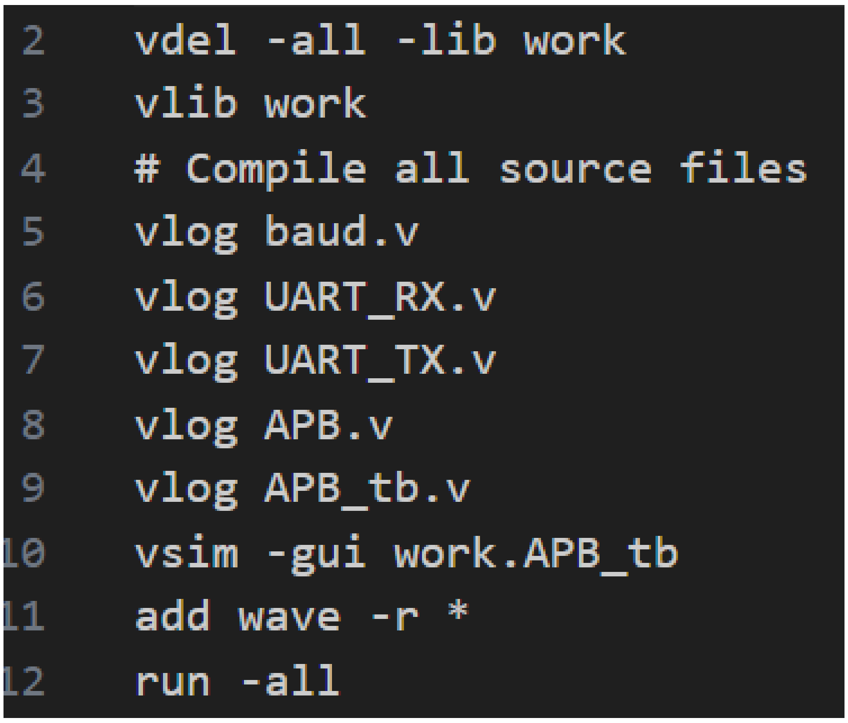


```
task apb_read(input [31:0] addr);
begin
    @(posedge PCLK);
    PSEL    <= 1;
    PWRITE  <= 0;
    PADDR   <= addr;
    PENABLE<= 0;

    @(posedge PCLK);
    PENABLE <= 1;

    @(posedge PCLK);
    while (!PREADY) @(posedge PCLK);

    $display("[%0t] APB READ @%h = %h", $time, addr, PRDATA);
    @(posedge PCLK);
    PSEL    <= 0;
    PENABLE<= 0;
end
endtask


initial begin
    // Init
    PSEL     = 0;
    PENABLE = 0;
    PWRITE  = 0;
    PADDR   = 0;
    PWDATA  = 0;

    PRESETn = 0;
    repeat(3) @(posedge PCLK);
    PRESETn = 1;
```



```
            // Write to CTRL register
            apb_write(32'h0000, 32'h1);
            apb_read(32'h0000);

            //tx=51
            apb_write(32'h0002, 32'h51);
            apb_read(32'h0002);

            // Wait for TX-RX transfer
            repeat(90000) @(posedge PCLK);

            //  equal 0x51 rx_data
            apb_read(32'h0003);

            // Read STATUS
            wait(PREADY);
            apb_read(32'h0001);


            $stop;
        end

endmodule
```

