DSD Lab 8

Register file:

```
module RegisterFile (
   input logic [31:0] data_in,    // 32-bit data input
   input logic clk, reset, enable, // 1-bit clock, reset, enable inputs
   output logic [31:0] data_out    // 32-bit data output
);
   // Register to store data
   logic [31:0] reg_data;

   // Data update and reset logic
   always_ff @(posedge clk) begin
      if (reset) reg_data <= 32'b0;         // Reset on positive edge of reset
      else if (enable) reg_data <= data_in;  // Update on enable high
   end

   // Output assignment
   assign data_out = reg_data;

endmodule
```

Test Bench:

```
module test12;
// Inputs
reg [31:0] in;
reg clk;
reg reset;
reg en;
// Outputs
wire [31:0] out;
```

```
// Instantiate the Unit Under Test (UUT)
register_file uut (
.in(in),
.out(out),
.clk(clk),
.reset(reset),
.en(en)
);
always #5 clk = ~clk;
initial begin
// Initialize Inputs
in = 0;
clk = 0;
reset = 1;
en = 0;
#10 en = 1; reset = 0; in = 10;
#10 in = 20;
#10 in = 30;
#10 en = 0; in = 40;
#10 in = 50;
#10 in = 50;
#100 $finish;
end
endmodule
```

Example: Two bit ALU

```
module alu_2bit (
input [1:0] A, // First 2-bit input
input [1:0] B, // Second 2-bit input
input [1:0] ALU_Sel, // ALU select signal
output reg [2:0] Result // Output result (3 bits to account for carry/borrow)
);
always @(*) begin
case(ALU_Sel)
2'b00: Result = A & B; 2'b01: Result = A | B; 2'b10: Result = A + B; 2'b11: Result = A - B; default: Result = 3'b000;
endcase
// AND
// OR
// ADD
// SUB
end
```

endmodule

```
module ALU_tb;
   // Testbench signals
   logic [3:0] a, b;          // 4-bit inputs
   logic [2:0] opcode;        // 3-bit operation code
   logic [3:0] result;        // 4-bit output
   logic zero;                // Zero flag

   // Instantiate the ALU module
   ALU dut (
      .a(a),
      .b(b),
      .opcode(opcode),
      .result(result),
      .zero(zero)
   );

   // Clock generation (optional, if needed for future synchronous designs)
   logic clk = 0;
   always #5 clk = ~clk;

   // Stimulus
   initial begin
      // Initialize inputs
      a = 4'b0011; b = 4'b0001; opcode = 3'b000; #10; // Add: 3 + 1 = 4
      $display("Time=%0t Add: a=%b, b=%b, result=%b, zero=%b", $time, a, b, result, zero);

      a = 4'b0010; b = 4'b0001; opcode = 3'b001; #10; // Subtract: 2 - 1 = 1
      $display("Time=%0t Subtract: a=%b, b=%b, result=%b, zero=%b", $time, a, b, result,
zero);
```

```
        a = 4'b0011; b = 4'b0001; opcode = 3'b010; #10; // AND: 3 & 1 = 1
        $display("Time=%0t AND: a=%b, b=%b, result=%b, zero=%b", $time, a, b, result, zero);

        a = 4'b0000; b = 4'b0000; opcode = 3'b011; #10; // OR: 0 | 0 = 0
        $display("Time=%0t OR: a=%b, b=%b, result=%b, zero=%b", $time, a, b, result, zero);

        a = 4'b0011; b = 4'b0011; opcode = 3'b100; #10; // XOR: 3 ^ 3 = 0
        $display("Time=%0t XOR: a=%b, b=%b, result=%b, zero=%b", $time, a, b, result, zero);

        $finish; // End simulation
    end
endmodule
```

```
// ALU Module
module ALU (
    input  logic [31:0] A,
    input  logic [31:0] B,
    input  logic [2:0]  opcode,
    output logic [31:0] result
);
    always_comb begin
        case (opcode)
            3'b000: result = 32'd0;       // 0
            3'b001: result = A + B;       // A + B
            3'b010: result = A - B;       // A - B
            3'b011: result = A & B;       // A & B
            3'b100: result = A / B;       // A / B
            3'b101: result = ~A;          // ~A
```

