**Module: R3: DLD + DSD**

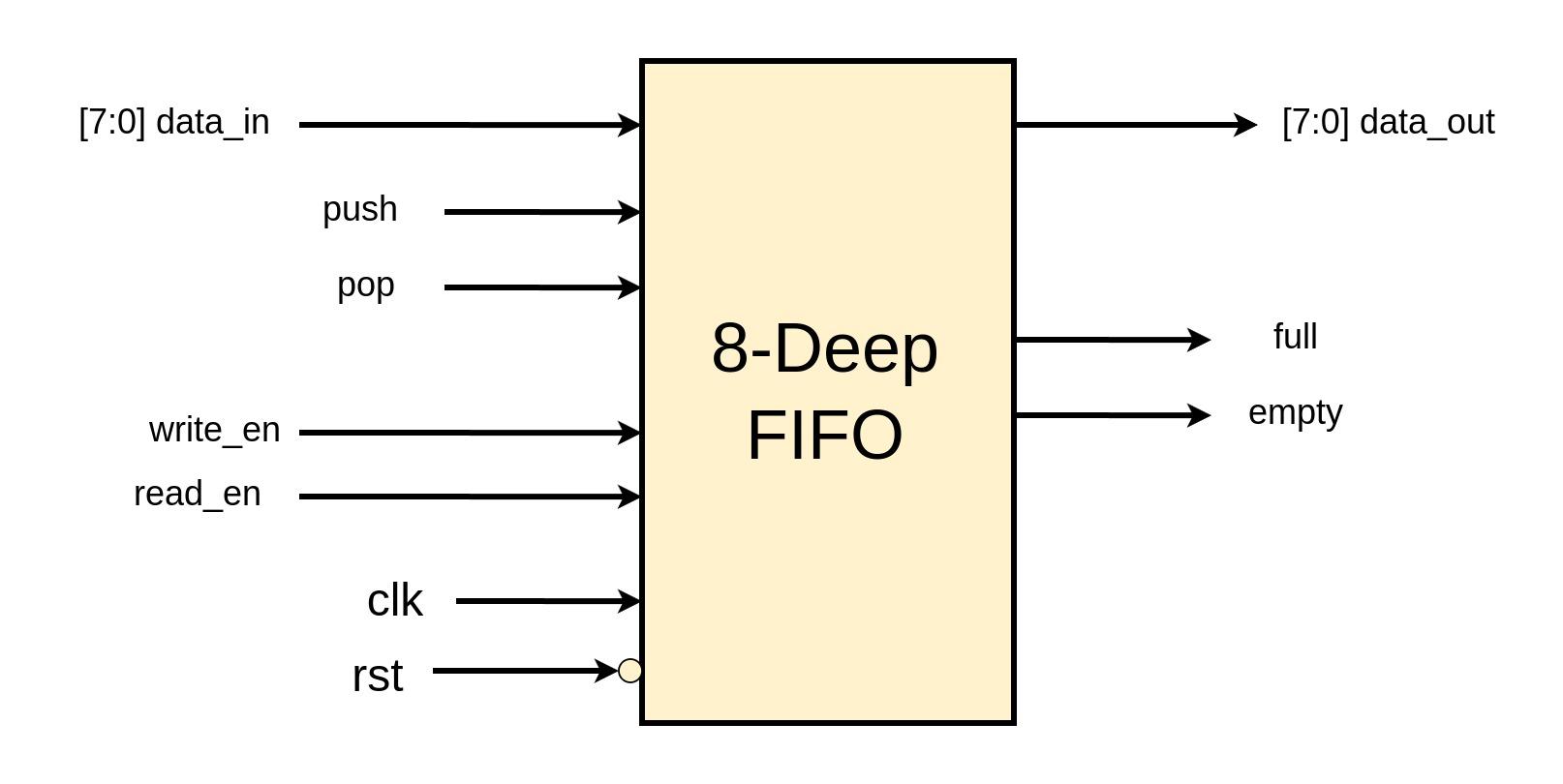
**Section:** Sequential Circuits **Task:** Final Project

