PART - B

Test bench code

```
`timescale 1ns / 1ps

module tb_receiver;

    // Inputs
    reg clk;
    reg rst;
    reg rx;

    // Outputs
    wire [3:0] data;
    wire done;

    // Instantiate the receiver
    receiver uut (
        .clk(clk),
        .rst(rst),
        .rx(rx),
        .data(data),
        .done(done)
    );

    // Generate a clock: 10 ns period
    always begin
        #5 clk = ~clk;
    end

    // Task to send one 6-bit framed packet: start, 4 bits (LSB first), stop
    task send_packet(input [3:0] payload);
        integer i;
        begin
            @(posedge clk); #1 rx <= 0;  // Start bit
            @(posedge clk); #1;

            for (i = 0; i < 4; i = i + 1) begin
                rx <= payload[i];
                @(posedge clk); #1;
            end

            rx <= 1;  // Stop bit
            @(posedge clk); #1;
        end
```

```
endtask

initial begin
    // Initial values
    clk = 0;
    rst = 1;
    rx  = 1;

    // Hold reset for 2 cycles
    repeat (2) @(posedge clk);
    rst <= 0;

    // Send valid packet: 4'b1010
    send_packet(4'b1010);

    // Idle for 4 cycles
    repeat (4) @(posedge clk);

    // Send valid packet: 4'b0111
    send_packet(4'b0111);

    // Idle for 4 cycles
    repeat (4) @(posedge clk);

    // Send invalid packet (bad stop bit)
    @(posedge clk); #1 rx <= 0;  // Start bit
    @(posedge clk); #1 rx <= 1;
    @(posedge clk); #1 rx <= 0;
    @(posedge clk); #1 rx <= 1;
    @(posedge clk); #1 rx <= 1;
    @(posedge clk); #1 rx <= 0;  // Incorrect stop bit

    // Let it sit idle for recovery
    repeat (6) @(posedge clk); #1;
    rx <= 1;

    // Send one more valid packet: 4'b1100
    repeat (2) @(posedge clk);
    send_packet(4'b1100);

    // End simulation
    repeat (10) @(posedge clk);
    $finish;
end

// Dump waveform for viewing in GTKWave or similar
initial begin
    $dumpfile("receiver.vcd");
```

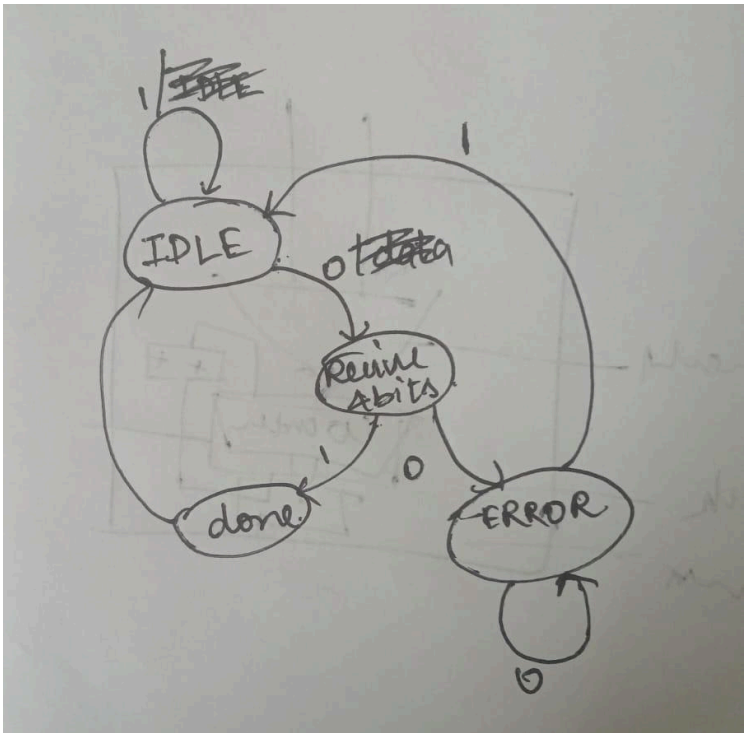
```
        $dumpvars(0, tb_receiver);
    end

endmodule
```