```
        3'b110: result = A;          // NoP (Assume output is same as A)
        default: result = 32'd0;
      endcase
    end
endmodule

module tb_ALU;
    logic [31:0] A, B;
    logic [2:0] opcode;
    logic [31:0] result;

    ALU uut (
      .A(A),
      .B(B),
      .opcode(opcode),
      .result(result)
    );

    initial begin
      A = 32'd10; B = 32'd5;

      for (int i = 0; i < 7; i++) begin
        opcode = i[2:0];
        #10;
        $display("Opcode = %0d, Result = %0d", opcode, result);
      end

      $finish;
    end
endmodule

module RegisterFile (
```

```
    input  logic        clk,
    input  logic        reset,
    input  logic        RegWrite,
    input  logic [4:0]  read_reg1,
    input  logic [4:0]  read_reg2,
    input  logic [4:0]  write_reg,
    input  logic [31:0] write_data,
    output logic [31:0] read_data1,
    output logic [31:0] read_data2
);
    logic [31:0] registers [31:0];

    // Synchronous Write on Positive Edge
    always_ff @(posedge clk or posedge reset) begin
        if (reset)
            for (int i = 0; i < 32; i++)
                registers[i] <= 32'd0;
        else if (RegWrite)
            registers[write_reg] <= write_data;
    end

    // Asynchronous Read on Negative Edge
    always_ff @(negedge clk) begin
        read_data1 <= registers[read_reg1];
        read_data2 <= registers[read_reg2];
    end
endmodule

`timescale 1ns/1ps
module tb_RegisterFile;
    // Testbench signals
    reg         clk;
```

```
reg        reset;
reg        RegWrite;
reg  [4:0]  read_reg1;
reg  [4:0]  read_reg2;
reg  [4:0]  write_reg;
reg  [31:0] write_data;
wire [31:0] read_data1;
wire [31:0] read_data2;

// Instantiate the RegisterFile module
RegisterFile uut (
    .clk(clk),
    .reset(reset),
    .RegWrite(RegWrite),
    .read_reg1(read_reg1),
    .read_reg2(read_reg2),
    .write_reg(write_reg),
    .write_data(write_data),
    .read_data1(read_data1),
    .read_data2(read_data2)
);

// Clock generation: 10ns period
always #5 clk = ~clk;

// Stimulus
initial begin
    // Initialize signals
    clk = 0;
    reset = 1;
    RegWrite = 0;
    read_reg1 = 0;
```

```
        read_reg2 = 0;
        write_reg = 0;
        write_data = 0;

        // Monitor the signals
        $monitor("Time=%0t | Reset=%b | RegWrite=%b | WriteReg=%d | WriteData=%h |
ReadReg1=%d | ReadData1=%h | ReadReg2=%d | ReadData2=%h",
                 $time, reset, RegWrite, write_reg, write_data, read_reg1, read_data1, read_reg2,
read_data2);

        // Test reset
        #10 reset = 0;  // Release reset

        // Test 1: Write to register 5 and read from it
        #10 write_reg = 5; write_data = 32'hDEADBEEF; RegWrite = 1;
        #10 RegWrite = 0;
        #10 read_reg1 = 5; read_reg2 = 0;  // Should read DEADBEEF from reg 5, 0 from reg 0

        // Test 2: Write to register 10 and read from both 5 and 10
        #10 write_reg = 10; write_data = 32'h12345678; RegWrite = 1;
        #10 RegWrite = 0;
        #10 read_reg1 = 5; read_reg2 = 10;  // Should read DEADBEEF from reg 5, 12345678
from reg 10

        // Test 3: Write to register 0 (should stay 0) and read
        #10 write_reg = 0; write_data = 32'hFFFFFFFF; RegWrite = 1;
        #10 RegWrite = 0;
        #10 read_reg1 = 0; read_reg2 = 10;  // Should read 0 from reg 0, 12345678 from reg 10

        // Test 4: Reset and read
        #10 reset = 1;
        #10 reset = 0;
```

```
    #10 read_reg1 = 5; read_reg2 = 10;  // Should read 0 from both after reset

