module fifo\_pointer\_logic #(

parameter ADDR\_WIDTH = 4 // FIFO depth = 2^ADDR\_WIDTH

)(

input wire wr\_clk,

input wire wr\_en,

output wire full,

input wire rd\_clk,

input wire rd\_en,

output wire empty

);

// Internal Signals

reg [ADDR\_WIDTH:0] wr\_bin = 0;

reg [ADDR\_WIDTH:0] wr\_gray = 0;

wire [ADDR\_WIDTH:0] wr\_gray\_next;

wire [ADDR\_WIDTH:0] wr\_bin\_next;

reg [ADDR\_WIDTH:0] rd\_bin = 0;

reg [ADDR\_WIDTH:0] rd\_gray = 0;

wire [ADDR\_WIDTH:0] rd\_gray\_next;

wire [ADDR\_WIDTH:0] rd\_bin\_next;

// Synchronizer outputs

wire [ADDR\_WIDTH:0] sync\_rd\_gray\_to\_wr;

wire [ADDR\_WIDTH:0] sync\_wr\_gray\_to\_rd;

//----------------------------------------

// Write Pointer Logic

//----------------------------------------

assign wr\_bin\_next = wr\_bin + (wr\_en & ~full);

assign wr\_gray\_next = (wr\_bin\_next >> 1) ^ wr\_bin\_next;

always @(posedge wr\_clk) begin

if (wr\_en & ~full) begin

wr\_bin <= wr\_bin\_next;

wr\_gray <= wr\_gray\_next;

end

end

//----------------------------------------

// Read Pointer Logic

//----------------------------------------

assign rd\_bin\_next = rd\_bin + (rd\_en & ~empty);

assign rd\_gray\_next = (rd\_bin\_next >> 1) ^ rd\_bin\_next;

always @(posedge rd\_clk) begin

if (rd\_en & ~empty) begin

rd\_bin <= rd\_bin\_next;

rd\_gray <= rd\_gray\_next;

end

end

//----------------------------------------

// Full and Empty Flag Logic

//----------------------------------------

// Full: if next wr\_gray == inverse MSBs of synced rd\_gray + LSBs same

assign full = (wr\_gray\_next == {~sync\_rd\_gray\_to\_wr[ADDR\_WIDTH:ADDR\_WIDTH-1],

sync\_rd\_gray\_to\_wr[ADDR\_WIDTH-2:0]});

// Empty: if rd\_gray == synced wr\_gray

assign empty = (rd\_gray == sync\_wr\_gray\_to\_rd);

//----------------------------------------

// Synchronizer Modules (2-FF)

//----------------------------------------

synchronizer #(.WIDTH(ADDR\_WIDTH+1)) sync\_rd\_to\_wr (

.clk(wr\_clk),

.async\_in(rd\_gray),

.sync\_out(sync\_rd\_gray\_to\_wr)

);

synchronizer #(.WIDTH(ADDR\_WIDTH+1)) sync\_wr\_to\_rd (

.clk(rd\_clk),

.async\_in(wr\_gray),

.sync\_out(sync\_wr\_gray\_to\_rd)

);

endmodule

//----------------------------------------

// 2-Flip-Flop Synchronizer Module

//----------------------------------------

module synchronizer #(

parameter WIDTH = 4

)(

input wire clk,

input wire [WIDTH-1:0] async\_in,

output reg [WIDTH-1:0] sync\_out

);

reg [WIDTH-1:0] sync\_ff1;

always @(posedge clk) begin

sync\_ff1 <= async\_in;

sync\_out <= sync\_ff1;

end

endmodule

//----------------------------------------

// Testbench for FIFO Pointer Logic

//----------------------------------------

module tb\_fifo\_pointer\_logic;

parameter ADDR\_WIDTH = 3;

reg wr\_clk = 0, rd\_clk = 0;

reg wr\_en = 0, rd\_en = 0;

wire full, empty;

// Instantiate the DUT

fifo\_pointer\_logic #(.ADDR\_WIDTH(ADDR\_WIDTH)) fifo (

.wr\_clk(wr\_clk),

.wr\_en(wr\_en),

.full(full),

.rd\_clk(rd\_clk),

.rd\_en(rd\_en),

.empty(empty)

);

// Generate write clock (10ns)

always #5 wr\_clk = ~wr\_clk;

// Generate read clock (15ns)

always #7.5 rd\_clk = ~rd\_clk;

initial begin

$display("Starting test...");

$dumpfile("fifo\_pointer.vcd");

$dumpvars(0, tb\_fifo\_pointer\_logic);

// Initially empty

#20;

// Write 5 times

repeat (5) begin

wr\_en = 1;

#10;

end

wr\_en = 0;

#50;

// Read 3 times

repeat (3) begin

rd\_en = 1;

#15;

end

rd\_en = 0;

#50;

// Fill FIFO to test full flag

wr\_en = 1;

repeat (8) #10;

wr\_en = 0;

#100;

// Empty FIFO completely

rd\_en = 1;

repeat (10) #15;

rd\_en = 0;

#100;

$finish;

end

endmodule