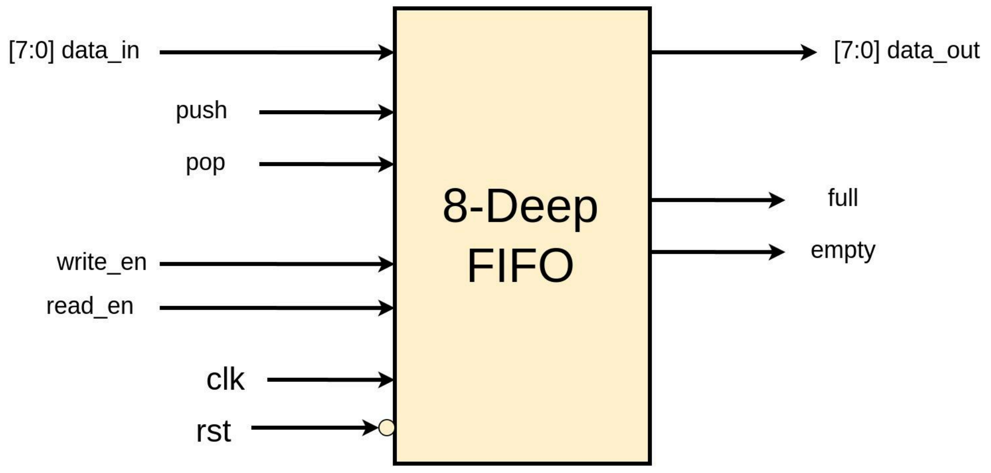
Module: R3: DLD + DSD
Section: Sequential Circuits **Task:** Final Project