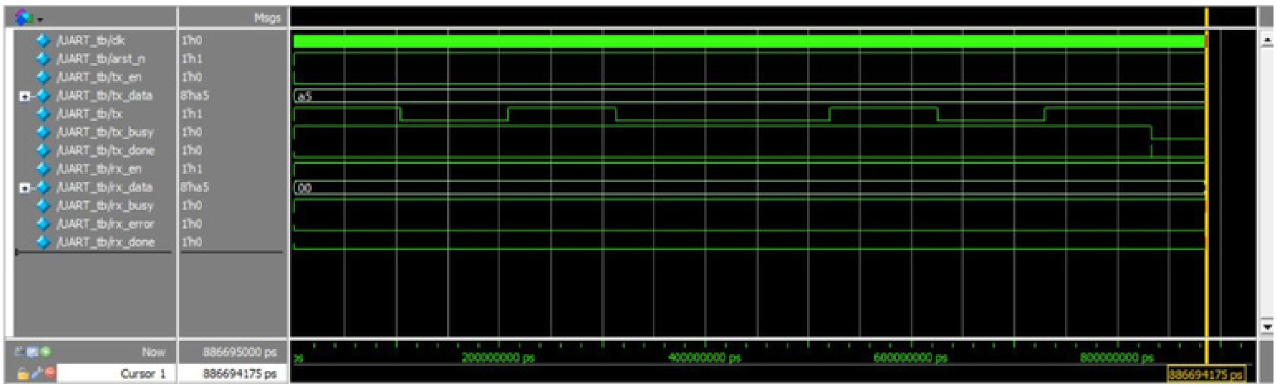
DO file

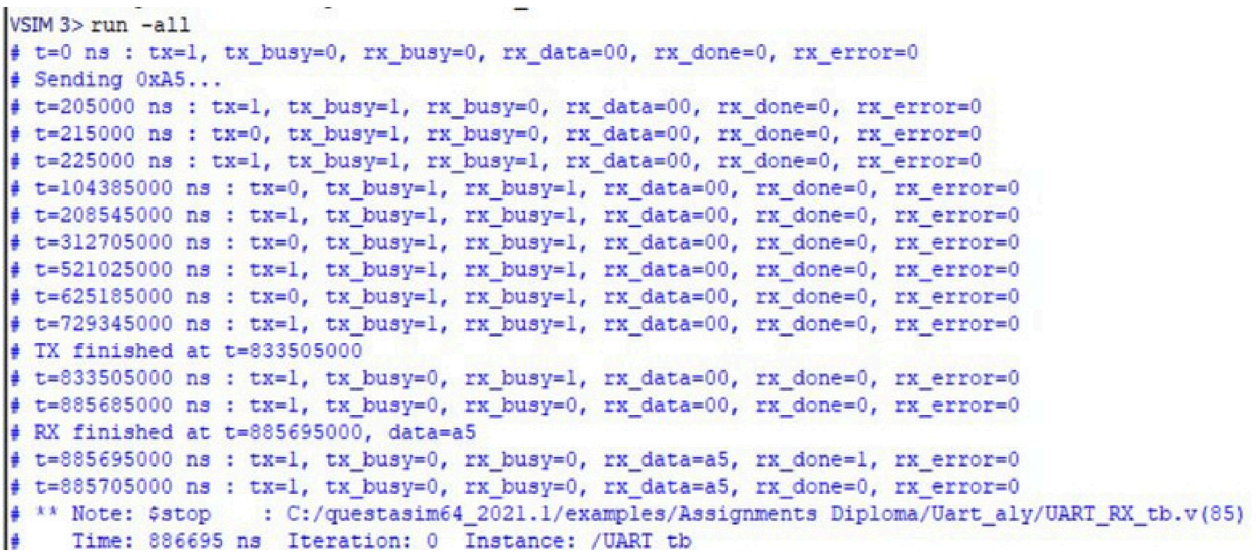


```
vdel -all -lib work
vlib work
# Compile all source files
vlog baud.v
vlog UART_RX.v
vlog UART_TX.v
vlog APB.v
vlog APB_tb.v
vsim -gui work.APB_tb
add wave -r *
run -all
```