    #20 $finish;  // End simulation
  end
endmodule
```

DSD Lab 7(PIPELINING)

```
module pipelined_multiplier_4x4 (
  input clk,
  input rst,
  input [3:0] A, B,
  output reg [7:0] P
);

reg [3:0] A_reg, B_reg;
reg [7:0] partial_sum1, partial_sum2;

// Stage 0: Latch inputs
always @(posedge clk or posedge rst) begin
  if (rst) begin
    A_reg <= 0;
    B_reg <= 0;
  end else begin
    A_reg <= A;
    B_reg <= B;
  end
end

// Generate partial products
```

```
wire [7:0] pp[3:0];
assign pp[0] = A_reg[0] ? {4'b0, B_reg} : 8'b0;
assign pp[1] = A_reg[1] ? {3'b0, B_reg, 1'b0} : 8'b0;
assign pp[2] = A_reg[2] ? {2'b0, B_reg, 2'b0} : 8'b0;
assign pp[3] = A_reg[3] ? {1'b0, B_reg, 3'b0} : 8'b0;

// Stage 1: Add first two and last two partial products
always @(posedge clk or posedge rst) begin
   if (rst)
      partial_sum1 <= 0;
   else
      partial_sum1 <= pp[0] + pp[1];
end

always @(posedge clk or posedge rst) begin
   if (rst)
      partial_sum2 <= 0;
   else
      partial_sum2 <= pp[2] + pp[3];
end

// Stage 2: Final sum
always @(posedge clk or posedge rst) begin
   if (rst)
      P <= 0;
   else
      P <= partial_sum1 + partial_sum2;
end

endmodule
```

```
`timescale 1ns/1ps
module tb_pipelined_multiplier_4x4;

reg clk, rst;
reg [3:0] A, B;
wire [7:0] P;

pipelined_multiplier_4x4 uut (
    .clk(clk),
    .rst(rst),
    .A(A),
    .B(B),
    .P(P)
);

always #5 clk = ~clk;

initial begin
    clk = 0;
    rst = 1;
    A = 0;
    B = 0;

    $monitor("Time = %0t | A = %d, B = %d => P = %d", $time, A, B, P);

    #10 rst = 0;
    #10 A = 4'd3; B = 4'd5;
    #10 A = 4'd15; B = 4'd15;
    #10 A = 4'd7; B = 4'd2;
    #10 A = 4'd0; B = 4'd12;
```

```
    #50 $stop;
end

endmodule
```

```
module pipelined_4bit_adder_2stage (
    input  logic        clk,
    input  logic        rst,
    input  logic [3:0]  A,
    input  logic [3:0]  B,
    output logic [3:0]  SUM,
    output logic        COUT
);

    // First stage outputs
    logic s0, s1, c1;

    // Pipeline registers (between stage 1 and 2)
    logic [1:0] sum_stage1;
    logic       carry_stage1;
    logic [1:0] a_high, b_high;

    // Stage 1: Add bit 0 and bit 1
    always_ff @(posedge clk or posedge rst) begin
        if (rst) begin
            sum_stage1    <= 2'b00;
            carry_stage1  <= 1'b0;
            a_high        <= 2'b00;
```

```
        b_high       <= 2'b00;
      end else begin
        // Full Adder for bit 0
        s0 = A[0] ^ B[0];
        logic c0 = A[0] & B[0];

        // Full Adder for bit 1
        s1 = A[1] ^ B[1] ^ c0;
        c1 = (A[1] & B[1]) | (A[1] & c0) | (B[1] & c0);

        // Store sum and carry in pipeline registers
        sum_stage1    <= {s1, s0};
        carry_stage1  <= c1;
        a_high       <= A[3:2];
        b_high       <= B[3:2];
      end
    end

    // Stage 2: Add bit 2 and bit 3
    logic s2, s3;
    logic c2, c3;

    always_ff @(posedge clk or posedge rst) begin
      if (rst) begin
        SUM  <= 4'b0000;
        COUT <= 1'b0;
      end else begin
        // Full Adder for bit 2
        s2 = a_high[0] ^ b_high[0] ^ carry_stage1;
        c2 = (a_high[0] & b_high[0]) | (a_high[0] & carry_stage1) | (b_high[0] &
carry_stage1);
```

```
        // Full Adder for bit 3
        s3 = a_high[1] ^ b_high[1] ^ c2;
        c3 = (a_high[1] & b_high[1]) | (a_high[1] & c2) | (b_high[1] & c2);

        // Output result
        SUM  <= {s3, s2, sum_stage1};
        COUT <= c3;
      end
