perplexity

We are generating a 8-bit UART hierarchically in Verilog. We have generated BaudRateGenerator, Uart8Receiver defined as follows:

// Baud Rate Generator Module
module BaudRateGenerator #(
parameter CLOCK_RATE = 100000000,
parameter BAUD_RATE = 9600,
parameter RX_OVERSAMPLE_RATE = 16
)(
input wire clk,
output wire rxClk,
output wire txClk
);

```
// Calculate the number of clock cycles per baud period for RX and TX
localparam RX_BAUD_CYCLES = CLOCK_RATE / (BAUD_RATE * RX_OVERSAMPLE_RATE);
localparam TX_BAUD_CYCLES = CLOCK_RATE / BAUD_RATE;

// Counters for RX and TX baud rate generation
reg [31:0] rxCounter = 0;
reg [31:0] txCounter = 0;

// Signals to generate RX and TX clocks
reg rxClkEnable = 0;
reg txClkEnable = 0;

always @(posedge clk) begin
    // Increment RX counter
    rxCounter <= rxCounter + 1;

    // Generate RX clock
    if (rxCounter == RX_BAUD_CYCLES - 1) begin
        rxCounter <= 0;
        rxClkEnable <= ~rxClkEnable;
    end

    // Increment TX counter
    txCounter <= txCounter + 1;

    // Generate TX clock
    if (txCounter == TX_BAUD_CYCLES - 1) begin
        txCounter <= 0;
        txClkEnable <= ~txClkEnable;
    end
end

// Assign output clocks using continuous assignment
assign rxClk = rxClkEnable;
assign txClk = txClkEnable;
```

endmodule

// UART Receiver Module
module Uart8Receiver #(
parameter RX_OVERSAMPLE_RATE = 16
)(
input wire clk,
input wire en,
input wire in,
output reg busy,
output reg done,
output reg err,
output reg [7:0] out
);

```
// Instantiate Baud Rate Generator
wire rxClk;
wire txClk; // Not used in receiver but included for completeness
BaudRateGenerator baudGen(
    .clk(clk),
    .rxClk(rxClk),
    .txClk(txClk)
);

// States for the receiver FSM
localparam IDLE = 0;
localparam START = 1;
localparam DATA = 2;
localparam STOP = 3;

reg [2:0] state = IDLE;

// Counter for oversampling
reg [3:0] oversampleCounter = 0;

// Data register
reg [7:0] data = 0;

// Bit counter
reg [2:0] bitCounter = 0;

always @(posedge rxClk) begin
    case (state)
        IDLE: begin
            busy <= 0;
            done <= 0;
            err <= 0;
            if (en && in == 0) begin // Start bit detected
                state <= START;
                oversampleCounter <= 0;
            end
        end

        START: begin
            if (oversampleCounter == RX_OVERSAMPLE_RATE / 2 - 1) begin
                if (in == 0) begin // Start bit confirmed
                    state <= DATA;
                    oversampleCounter <= 0;
                    bitCounter <= 0;
                end else begin // Error: Not a start bit
                    state <= IDLE;
                    err <= 1;
                end
            end else begin
                oversampleCounter <= oversampleCounter + 1;
            end
        end

        DATA: begin
            if (oversampleCounter == RX_OVERSAMPLE_RATE / 2 - 1) begin
                data[7 - bitCounter] <= in;
                bitCounter <= bitCounter + 1;
                oversampleCounter <= 0;
                if (bitCounter == 8) begin
                    state <= STOP;
                end
            end else begin
                oversampleCounter <= oversampleCounter + 1;
            end
        end

        STOP: begin
            if (oversampleCounter == RX_OVERSAMPLE_RATE / 2 - 1) begin
                if (in == 1) begin // Stop bit confirmed
                    state <= IDLE;
                    busy <= 1;
                    done <= 1;
                    out <= data;
                end else begin // Error: Not a stop bit
                    state <= IDLE;
                    err <= 1;
                end
            end else begin
                oversampleCounter <= oversampleCounter + 1;
            end
        end

        default: state <= IDLE;
    endcase
end

always @(posedge clk) begin
```

```
    if (state == IDLE) begin
        busy <= 0;
    end else begin
        busy <= 1;
    end
end
```

endmodule

Please include the previous module(s) in your response and use them to hierarchically generate a Transmitter defined as:
module Uart8Transmitter #(
parameter TURBO_FRAMES = 0
)(
input wire clk,
input wire en,
input wire start,
input wire [7:0] in,
output reg busy,
output reg done,
output reg out
)
//Insert code here
endmodule

