KEITHSTON AND PARTNERS

PROJECT
APB UART

Prepared by :
mohamed khames

Prepared for :
Eng. mohamed tareq

Design code for TX :-

```
module uart_tx (
    input  wire        clk,
    input  wire        rst,        // synchronous active-high
    input  wire [31:0] baud_div,
    input  wire        tx_start,   // 1-cycle pulse
    input  wire [7:0]  tx_data,
    output reg         tx,
    output reg         tx_busy,
    output reg         tx_done
);

    // states
    localparam S_IDLE  = 3'd0;
    localparam S_START = 3'd1;
    localparam S_DATA  = 3'd2;
    localparam S_STOP  = 3'd3;
    localparam S_DONE  = 3'd4;

    reg [2:0] state, nxt_state;
    reg [31:0] baud_cnt;
    reg [2:0] bit_cnt;
    reg [7:0] shift_reg;

    // sequential
    always @(posedge clk) begin
        if (rst) begin
            state <= S_IDLE;
            baud_cnt <= 32'd0;
            bit_cnt <= 3'd0;
            shift_reg <= 8'd0;
            tx <= 1'b1;
            tx_busy <= 1'b0;
            tx_done <= 1'b0;
        end else begin
            state <= nxt_state;

            // default tx_done low; pulse one cycle when entering DONE
            if (state == S_DONE)
                tx_done <= 1'b1;
            else
                tx_done <= 1'b0;

            case (state)
                S_IDLE: begin
                    tx <= 1'b1;
                    tx_busy <= 1'b0;
                    if (tx_start) begin
                        shift_reg <= tx_data;
                        // load counter for one bit: use baud_div-1 so it counts down to 0 inclusive
                        baud_cnt <= (baud_div == 0) ? 32'd1 : baud_div - 1;
                        bit_cnt <= 3'd0;
                        tx_busy <= 1'b1;
                    end
                end

                S_START: begin
                    tx <= 1'b0; // start bit
                    tx_busy <= 1'b1;
                    if (baud_cnt == 0) begin
                        // finished start bit; reload for data bit
                        baud_cnt <= (baud_div == 0) ? 32'd1 : baud_div - 1;
                    end else begin
                        baud_cnt <= baud_cnt - 1;
                    end
                end

                S_DATA: begin
                    tx <= shift_reg[0];
                    tx_busy <= 1'b1;
                    if (baud_cnt == 0) begin
                        // shift after bit time
                        shift_reg <= {1'b0, shift_reg[7:1]};
                        bit_cnt <= bit_cnt + 1;
                        baud_cnt <= (baud_div == 0) ? 32'd1 : baud_div - 1;
                    end else begin
                        baud_cnt <= baud_cnt - 1;
                    end
                end

                S_STOP: begin
                    tx <= 1'b1; // stop bit
                    tx_busy <= 1'b1;
                    if (baud_cnt == 0) begin
                        // finished
                    end else begin
                        baud_cnt <= baud_cnt - 1;
                    end
                end

                S_DONE: begin
```