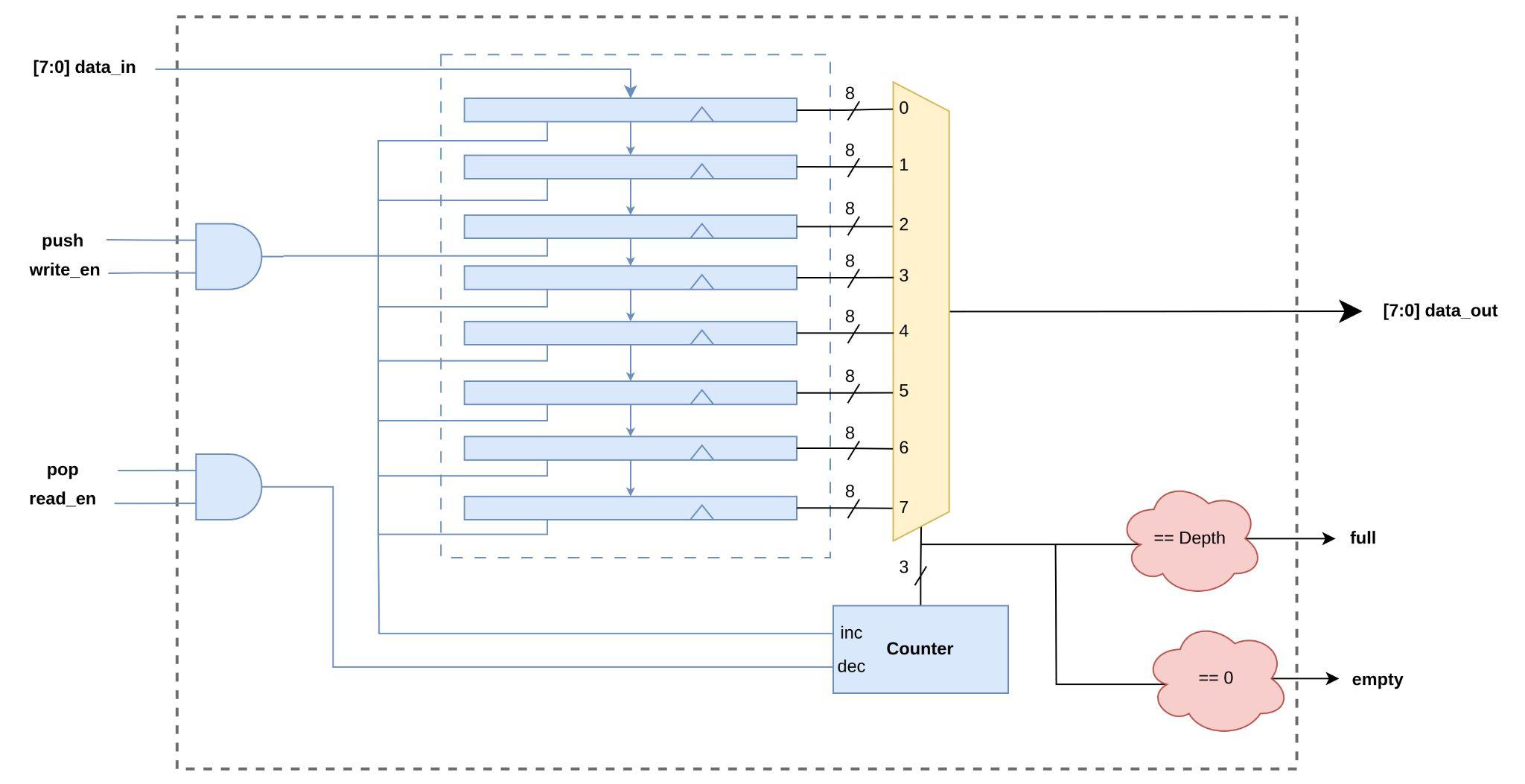
**Design Problem**

**FIFO Design**

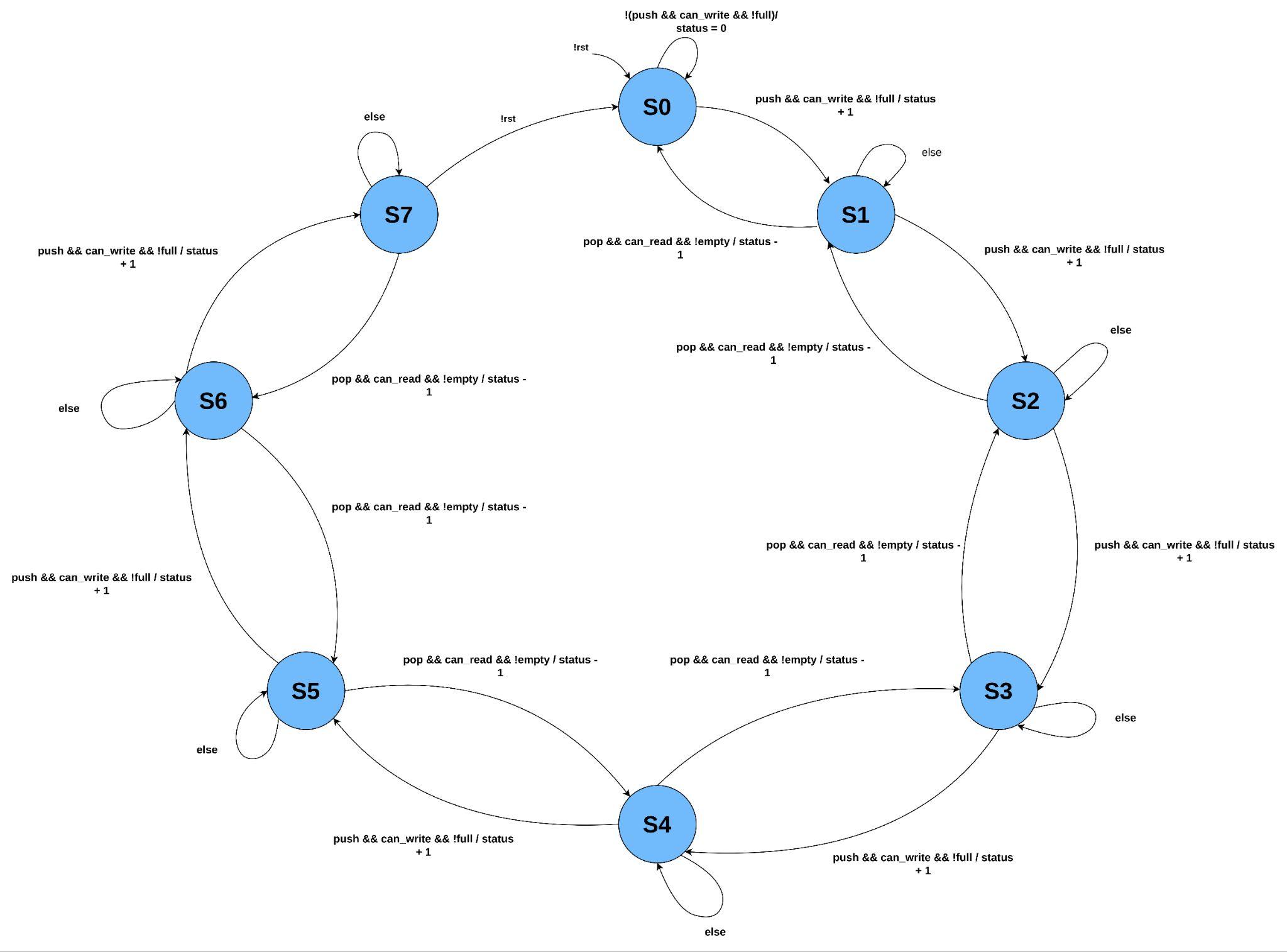
* **Question: Design an Asynchronous active low reset FIFO:**

1. **Schematic Diagram:**

****

****

1. **FSM Diagram:**