Here's how you can implement the `Uart8Transmitter` module using the `BaudRateGenerator` module for generating the transmit clock:

```
// Baud Rate Generator Module
module BaudRateGenerator #(
    parameter CLOCK_RATE = 100000000,
    parameter BAUD_RATE = 9600,
    parameter RX_OVERSAMPLE_RATE = 16
)(
    input wire clk,
    output wire rxClk,
    output wire txClk
);

    // Calculate the number of clock cycles per baud period for RX and TX
    localparam RX_BAUD_CYCLES = CLOCK_RATE / (BAUD_RATE * RX_OVERSAMPLE_RATE);
    localparam TX_BAUD_CYCLES = CLOCK_RATE / BAUD_RATE;

    // Counters for RX and TX baud rate generation
    reg [31:0] rxCounter = 0;
    reg [31:0] txCounter = 0;

    // Signals to generate RX and TX clocks
    reg rxClkEnable = 0;
    reg txClkEnable = 0;

    always @(posedge clk) begin
```

```
        // Increment RX counter
        rxCounter <= rxCounter + 1;

        // Generate RX clock
        if (rxCounter == RX_BAUD_CYCLES - 1) begin
            rxCounter <= 0;
            rxClkEnable <= ~rxClkEnable;
        end

        // Increment TX counter
        txCounter <= txCounter + 1;

        // Generate TX clock
        if (txCounter == TX_BAUD_CYCLES - 1) begin
            txCounter <= 0;
            txClkEnable <= ~txClkEnable;
        end
    end

    // Assign output clocks using continuous assignment
    assign rxClk = rxClkEnable;
    assign txClk = txClkEnable;

endmodule

// UART Receiver Module
module Uart8Receiver #(
    parameter RX_OVERSAMPLE_RATE = 16
)(
    input wire clk,
    input wire en,
    input wire in,
    output reg busy,
    output reg done,
    output reg err,
    output reg [7:0] out
);

    // Instantiate Baud Rate Generator
    wire rxClk;
    wire txClk; // Not used in receiver but included for completeness
    BaudRateGenerator baudGen(
        .clk(clk),
        .rxClk(rxClk),
        .txClk(txClk)
    );

    // States for the receiver FSM
    localparam IDLE = 0;
    localparam START = 1;
    localparam DATA = 2;
    localparam STOP = 3;

    reg [2:0] state = IDLE;

    // Counter for oversampling
    reg [3:0] oversampleCounter = 0;

    // Data register
    reg [7:0] data = 0;

    // Bit counter
    reg [2:0] bitCounter = 0;

    always @(posedge rxClk) begin
        case (state)
            IDLE: begin
                busy <= 0;
                done <= 0;
                err <= 0;
                if (en && in == 0) begin // Start bit detected
                    state <= START;
                    oversampleCounter <= 0;
                end
            end

            START: begin
                if (oversampleCounter == RX_OVERSAMPLE_RATE / 2 - 1) begin
                    if (in == 0) begin // Start bit confirmed
                        state <= DATA;
                        oversampleCounter <= 0;
                        bitCounter <= 0;
                    end else begin // Error: Not a start bit
                        state <= IDLE;
                        err <= 1;
                    end
                end else begin
                    oversampleCounter <= oversampleCounter + 1;
                end
            end

            DATA: begin
                if (oversampleCounter == RX_OVERSAMPLE_RATE / 2 - 1) begin
                    data[7 - bitCounter] <= in;
                    bitCounter <= bitCounter + 1;
                    oversampleCounter <= 0;
                    if (bitCounter == 8) begin
                        state <= STOP;
                    end
                end else begin
                    oversampleCounter <= oversampleCounter + 1;
                end
            end

            STOP: begin
                if (oversampleCounter == RX_OVERSAMPLE_RATE / 2 - 1) begin
                    if (in == 1) begin // Stop bit confirmed
                        state <= IDLE;
                        busy <= 1;
                        done <= 1;
                        out <= data;
                    end else begin // Error: Not a stop bit
```

```
                        state <= IDLE;
                        err <= 1;
                    end
                end else begin
                    oversampleCounter <= oversampleCounter + 1;
                end
            end

            default: state <= IDLE;
        endcase
    end

    always @(posedge clk) begin
        if (state == IDLE) begin
            busy <= 0;
        end else begin
            busy <= 1;
        end
    end

endmodule

// UART Transmitter Module
module Uart8Transmitter #(
    parameter TURBO_FRAMES = 0
)(
    input wire clk,
    input wire en,
    input wire start,
    input wire [7:0] in,
    output reg busy,
    output reg done,
    output reg out
);

    // Instantiate Baud Rate Generator
    wire rxClk; // Not used in transmitter but included for completeness
    wire txClk;
    BaudRateGenerator baudGen(
        .clk(clk),
        .rxClk(rxClk),
        .txClk(txClk)
    );

    // States for the transmitter FSM
    localparam IDLE = 0;
    localparam START = 1;
    localparam DATA = 2;
    localparam STOP = 3;

    reg [2:0] state = IDLE;

    // Data register
    reg [7:0] data = 0;

    // Bit counter
    reg [2:0] bitCounter = 0;

    always @(posedge txClk) begin
        case (state)
            IDLE: begin
                busy <= 0;
                done <= 0;
                out <= 1; // Idle state: High output
                if (en && start) begin
                    data <= in;
                    state <= START;
                    bitCounter <= 0;
                end
            end

            START: begin
                out <= 0; // Start bit
                state <= DATA;
            end

            DATA: begin
                out <= data[7 - bitCounter];
                bitCounter <= bitCounter + 1;
                if (bitCounter == 8) begin
                    state <= STOP;
                end
            end

            STOP: begin
                out <= 1; // Stop bit
                state <= IDLE;
                busy <= 1;
                done <= 1;
            end

            default: state <= IDLE;
        endcase
    end

    always @(posedge clk) begin
        if (state != IDLE) begin
            busy <= 1;
        end
    end

endmodule
```

This transmitter module uses a finite state machine (FSM) to manage the transmission of UART data. It includes states for idle, start bit transmission, data transmission, and stop bit transmission. The `BaudRateGenerator` module is used to generate the transmit clock (`txClk`) based on the system clock (`clk`).