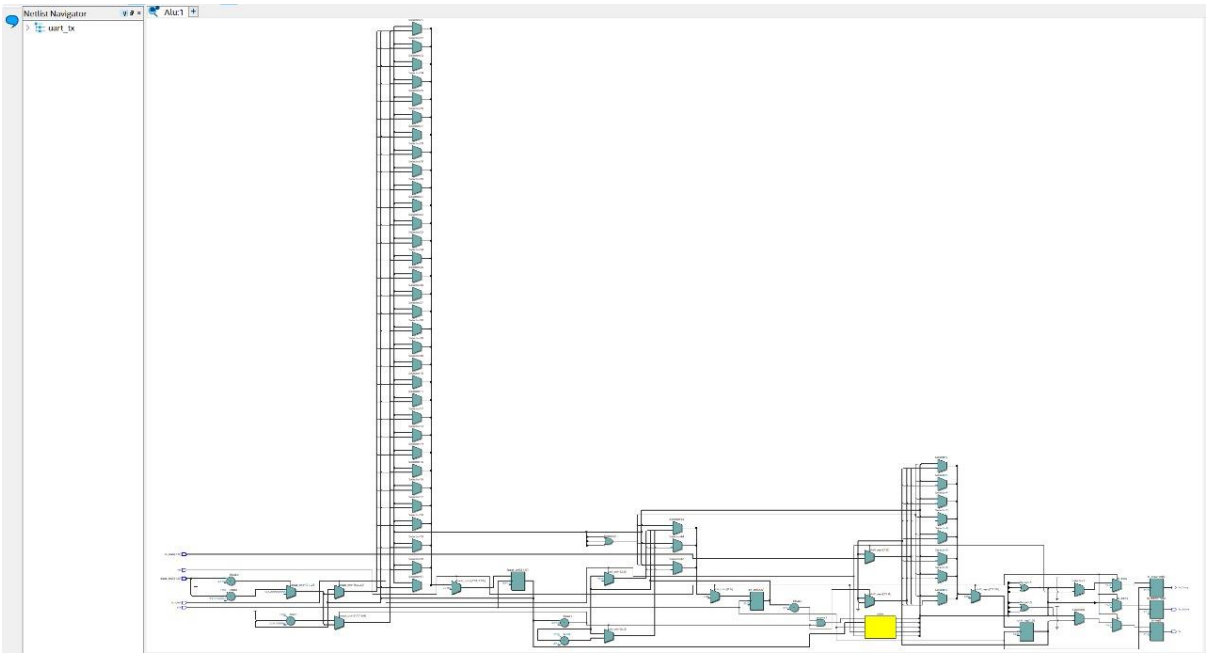
```
                    tx <= 1'b1;
                    tx_busy <= 1'b0;
                end

                default: begin
                    tx <= 1'b1;
                    tx_busy <= 1'b0;
                end
            endcase
        end
    end

    // combinational next-state
    always @(*) begin
        nxt_state = state;
        case (state)
            S_IDLE: if (tx_start) nxt_state = S_START;
            S_START: if (baud_cnt == 32'd0) nxt_state = S_DATA;
            S_DATA: if ((baud_cnt == 32'd0) && (bit_cnt == 3'd7)) nxt_state = S_STOP;
            S_STOP: if (baud_cnt == 32'd0) nxt_state = S_DONE;
            S_DONE: nxt_state = S_IDLE;
        endcase
    end

endmodule

```

Test bunch for TX :-

```
module tb_uart_tx;
    reg clk;
    reg rst;
    reg [31:0] baud_div;
    reg tx_start;
    reg [7:0] tx_data;
    wire tx;
    wire tx_busy;
    wire tx_done;

    // Clock 100MHz = 10ns period
    initial clk = 0;
    always #5 clk = ~clk;

    // DUT
    uart_tx uut (
        .clk(clk),
        .rst(rst),
        .baud_div(baud_div),
        .tx_start(tx_start),
        .tx_data(tx_data),
        .tx(tx),
        .tx_busy(tx_busy),
        .tx_done(tx_done)
    );

    initial begin
        $dumpfile("tb_uart_tx.vcd");
        $dumpvars(0, tb_uart_tx);

        // init
        rst = 1;
        baud_div = 32'd4;
        tx_start = 0;
        tx_data  = 8'h00;

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