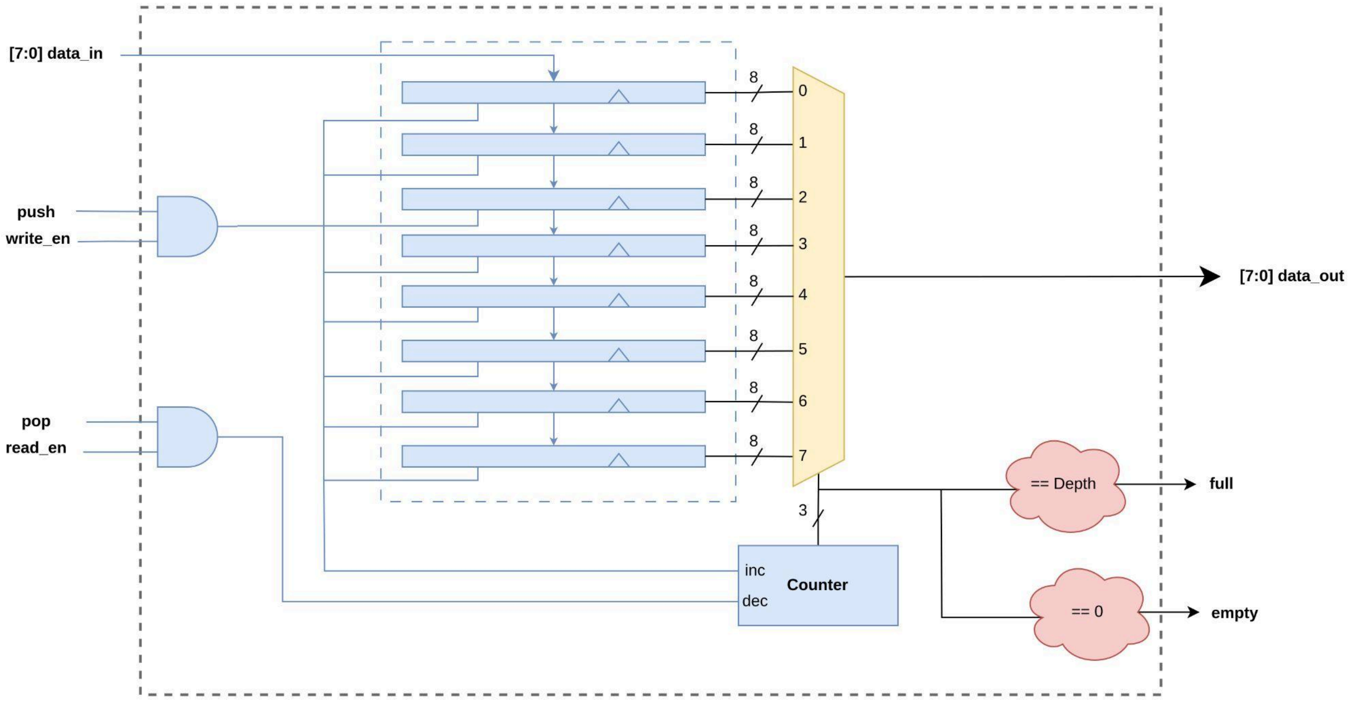
Design Problem

FIFO Design

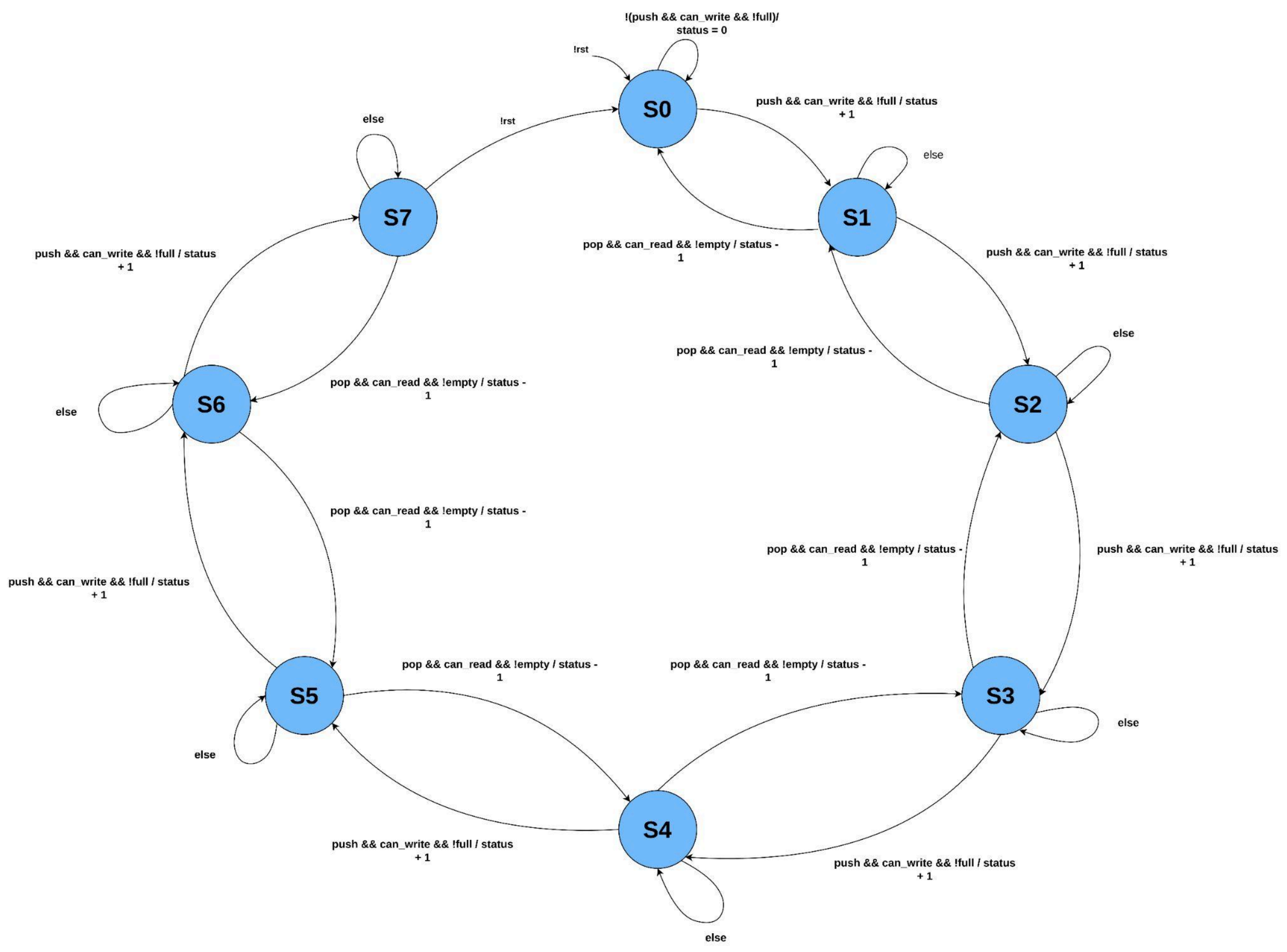
➢ **Question: Design an Asynchronous active low reset FIFO:**

1. **Schematic Diagram:**





2. **FSM Diagram:**


S0
S1
S2
S3
S4
S5
S6
S7
!(push && can_write && !full)/ status = 0
!rst
!rst
push && can_write && !full / status + 1
pop && can_read && !empty / status - 1
else

3. Verilog Code:

```
    module fifo #(parameter DATA_SIZE = 8, parameter ADDRESS_SIZE
= 3)
             (input clk, rst, push, pop,
              input [DATA_SIZE-1:0] data_in,
              input can_read, can_write,
              output reg [DATA_SIZE-1:0] data_out,
              output reg full, empty);

    localparam ADDRESS_DEPTH = 2**ADDRESS_SIZE;
    reg [DATA_SIZE-1:0] memory [ADDRESS_DEPTH-1:0];
    reg [ADDRESS_SIZE-1:0] read_ptr, write_ptr;
    reg [ADDRESS_SIZE:0] status; // Extra bit for distinguishing
full and empty
```