6.0 Simulation Results

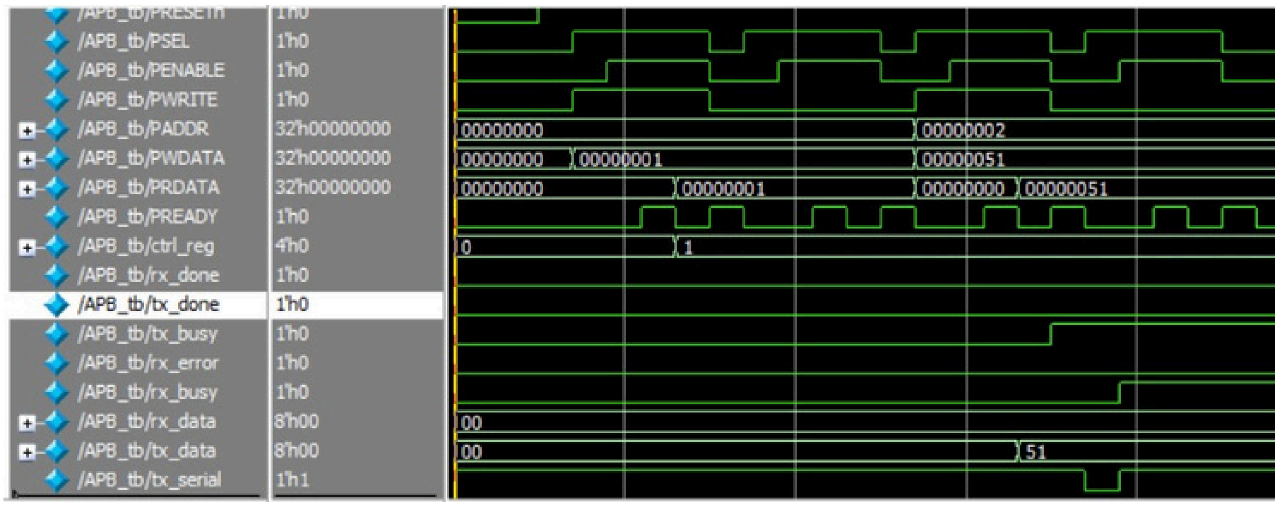
TX & RX



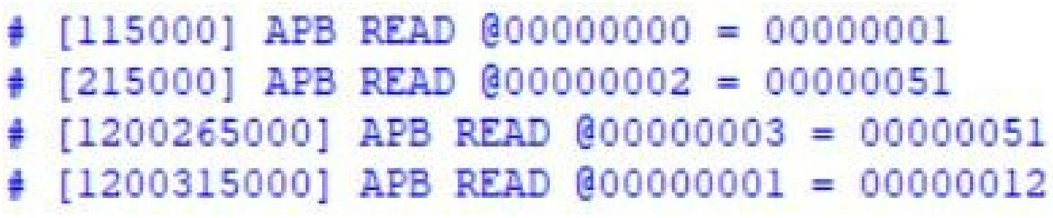


```
VSIM 3> run -all
# t=0 ns : tx=1, tx_busy=0, rx_busy=0, rx_data=00, rx_done=0, rx_error=0
# Sending 0xA5...
# t=205000 ns : tx=1, tx_busy=1, rx_busy=0, rx_data=00, rx_done=0, rx_error=0
# t=215000 ns : tx=0, tx_busy=1, rx_busy=0, rx_data=00, rx_done=0, rx_error=0
# t=225000 ns : tx=1, tx_busy=1, rx_busy=1, rx_data=00, rx_done=0, rx_error=0
# t=104385000 ns : tx=0, tx_busy=1, rx_busy=1, rx_data=00, rx_done=0, rx_error=0
# t=208545000 ns : tx=1, tx_busy=1, rx_busy=1, rx_data=00, rx_done=0, rx_error=0
# t=312705000 ns : tx=0, tx_busy=1, rx_busy=1, rx_data=00, rx_done=0, rx_error=0
# t=521025000 ns : tx=1, tx_busy=1, rx_busy=1, rx_data=00, rx_done=0, rx_error=0
# t=625185000 ns : tx=0, tx_busy=1, rx_busy=1, rx_data=00, rx_done=0, rx_error=0
# t=729345000 ns : tx=1, tx_busy=1, rx_busy=1, rx_data=00, rx_done=0, rx_error=0
# TX finished at t=833505000
# t=833505000 ns : tx=1, tx_busy=0, rx_busy=1, rx_data=00, rx_done=0, rx_error=0
# t=885685000 ns : tx=1, tx_busy=0, rx_busy=0, rx_data=00, rx_done=0, rx_error=0
# RX finished at t=885695000, data=a5
# t=885695000 ns : tx=1, tx_busy=0, rx_busy=0, rx_data=a5, rx_done=1, rx_error=0
# t=885705000 ns : tx=1, tx_busy=0, rx_busy=0, rx_data=a5, rx_done=0, rx_error=0
# ** Note: $stop    : C:/questasim64_2021.1/examples/Assignments Diploma/Uart_aly/UART_RX_tb.v(85)
#    Time: 886695 ns  Iteration: 0  Instance: /UART tb
```