```
41
42          // send 0x55
43          tx_data = 8'h55;
44          @(posedge clk);
45          tx_start = 1;        // pulse
46          @(posedge clk);
47          tx_start = 0;
48
49          wait (tx_done);
50          $display("Finished sending 0x55 at time %0t", $time);
51
52          // delay then send 0xA3
53          #50;
54          tx_data = 8'hA3;
55          @(posedge clk);
56          tx_start = 1;
57          @(posedge clk);
58          tx_start = 0;
59
60          wait (tx_done);
61          $display("Finished sending 0xA3 at time %0t", $time);
62
63          #200;
64          $finish;
65      end
66  endmodule
```

RTL RESULTS FOR TX :-



DESGIN CODE FOR RX :-

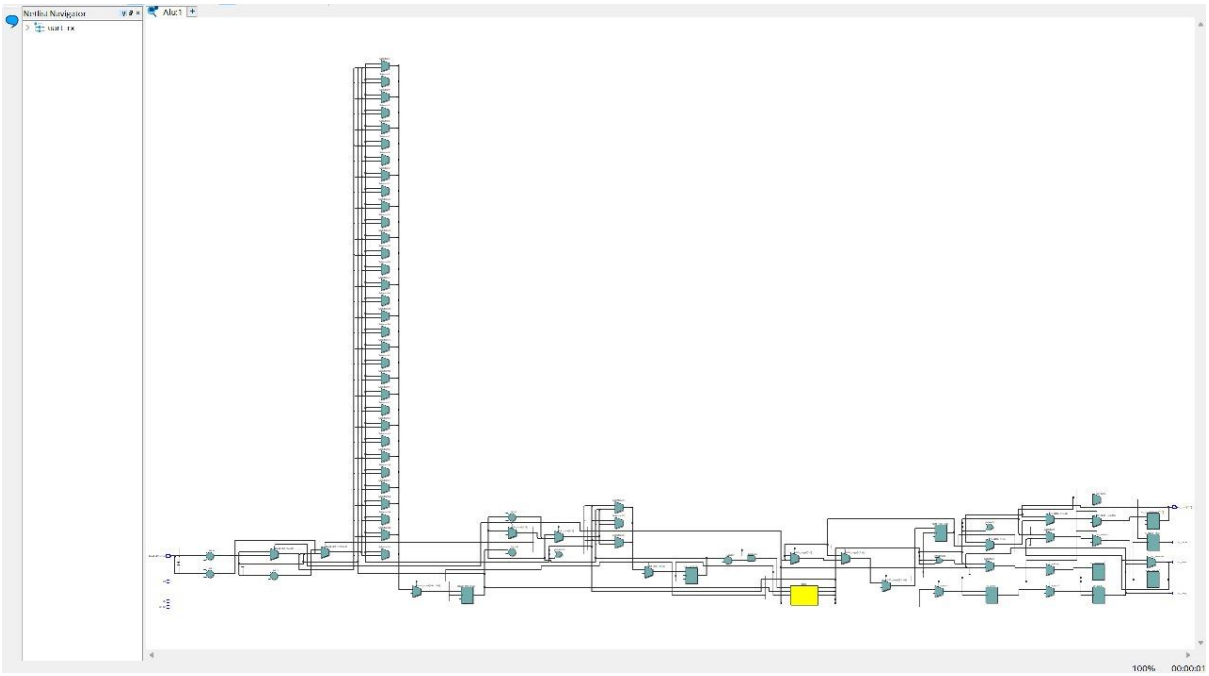
```
module uart_rx (
    input  wire        clk,
    input  wire        rst,
    input  wire [31:0] baud_div,
    input  wire        rx_pin,
    output reg [7:0]   rx_data,
    output reg         rx_done,
    output reg         rx_busy,
    output reg         rx_error
);

    // states
    localparam R_IDLE   = 3'd0;
    localparam R_START  = 3'd1;
    localparam R_DATA   = 3'd2;
    localparam R_STOP   = 3'd3;
    localparam R_DONE   = 3'd4;

    reg [2:0] state, nxt_state;
    reg [31:0] baud_cnt;
    reg [2:0] bit_cnt;
    reg [7:0] shift_reg;

    // synchronizer for rx_pin (2-flop)
    reg rx_sync1, rx_sync2;
    always @(posedge clk) begin
        if (rst) begin
            rx_sync1 <= 1'b1;
            rx_sync2 <= 1'b1;
        end else begin
            rx_sync1 <= rx_pin;
            rx_sync2 <= rx_sync1;
        end
    end
    wire rx_synced = rx_sync2;

    // sequential FSM
    always @(posedge clk) begin
        if (rst) begin
          state <= R_IDLE;
          baud_cnt <= 32'd0;
          bit_cnt <= 3'd0;
          shift_reg <= 8'd0;
          rx_data <= 8'd0;
          rx_done <= 1'b0;
          rx_busy <= 1'b0;
          rx_error <= 1'b0;
      end else begin
          state <= nxt_state;