VERILOG CODE

```
module receiver (
    input clk,
    input rst,
    input rx,              // Serial bitstream input
    output reg [3:0] data, // Received 4-bit data
    output reg done        // Done pulse when valid byte received
);

    // Counter for 4 data bits
    reg [2:0] bit_cnt;
    reg [3:0] shift_reg;

    // Define FSM states using SystemVerilog enum
    typedef enum reg [2:0] {
        IDLE       = 3'd0,
        RECEIVE    = 3'd1,
        WAIT_STOP  = 3'd2,
        DONE       = 3'd3,
        ERROR      = 3'd4
    } state_t;

    state_t state, next_state;

    // Sequential block for state, output, and data
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            state      <= IDLE;
            bit_cnt    <= 3'd0;
            shift_reg  <= 4'd0;
            data       <= 4'd0;
            done       <= 1'b0;
        end else begin
            state <= next_state;

            case (state)
                IDLE: begin
                    done <= 1'b0;
                    if (rx == 1'b0) begin  // Start bit detected
                        bit_cnt   <= 3'd0;
                        shift_reg <= 4'd0;
                    end
                end

                RECEIVE: begin
```

```
                shift_reg <= {rx, shift_reg[3:1]};  // Shift in new bit
                bit_cnt   <= bit_cnt + 1;
            end

            WAIT_STOP: begin
                if (rx == 1'b1) begin  // Valid stop bit
                    data <= shift_reg;
                    done <= 1'b1;
                end
            end

            DONE: begin
                done <= 1'b0;  // Pulse done signal
            end

            ERROR: begin
                done <= 1'b0;  // Wait for idle line
            end
        endcase
    end
end

// Combinational next-state logic
always @(*) begin
    next_state = state;
    case (state)
        IDLE:
            next_state = (rx == 1'b0) ? RECEIVE : IDLE;

        RECEIVE:
            next_state = (bit_cnt == 3'd3) ? WAIT_STOP : RECEIVE;

        WAIT_STOP:
            next_state = (rx == 1'b1) ? DONE : ERROR;

        DONE:
            next_state = IDLE;

        ERROR:
            next_state = (rx == 1'b1) ? IDLE : ERROR;
    endcase
end

endmodule
```

STATE DIAGRAM:



This is a Verilog implementation of a serial bit receiver using a finite state machine (FSM). The design follows a standard framed format where each valid transmission consists of a start bit (`0`), followed by 4 data bits, and ends with a stop bit (`1`). The line remains at logic high (`1`) when idle.

The FSM has the following states:

- **`IDLE`: waits for the start bit**
- **`RECEIVE`: collects 4 bits one by one**
- **`WAIT_STOP`: checks for the stop bit**
- **`DONE`: indicates the data is valid and received**
- **`ERROR`: entered if the stop bit is invalid**

The FSM uses a `state` and `next_state` approach. The current state is updated every clock cycle, and transitions are determined based on inputs. The output `done` is asserted only in the `DONE` state, making it a Moore FSM.

When in `IDLE`, if a `0` is detected on `rx`, it moves to `RECEIVE`. During the `RECEIVE` state, a 4-bit shift register stores incoming bits. After 4 bits, it expects a stop bit. If the

stop bit is correct (`rx == 1`), it transitions to `DONE` and outputs `done = 1` for one cycle. If the stop bit is not 1, it goes to `ERROR`, where it waits until the line goes idle again before returning to `IDLE`.

This type of FSM can be used in low-level serial data handling systems like UART or other custom serial protocols. The design uses `typedef enum` for defining state values and ensures stable transitions with synchronous reset and proper error recovery.

This receiver can be further extended with parity checking, wider data frames, or buffering using a FIFO for continuous reception.