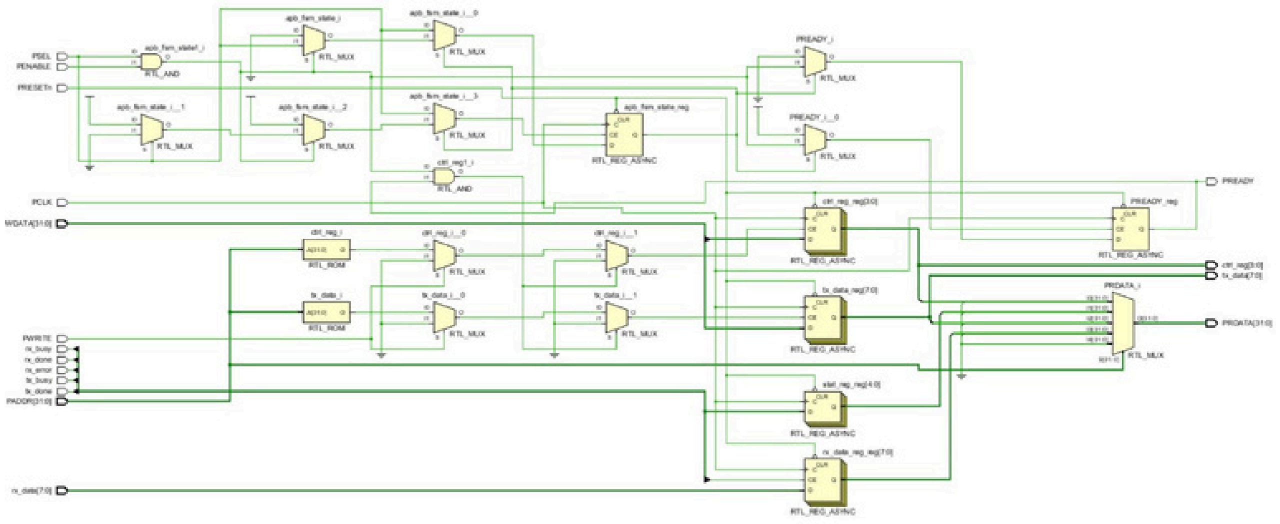
APB output

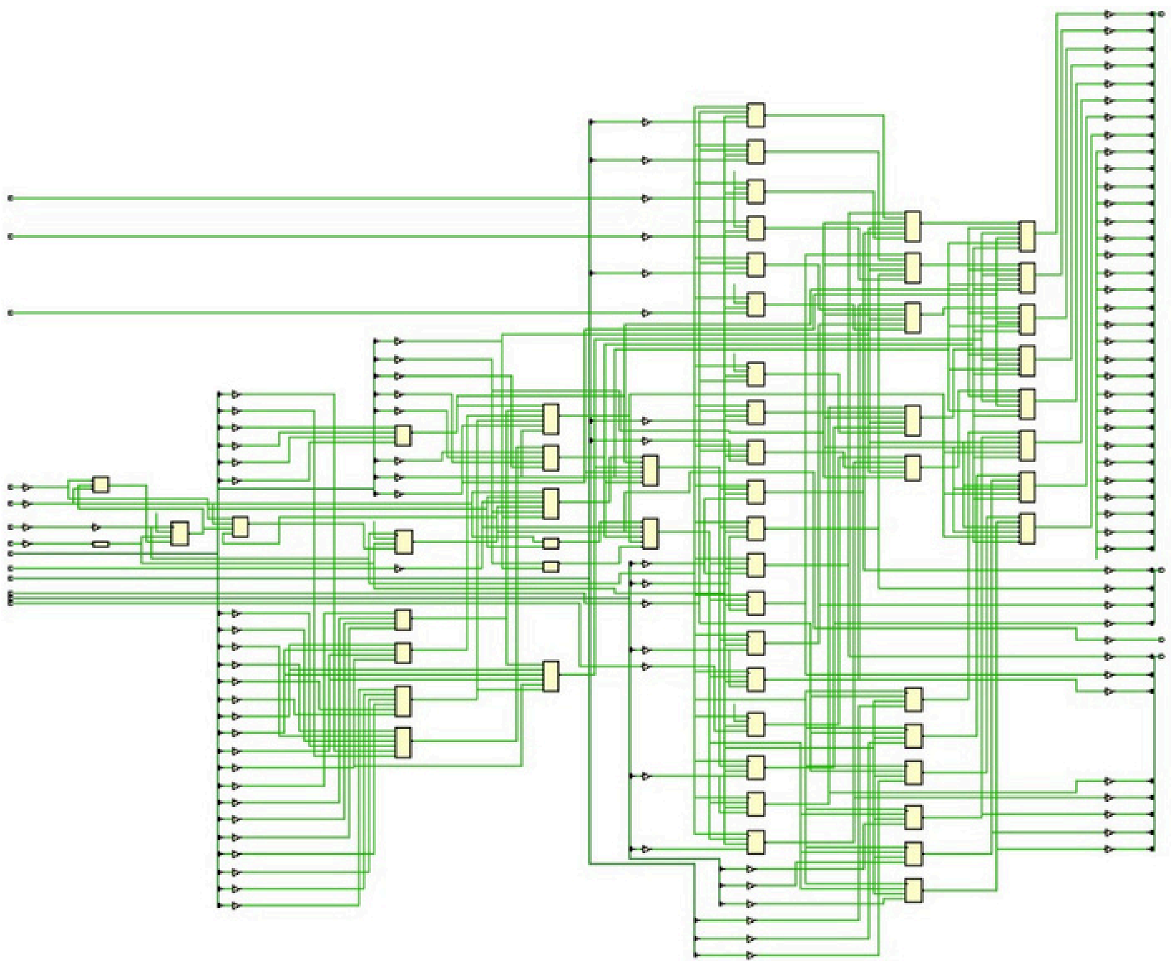


```
/APB_tb/PRESETn   1'h0
/APB_tb/PSEL      1'h0
/APB_tb/PENABLE   1'h0
/APB_tb/PWRITE    1'h0
/APB_tb/PADDR     32'h00000000
/APB_tb/PWDATA    32'h00000000
/APB_tb/PRDATA    32'h00000000
/APB_tb/PREADY    1'h0
/APB_tb/ctrl_reg  4'h0
/APB_tb/rx_done   1'h0
/APB_tb/tx_done   1'h0
/APB_tb/tx_busy   1'h0
/APB_tb/rx_error  1'h0
/APB_tb/rx_busy   1'h0
/APB_tb/rx_data   8'h00
/APB_tb/tx_data   8'h00
/APB_tb/tx_serial 1'h1
```



```
# [115000] APB READ @00000000 = 00000001
# [215000] APB READ @00000002 = 00000051
# [1200265000] APB READ @00000003 = 00000051
# [1200315000] APB READ @00000001 = 00000012
```


PSEL
PENABLE
PRESETn
PCLK
WDATA[31:0]
PWRITE
rx_busy
rx_done
rx_error
tx_busy
tx_done
PADDR[31:0]
rx_data[7:0]
apb_fsm_state1_i
RTL_AND
apb_fsm_state_i
apb_fsm_state_i__0
apb_fsm_state_i__1
apb_fsm_state_i__2
apb_fsm_state_i__3
RTL_MUX
ctrl_reg1_i
apb_fsm_state_reg
RTL_REG_ASYNC
ctrl_reg_i
tx_data_i
RTL_ROM
ctrl_reg_i__0
ctrl_reg_i__1
tx_data_i__0
tx_data_i__1
PREADY_i
PREADY_i__0
ctrl_reg_reg[3:0]
tx_data_reg[7:0]
stat_reg_reg[4:0]
rx_data_reg_reg[7:0]
PREADY_reg
PRDATA_i
PREADY
ctrl_reg[3:0]
tx_data[7:0]
PRDATA[31:0]



7.0 Conclusion

Successfully implemented UART TX, RX, baud generator, and APB wrapper.
Verified through modular testbenches.
Design can be extended with parity support and FIFO buffers for higher throughput.