          // Default outputs
          rx_done <= 1'b0;

          case (state)
              R_IDLE: begin
                  rx_busy <= 1'b0;
                  rx_error <= 1'b0;
                  if (rx_synced == 1'b0) begin // Start bit detected
                      baud_cnt <= (baud_div >> 1) - 1; // Sample at middle of start bit
                      rx_busy <= 1'b1;
                  end
              end

              R_START: begin
                  if (baud_cnt == 32'd0) begin
                      // Verify start bit is still low
                      if (rx_synced == 1'b0) begin
                          baud_cnt <= baud_div - 1; // Full bit period for data bits
                          bit_cnt <= 3'd0;
                      end
                  end else begin
                      baud_cnt <= baud_cnt - 1;
                  end
              end

              R_DATA: begin
```

```
                    if (baud_cnt == 32'd0) begin
                        // Shift in LSB first
                        shift_reg <= {rx_synced, shift_reg[7:1]};
                        bit_cnt <= bit_cnt + 1;
                        baud_cnt <= baud_div - 1;
                    end else begin
                        baud_cnt <= baud_cnt - 1;
                    end
                end

                R_STOP: begin
                    if (baud_cnt == 32'd0) begin
                        // Check stop bit
                        if (rx_synced == 1'b1) begin
                            rx_data <= shift_reg;
                            rx_error <= 1'b0;
                        end else begin
                            rx_data <= shift_reg;
                            rx_error <= 1'b1; // Framing error
                        end
                    end else begin
                        baud_cnt <= baud_cnt - 1;
                    end
                end

                R_DONE: begin
                    rx_done <= 1'b1;
                    rx_busy <= 1'b0;
                end
            endcase
        end
    end

    // next-state combinational logic
    always @(*) begin
        nxt_state = state;
        case (state)
            R_IDLE: begin
                if (rx_synced == 1'b0) // Start bit detected
                    nxt_state = R_START;
            end

            R_START: begin
                if (baud_cnt == 32'd0) begin
                    if (rx_synced == 1'b0) // Valid start bit
                        nxt_state = R_DATA;
                    else // False start
                        nxt_state = R_IDLE;
                end
            end

            R_DATA: begin
                if ((baud_cnt == 32'd0) && (bit_cnt == 3'd7))
                    nxt_state = R_STOP;
            end

            R_STOP: begin
                if (baud_cnt == 32'd0)
                    nxt_state = R_DONE;
            end

            R_DONE: begin
                nxt_state = R_IDLE;
            end

            default: nxt_state = R_IDLE;
        endcase
    end

endmodule
```

TEST BUNCH FOR RX :-

```
module tb_uart_rx;
    reg clk = 0;
    always #5 clk = ~clk; // 100MHz

    reg rst;
    reg [31:0] baud_div;
    reg rx_pin;
    wire [7:0] rx_data;
    wire rx_done;
    wire rx_busy;
    wire rx_error;

    uart_rx uut (
        .clk(clk),
        .rst(rst),
        .baud_div(baud_div),
        .rx_pin(rx_pin),
        .rx_data(rx_data),
        .rx_done(rx_done),
        .rx_busy(rx_busy),
        .rx_error(rx_error)
    );

    // helper task: send serial byte LSB-first with start/stop
    task send_byte;
        input [7:0] b;
        integer i;
        reg [31:0] bit_ticks;
        begin
            bit_ticks = baud_div;
            // start bit
            rx_pin = 0;
            repeat(bit_ticks) @(posedge clk);
            // data bits LSB first
            for (i=0;i<8;i=i+1) begin
                rx_pin = b[i];
                repeat(bit_ticks) @(posedge clk);
            end
            // stop bit
            rx_pin = 1;
            repeat(bit_ticks) @(posedge clk);
        end
    endtask

    initial begin
        $dumpfile("tb_uart_rx.vcd");
        $dumpvars(0,tb_uart_rx);
        rst = 1; rx_pin = 1; baud_div = 32'd868;
        #20;
        rst = 0;
        #20;