    end

endmodule
```

```
module pipelined_4bit_ripple_carry_adder (
    input  logic        clk,
    input  logic        rst,
    input  logic [3:0]  A,
    input  logic [3:0]  B,
    output logic [3:0]  SUM,
    output logic        COUT
);

    // Stage 1 Registers
    logic s0, c0;
    logic a0, b0;

    // Stage 2 Registers
    logic s1, c1;
    logic a1, b1;
```

```
// Stage 3 Registers
logic s2, c2;
logic a2, b2;

// Stage 4 Registers
logic s3;
logic a3, b3;
logic cout_reg;

// Register inputs
always_ff @(posedge clk or posedge rst) begin
  if (rst) begin
    {a0, b0, a1, b1, a2, b2, a3, b3} <= 8'b0;
  end else begin
    a0 <= A[0]; b0 <= B[0];
    a1 <= A[1]; b1 <= B[1];
    a2 <= A[2]; b2 <= B[2];
    a3 <= A[3]; b3 <= B[3];
  end
end

// Stage 1: Add LSBs
always_ff @(posedge clk or posedge rst) begin
  if (rst) begin
    s0 <= 0; c0 <= 0;
  end else begin
    s0 <= a0 ^ b0;
    c0 <= a0 & b0;
  end
end

// Stage 2
```

```
always_ff @(posedge clk or posedge rst) begin
   if (rst) begin
      s1 <= 0; c1 <= 0;
   end else begin
      s1 <= a1 ^ b1 ^ c0;
      c1 <= (a1 & b1) | (a1 & c0) | (b1 & c0);
   end
end

// Stage 3
always_ff @(posedge clk or posedge rst) begin
   if (rst) begin
      s2 <= 0; c2 <= 0;
   end else begin
      s2 <= a2 ^ b2 ^ c1;
      c2 <= (a2 & b2) | (a2 & c1) | (b2 & c1);
   end
end

// Stage 4
always_ff @(posedge clk or posedge rst) begin
   if (rst) begin
      s3 <= 0; cout_reg <= 0;
   end else begin
      s3 <= a3 ^ b3 ^ c2;
      cout_reg <= (a3 & b3) | (a3 & c2) | (b3 & c2);
   end
end

// Output
assign SUM  = {s3, s2, s1, s0};
assign COUT = cout_reg;
```

```
endmodule
```

FSM

```
module consec(z,clk,rst,in);
input clk,rst,in;
output reg z;
reg [3:0]cs,ns;

//(* fsm_encoding = "gray" *) reg [3:0] cs,ns;
always@(posedge clk) begin
if (rst) begin
cs <= 4'b0000;
end else begin
cs <= ns;
end
end

always@(*) begin
case(cs)

4'b0000:
begin
z=0;
if(in==0)
ns=4'b0101;
```

```
else
ns=4'b0001;
end

4'b0001:
begin
z=0;
if(in==0)
ns=4'b0101;
else
ns=4'b0010;
end

4'b0010:
begin
z=0;
if(in==0)
ns=4'b0101;
else
ns=4'b0011;
end

4'b0011:
begin
z=0;
if(in==0)
ns=4'b0101;
else
ns=4'b0100;
end

4'b0100:
```

```
begin
z=1;
if(in==0)
ns=4'b0101;
else begin
ns=4'b0100;
end
end

4'b0101:
begin
z=0;
if(in==0)
ns=4'b0110;
else
ns=4'b0001;
end

4'b0110:
begin
z=0;
if(in==0)
ns=4'b0111;
else
ns=4'b0001;
end

4'b0111:
begin
z=0;
if(in==0)
ns=4'b1000;
```

```
else
ns=4'b0001;
end

4'b1000:
begin
z=1;
if(in==0)begin
ns=4'b1000;
end
else
ns=4'b0001;
end

default:
begin
z=0;
ns=0;
end

endcase
end

endmodule
```

```
module task2(z,clk,rst,w);
input clk,rst,w;
output reg z;
```

```
reg [3:0]store,store2;

always@(posedge clk) begin
if (rst) begin
store <= 0;
store2<=0;
z <= 0;
end
else
store<= {store,w};
store2<= {store2,w};

if (store == 4'b0000 || store2 == 4'b1111) begin
z <= 1;
end
else
z <= 0;
end
endmodule
```

Zero_99

```
module zero_99(
   input clock,
   input reset,
   output a, b, c, d, e, f, g,
   output dp,
```

```
    output [7:0] an
);
    reg [3:0] first;  // Units digit (0-9)
    reg [3:0] second; // Tens digit (0-9)
    reg [23:0] delay; // For 0.1-second delay
    wire test;

    // Delay counter for 0.1-second updates
    always @(posedge clock or posedge reset) begin
        if (reset) delay <= 0;
        else delay <= delay + 1;
    end
    assign test = &delay; // High when delay overflows

    // Counter logic: Increment digits every 0.1 second