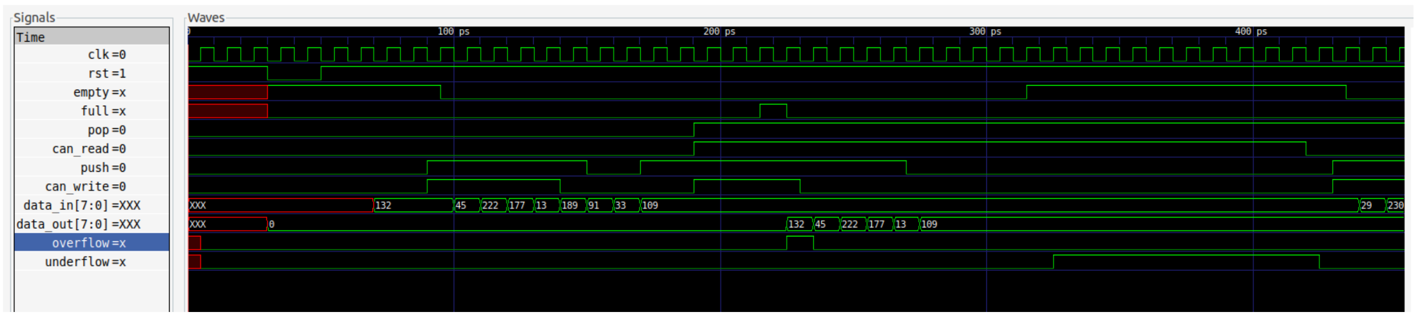
```
    reg underflow, overflow;

    // States
    localparam S0 = 3'b000;
    localparam S1 = 3'b001;
    localparam S2 = 3'b010;
    localparam S3 = 3'b011;
    localparam S4 = 3'b100;
    localparam S5 = 3'b101;
    localparam S6 = 3'b110;
    localparam S7 = 3'b111;

    reg [2:0] state, next_state;

    always @(posedge clk or negedge rst) begin
        if (!rst) begin
            state <= S0;
            data_out <= 0;
            status <= 0;
            read_ptr <= 0;
            write_ptr <= 0;
            overflow <= 0;
            underflow <= 0;
            full <= 0;
            empty <= 1;
        end else begin
            state <= next_state;
        end
    end

    always @(posedge clk) begin
        if (!rst) begin
            data_out <= 0;
            status <= 0;
            read_ptr <= 0;
            write_ptr <= 0;
            overflow <= 0;
            underflow <= 0;
            full <= 0;
            empty <= 1;
        end else begin
            case (state)
```

```
                S0: begin
                    if (push && can_write && !full) begin
                        memory[write_ptr] <= data_in;
                        write_ptr <= write_ptr + 1;
                        status <= status + 1;
                        next_state <= S1;
                    end else begin
                        next_state <= S0;
                    end
                end

                S1: begin
                    if (push && can_write && !full) begin
                        memory[write_ptr] <= data_in;
                        write_ptr <= write_ptr + 1;
                        status <= status + 1;
                        next_state <= S2;
                    end else if (pop && can_read && !empty) begin
                        data_out <= memory[read_ptr];
                        read_ptr <= read_ptr + 1;
                        status <= status - 1;
                        next_state <= S0;
                    end else begin
                        next_state <= S1;
                    end
                end

                S2: begin
                    if (push && can_write && !full) begin
                        memory[write_ptr] <= data_in;
                        write_ptr <= write_ptr + 1;
                        status <= status + 1;
                        next_state <= S3;
                    end else if (pop && can_read && !empty) begin
                        data_out <= memory[read_ptr];
                        read_ptr <= read_ptr + 1;
                        status <= status - 1;
                        next_state <= S1;
                    end else begin
                        next_state <= S2;
                    end
                end
                S3: begin
                    if (push && can_write && !full) begin
                        memory[write_ptr] <= data_in;
                        write_ptr <= write_ptr + 1;
                        status <= status + 1;
                        next_state <= S4;
                    end else if (pop && can_read && !empty) begin
                        data_out <= memory[read_ptr];
                        read_ptr <= read_ptr + 1;
                        status <= status - 1;
                        next_state <= S2;
                    end else begin
                        next_state <= S3;
                    end
                end

                S4: begin
                    if (push && can_write && !full) begin
                        memory[write_ptr] <= data_in;
                        write_ptr <= write_ptr + 1;
                        status <= status + 1;
                        next_state <= S5;
                    end else if (pop && can_read && !empty) begin
                        data_out <= memory[read_ptr];
                        read_ptr <= read_ptr + 1;
                        status <= status - 1;
                        next_state <= S3;
                    end else begin
                        next_state <= S4;
                    end
                end

                S5: begin
                    if (push && can_write && !full) begin
                        memory[write_ptr] <= data_in;
                        write_ptr <= write_ptr + 1;
                        status <= status + 1;
                        next_state <= S6;
                    end else if (pop && can_read && !empty) begin
                        data_out <= memory[read_ptr];
                        read_ptr <= read_ptr + 1;
                        status <= status - 1;
                        next_state <= S4;
                    end else begin
                        next_state <= S5;
                    end
                end

                S6: begin
                    if (push && can_write && !full) begin
                        memory[write_ptr] <= data_in;
                        write_ptr <= write_ptr + 1;
                        status <= status + 1;
                        next_state <= S7;
                    end else if (pop && can_read && !empty) begin
                        data_out <= memory[read_ptr];
                        read_ptr <= read_ptr + 1;
                        status <= status - 1;
                        next_state <= S5;
                    end else begin
                        next_state <= S6;
                    end
                end

                S7: begin
                    if (push && can_write) begin
                        next_state <= S7;
                    end else if (pop && can_read && !empty) begin
                        data_out <= memory[read_ptr];
                        read_ptr <= read_ptr + 1;
                        status <= status - 1;
                        next_state <= S6;
                    end else begin
                        next_state <= S7;
                    end
                end
            endcase
        end
    end

    always @* begin
        full = (status == ADDRESS_DEPTH);
        empty = (status == 0);
    end

    always @(posedge clk) begin
        if (state == S0 && empty && pop && can_read)
            underflow <= 1;
        else if( push && can_write && full )
            overflow <= 1;
        else begin
            underflow <= 0;
            overflow <= 0;
        end
    end

endmodule
```