        // send 0xA5
        send_byte(8'hA5);

        // wait for rx_done
        wait(rx_done);
        $display("RX got %02h at time %0t (err=%0b)", rx_data, $time, rx_error);

        #200;
        $finish;
    end
endmodule

```

RTL FOR RX :-



DESIGN CODE FOR APB :-

```
module apb_uart (
    input  wire        pclk,
    input  wire        presetn,   // active-low reset (APB style)
    input  wire [31:0] paddr,
    input  wire        psel,
    input  wire        penable,
    input  wire        pwrite,
    input  wire [31:0] pwdata,
    output reg  [31:0] prdata,
    output reg         pready,
    // to UART modules
    output wire [31:0] baud_div,
    output reg         tx_start,    // single-cycle pulse
    output reg  [7:0]  tx_data,
    input  wire        tx_busy,
    input  wire        tx_done,
    output reg         rx_reset,    // optional resets to rx/t x
    input  wire [7:0]  rx_data,
    input  wire        rx_done,
    input  wire        rx_busy,
    input  wire        rx_error
);

    // internal reset active-high for our logic
    wire rst = ~presetn;

    // registers
    reg [31:0] ctrl_reg;
    reg [31:0] status_reg;
    reg [31:0] tx_data_reg;
    reg [31:0] rx_data_reg;
    reg [31:0] bauddiv_reg;

    // address mapping (word-aligned addresses)
    localparam ADDR_CTRL  = 32'h0000;
    localparam ADDR_STATUS= 32'h0001;
    localparam ADDR_TX    = 32'h0002;
    localparam ADDR_RX    = 32'h0003;
    localparam ADDR_BAUD  = 32'h0004;
```

```
    always @(posedge pclk) begin
        if (rst) begin
            pready <= 1'b0;
            prdata <= 32'd0;
            ctrl_reg <= 32'd0;
            status_reg <= 32'd0;
            tx_data_reg <= 32'd0;
            rx_data_reg <= 32'd0;
            bauddiv_reg <= 32'd868; // default 115200 @100MHz approx
            tx_start <= 1'b0;
            tx_data <= 8'd0;
            rx_reset <= 1'b0;
        end else begin
            // default outputs
            pready <= 1'b0;
            tx_start <= 1'b0;
            rx_reset <= 1'b0;

            // update status_reg from UART live signals (bits placed accordingly)
            status_reg[0] <= rx_busy;
            status_reg[1] <= tx_busy;
            status_reg[2] <= rx_done;
            status_reg[3] <= tx_done;
            status_reg[4] <= rx_error;

            if (psel & penable) begin
                pready <= 1'b1; // we complete transaction in this cycle
                if (pwrite) begin
                    // write transaction
                    case (paddr)
                        ADDR_CTRL: begin
                            ctrl_reg <= pwdata;
                            // if tx_en bit set => generate tx_start pulse & load tx_data_reg to tx_data
                            if (pwdata[0]) begin
                                tx_data <= tx_data_reg[7:0]; // assume previously written
                                tx_start <= 1'b1;
                            end
                            // rx_rst or tx_rst handling (bits 2 & 3)
                            if (pwdata[2]) begin

                            end
                            if (pwdata[3]) begin
                                rx_reset <= 1'b1; // pulse reset to RX if desired
                            end
                        end
```

```
                        ADDR_TX: begin
                            tx_data_reg <= pwdata;

                        end

                        ADDR_BAUD: begin
                            bauddiv_reg <= pwdata;
                        end