    always @(posedge test or posedge reset) begin
        if (reset) begin
            first <= 0;
            second <= 0;
        end else if (first == 4'd9) begin
            first <= 0;
            if (second == 4'd9) second <= 0;
            else second <= second + 1;
        end else first <= first + 1;
    end

    // Multiplexing for 4-digit display
    localparam N = 18;
    reg [N-1:0] count;
    always @(posedge clock or posedge reset) begin
        if (reset) count <= 0;
        else count <= count + 1;
```

```
end

reg [6:0] sseg;
reg [7:0] an_temp;
always @(*) begin
   case(count[N-1:N-2])
      2'b00: begin
         sseg = first;
         an_temp = 8'b11111110;
      end
      2'b01: begin
         sseg = second;
         an_temp = 8'b11111101;
      end
      2'b10: begin
         sseg = 6'ha;
         an_temp = 8'b11111011;
      end
      2'b11: begin
         sseg = 6'ha;
         an_temp = 8'b11110111;
      end
   endcase
end
assign an = an_temp;

// 7-segment display encoding
reg [6:0] sseg_temp;
always @(*) begin
   case(sseg)
      4'd0: sseg_temp = 7'b1000000;
      4'd1: sseg_temp = 7'b1111001;
```

```
        4'd2: sseg_temp = 7'b0100100;
        4'd3: sseg_temp = 7'b0110000;
        4'd4: sseg_temp = 7'b0011001;
        4'd5: sseg_temp = 7'b0010010;
        4'd6: sseg_temp = 7'b0000010;
        4'd7: sseg_temp = 7'b1111000;
        4'd8: sseg_temp = 7'b0000000;
        4'd9: sseg_temp = 7'b0010000;
        default: sseg_temp = 7'b0111111;
      endcase
    end
    assign {g, f, e, d, c, b, a} = sseg_temp;
    assign dp = 1'b1;
endmodule
```

Z_99_up_down

```
always_ff @(posedge clk or posedge rst)begin
        if(rst)
        begin
        f<=0;
        s<=0;
        end

        else if (up_enable)
        begin

        if(f==4'b9)
        begin
        f<=0;
        if(s==4'b9)
        s<=0;
        else
        s<=s+1;
        end
        else
```

```
        f<=f+1;
        end

        else if (down_enable)
        begin

        if(f==4'b0)
        begin
        f<=9;
        if(s==4'b0)
        s<=9;
        else
        s<=s-1;
        end
        else
        f<=f-1;
        end

        else begin
        s <= s;
        f <= f;
        end
end
```

Digital_CLocl_CODE

```
module segments(
  input [5:0] number,       // 6-bit input (0 to 30)
  output reg [13:0] segments  // 14-bit output for two 7-segment displays
);
  always @*
    case(number)
      6'd00: segments = 14'b0111111_0111111; // 0
      6'd01: segments = 14'b0111111_0000110; // 1
      6'd02: segments = 14'b0111111_1011011; // 2
      6'd03: segments = 14'b0111111_1001111; // 3
      6'd04: segments = 14'b0111111_1100110; // 4
      6'd05: segments = 14'b0111111_1101101; // 5
      6'd06: segments = 14'b0111111_1111101; // 6
      6'd07: segments = 14'b0111111_0000111; // 7
```

```
        6'd08: segments = 14'b0111111_1111111; // 8
        6'd09: segments = 14'b0111111_1101111; // 9
        6'd10: segments = 14'b0000110_0111111; // 10
        6'd11: segments = 14'b0000110_0000110; // 11
        6'd12: segments = 14'b0000110_1011011; // 12
        6'd13: segments = 14'b0000110_1001111; // 13
        6'd14: segments = 14'b0000110_1100110; // 14
        6'd15: segments = 14'b0000110_1101101; // 15
        6'd16: segments = 14'b0000110_1111101; // 16
        6'd17: segments = 14'b0000110_0000111; // 17
        6'd18: segments = 14'b0000110_1111111; // 18
        6'd19: segments = 14'b0000110_1101111; // 19