4. Testbench:

```
    module tb_fifo;

    parameter DATA_SIZE = 8;
    parameter ADDRESS_SIZE = 3;

    reg clk, rst, push, pop;
    reg [DATA_SIZE-1 : 0] data_in;
    reg can_read;
    reg can_write;

    wire [DATA_SIZE-1 : 0] data_out;
    wire full, empty;

    fifo dut (.clk(clk),
                .rst(rst),
                .push(push),
                .pop(pop),
                .data_in(data_in),
                .can_read(can_read),
                .can_write(can_write),
                .data_out(data_out),
                .full(full),
                .empty(empty));

    always #5 clk = ~clk;
    initial begin

        $dumpvars;

        clk = 0;
        rst = 1;
        push = 0;
        pop = 0;
        can_read = 0;
        can_write = 0;
        #30;

        rst = 0;
        #20;

        rst = 1;
        #20;

        data_in = 8'd132;
        #20;

        push = 1;
        can_write = 1;
        #10;

        data_in = 8'd45;
        #10;

        data_in = 8'd222;
        #10;

        data_in = 8'd177;
        #10;

        data_in = 8'd13;
        #10;

        data_in = 8'd189;
        can_write = 0;
        #10;

        data_in = 8'd91;
        push = 0;
        #10;

        data_in = 8'd33;
        #10;

        data_in = 8'd109;
        push = 1;

        #20;
        can_write = 1;
        pop = 1;
        can_read = 1;

        #40;
        pop = 1;
        can_read = 1;
        can_write = 0;

        #40;

        push = 0;
        #150;

        can_read = 0;

        #10;
        push = 1;
        can_write = 1;


        #10 data_in = 8'd29;
        #10 data_in = 8'd230;
        #10 data_in = 8'd138;
        #10 data_in = 8'd213;
        #10 data_in = 8'd254;
        #10 data_in = 8'd243;
        #10 data_in = 8'd107;
        #10 data_in = 8'd85;
        #10 data_in = 8'd12;

        #20 push = 0;
        can_read = 1;

        #120; //Reading from Empty FIFO

        can_read = 0;
        pop = 0;
        push = 1;

        #10 data_in = 8'd129;
        #10 data_in = 8'd209;
        #10 data_in = 8'd157;
        #10 data_in = 8'd13;
        #10 data_in = 8'd54;
        #10 data_in = 8'd103;
        #10 data_in = 8'd247;
        #10 data_in = 8'd51;
        #10 data_in = 8'd186;

        #20 rst = 0;

        #30 rst = 1;

        #120;

        $finish;
    end

    endmodule
```

5. Output:


Signals
Waves
Time
clk=0
rst=1
empty=x
full=x
pop=0
can_read=0
push=0
can_write=0
data_in[7:0]=XXX
data_out[7:0]=XXX
overflow=x
underflow=x
100 ps
200 ps
300 ps
400 ps
XXX
XXX