                        default: begin
                            // ignore
                        end
                    endcase
                end else begin
                    // read transaction
                    case (paddr)
                        ADDR_CTRL: prdata <= ctrl_reg;
                        ADDR_STATUS: prdata <= status_reg;
                        ADDR_TX: prdata <= tx_data_reg;
                        ADDR_RX: prdata <= {24'd0, rx_data}; // low byte = rx_data
                        ADDR_BAUD: prdata <= bauddiv_reg;
                        default: prdata <= 32'hDEAD_BEEF;
                    endcase
                end
            end
        end
    end

    assign baud_div = bauddiv_reg;

endmodule
```

TEST BUNCH FOR APB :-

```
module tb_apb_uart;
    reg pclk = 0;
    always #5 pclk = ~pclk; // 100MHz

    reg presetn;

    reg [31:0] paddr;
    reg psel;
    reg penable;
    reg pwrite;
    reg [31:0] pwdata;
    wire [31:0] prdata;
    wire pready;

    // regs for status and rx
    reg [31:0] s;
    reg [31:0] rxd;

    // wires to UART modules
    wire [31:0] baud_div;
    wire tx_start;
    wire [7:0] tx_data;
    wire tx_busy;
    wire tx_done;
    wire [7:0] rx_data;
    wire rx_done;
    wire rx_busy;
    wire rx_error;

    // instantiate apb_uart
    apb_uart uut_apb (
        .pclk(pclk),
        .presetn(presetn),
        .paddr(paddr),
        .psel(psel),
        .penable(penable),
        .pwrite(pwrite),
        .pwdata(pwdata),
        .prdata(prdata),
        .pready(pready),
        .baud_div(baud_div),
        .tx_start(tx_start),
        .tx_data(tx_data),
        .tx_busy(tx_busy),
        .tx_done(tx_done),
```

```
        .rx_reset(),
        .rx_data(rx_data),
        .rx_done(rx_done),
        .rx_busy(rx_busy),
        .rx_error(rx_error)
    );

    // loopback
    wire tx_line;
    uart_tx UTX (
        .clk(pclk),
        .rst(~presetn),
        .baud_div(baud_div),
        .tx_start(tx_start),
        .tx_data(tx_data),
        .tx(tx_line),
        .tx_busy(tx_busy),
        .tx_done(tx_done)
    );

    uart_rx URX (
        .clk(pclk),
        .rst(~presetn),
        .baud_div(baud_div),
        .rx_pin(tx_line),
        .rx_data(rx_data),
        .rx_done(rx_done),
        .rx_busy(rx_busy),
        .rx_error(rx_error)
    );

    // APB tasks
    task apb_write;
        input [31:0] addr;
        input [31:0] data;
        begin
            @(posedge pclk);
            paddr   <= addr;
            pwdata  <= data;
            pwrite  <= 1;
            psel    <= 1;
            penable <= 1;
            @(posedge pclk);
            while (!pready) @(posedge pclk);
            psel    <= 0;
            penable <= 0;
            pwrite  <= 0;
            @(posedge pclk);
        end
    endtask

    task apb_read;
        input  [31:0] addr;
        output [31:0] data;
        begin
            @(posedge pclk);
            paddr   <= addr;
            pwrite  <= 0;
            psel    <= 1;
            penable <= 1;
            @(posedge pclk);
            while (!pready) @(posedge pclk);
            data = prdata;
            psel    <= 0;
            penable <= 0;
            @(posedge pclk);
        end
    endtask

    initial begin
        $dumpfile("tb_apb_uart.vcd");
        $dumpvars(0,tb_apb_uart);
        presetn = 0;
        psel = 0; penable = 0; pwrite = 0; paddr = 0; pwdata = 0;
        #40;
        presetn = 1;
        #40;

        // write TX_DATA = 0x5A
        apb_write(32'h0002, 32'h0000005A);

        // pulse CTRL.tx_en = bit0
        apb_write(32'h0000, 32'h1);

        // poll STATUS until tx_done
        repeat (2000) begin
            apb_read(32'h0001, s);
            if (s[3]) begin
                $display("APB saw tx_done at time %0t", $time);
                disable poll_loop;
            end
            #100;
        end
        poll_loop ;

        // read RX_DATA
        apb_read(32'h0003, rxd);
        $display("APB read RX_DATA = %02h", rxd[7:0]);

        #200;
        $finish;
    end
endmodule
```

RTL RESULTS FOR APB :-