        6'd20: segments = 14'b1011011_0111111; // 20
        6'd21: segments = 14'b1011011_0000110; // 21
        6'd22: segments = 14'b1011011_1011011; // 22
        6'd23: segments = 14'b1011011_1001111; // 23
        6'd24: segments = 14'b1011011_1100110; // 24
        6'd25: segments = 14'b1011011_1101101; // 25
        6'd26: segments = 14'b1011011_1111101; // 26
        6'd27: segments = 14'b1011011_0000111; // 27
        6'd28: segments = 14'b1011011_1111111; // 28
        6'd29: segments = 14'b1011011_1101111; // 29
        6'd30: segments = 14'b1001111_0111111; // 30
    endcase
endmodule


module clock(
    input clk,              // 100 MHz FPGA clock
    input reset,            // Resets the clock
    output reg [6:0] segments,
    output reg [7:0] anodes   // Controls active digit
);
    reg [32:0] count;  // Clock divider counter
    reg clr_count;

    reg [4:0] sec;     // Seconds (0-30, 5 bits)
    reg clr_sec;

    reg [2:0] min;     // Minutes (0-7, 3 bits)
    reg clr_min;

    reg [2:0] hrs;     // Hours (0-7, 3 bits)
    reg clr_hrs;

    wire [6:0] sec_mss, sec_lss;  // Tens and ones of seconds
    wire [6:0] mins_mss, mins_lss; // Tens and ones of minutes
```

```
wire [6:0] hrs_mss, hrs_lss;   // Tens and ones of hours

reg [31:0] seg_count;  // Multiplexing counter

// Clock divider
always @(posedge clk)
   if (reset || clr_count) count <= #1 0;
   else count <= #1 count + 1;
always @* clr_count = (count == 33'd99_999_999);  // ~1 second

// Seconds counter
always @(posedge clk)
   if (reset || clr_sec) sec <= #1 0;
   else if (clr_count) sec <= #1 sec + 1;
always @* clr_sec = clr_count & (sec == 6'd30);  // Reset at 30 seconds

// Minutes counter
always @(posedge clk)
   if (reset || clr_min) min <= #1 0;
   else if (clr_sec) min <= #1 min + 1;
always @* clr_min = clr_sec & (min == 5'd2);  // Reset at 2 minutes

// Hours counter
always @(posedge clk)
   if (reset || clr_hrs) hrs <= #1 0;
   else if (clr_min) hrs <= #1 hrs + 1;
always @* clr_hrs = clr_sec & (min == 5'd2) & (hrs == 5'd2);  // Reset at 2 hours

// Segment generation
segments sec_seg (.number({1'b0, sec}), .segments({sec_mss, sec_lss}));
segments mins_seg (.number({3'b0, min}), .segments({mins_mss, mins_lss}));
segments hrs_seg (.number({3'b0, hrs}), .segments({hrs_mss, hrs_lss}));

// Multiplexing counter
always @(posedge clk)
   if (reset) seg_count <= #1 0;
   else seg_count <= #1 seg_count + 1;

// Segment selection
always @*
   case (seg_count[19:17])
      3'd0: segments = ~sec_lss;   // Ones of seconds
      3'd1: segments = ~sec_mss;   // Tens of seconds
      3'd2: segments = ~mins_lss;  // Ones of minutes
      3'd3: segments = ~mins_mss;  // Tens of minutes
      3'd4: segments = ~hrs_lss;   // Ones of hours
      3'd5: segments = ~hrs_mss;   // Tens of hours
```

```
            default: segments = 0;
        endcase

    // Anode control
    always @*
        case (seg_count[19:17])
            3'd0: anodes = 8'b11111110;  // Digit 0
            3'd1: anodes = 8'b11111101;  // Digit 1
            3'd2: anodes = 8'b11111011;  // Digit 2
            3'd3: anodes = 8'b11110111;  // Digit 3
            3'd4: anodes = 8'b11101111;  // Digit 4
            3'd5: anodes = 8'b11011111;  // Digit 5
            default: anodes = 8'b11111111;
        endcase
endmodule
```

OPT_CLocK_CODE

```
`timescale 1ns / 1ps


module clock_2(
    input clk,              // Main clock signal
    input reset,            // Reset signal