****

1. **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

1. **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;

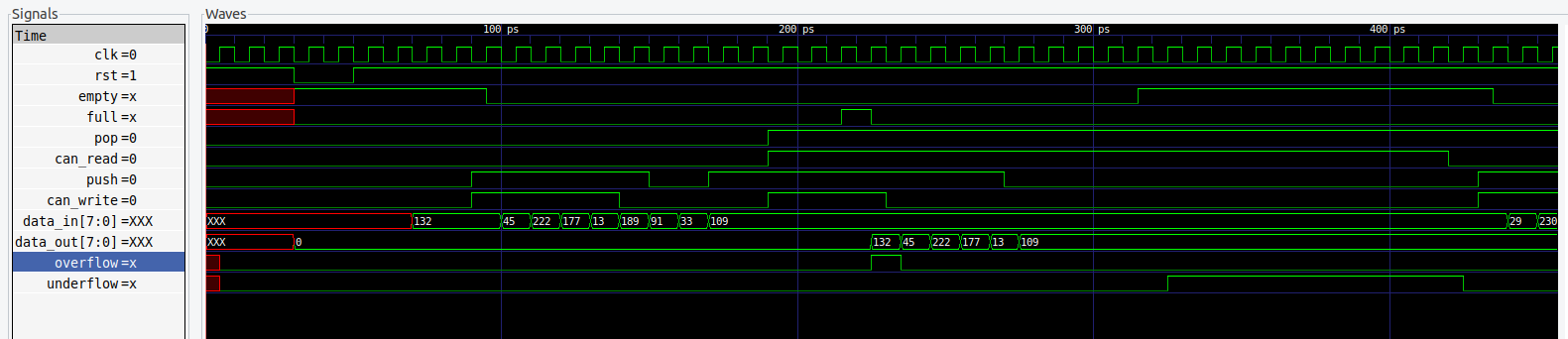
#120;

$finish;

end

endmodule

1. **Output:**

****