    output reg [6:0] ssd,    // Seven-segment display output
    output reg [7:0] anode    // Anode output for multiplexing
);

    reg [25:0] count;        // Counter for generating 1Hz clock
    reg clk_1hz;            // 1Hz clock signal
    reg [3:0] sec_lss;       // Least significant second digit
    reg [3:0] sec_mss;       // Most significant second digit
    reg [3:0] min_lss;       // Least significant minute digit
    reg [3:0] min_mss;       // Most significant minute digit
    reg [3:0] hr_lss;        // Least significant hour digit
    reg [3:0] hr_mss;        // Most significant hour digit
    reg [19:0] seg_count;     // Counter for segment multiplexing
    reg [3:0] number;        // Digit to be displayed

    // Generate 1Hz clock from 100MHz clock
```

```
always @(posedge clk or posedge reset) begin
   if (reset) begin
      count <= 0;
      clk_1hz <= 0;
   end else if (count == 50_000_000) begin
      clk_1hz <= ~clk_1hz;
      count <= 0;
   end else begin
      count <= count + 1;
   end
end

// Clock logic for seconds, minutes, and hours
always @(posedge clk_1hz or posedge reset) begin
   if (reset) begin
      sec_lss <= 0;
      sec_mss <= 0;
      min_lss <= 0;
      min_mss <= 0;
      hr_lss <= 0;
      hr_mss <= 0;
   end else begin
      if (sec_lss == 9) begin
         sec_lss <= 0;
         if (sec_mss == 5) begin
            sec_mss <= 0;
            if (min_lss == 9) begin
               min_lss <= 0;
               if (min_mss == 5) begin
                  min_mss <= 0;
                  if (hr_lss == 3 && hr_mss == 2) begin
                     hr_lss <= 0;
                     hr_mss <= 0;
                  end else if (hr_lss == 9) begin
                     hr_lss <= 0;
                     hr_mss <= hr_mss + 1;
                  end else begin
                     hr_lss <= hr_lss + 1;
                  end
               end else begin
                  min_mss <= min_mss + 1;
               end
            end else begin
               min_lss <= min_lss + 1;
            end
         end else begin
            sec_mss <= sec_mss + 1;
```

```
        end
      end else begin
        sec_lss <= sec_lss + 1;
      end
    end
end

// Multiplexing logic for seven-segment display
always @(posedge clk) begin
    seg_count <= seg_count + 1;
end

always @(*) begin
    case (seg_count[19:17])
        3'd0: begin
            number = sec_lss;  // Display least significant second digit
            anode = 8'b11111110;
        end
        3'd1: begin
            number = sec_mss;  // Display most significant second digit
            anode = 8'b11111101;
        end
        3'd2: begin
            number = min_lss;  // Display least significant minute digit
            anode = 8'b11111011;
        end
        3'd3: begin
            number = min_mss;  // Display most significant minute digit
            anode = 8'b11110111;
        end
        3'd4: begin
            number = hr_lss;   // Display least significant hour digit
            anode = 8'b11101111;
        end
        3'd5: begin
            number = hr_mss;   // Display most significant hour digit
            anode = 8'b11011111;
        end
        default: begin
            number = 4'b1111;  // Turn off all segments
            anode = 8'b11111111;
        end
    endcase
end

// Seven-segment display encoding
always @(*) begin
```

```
        case (number)
            4'd0: ssd = 7'b1000000; // 0
            4'd1: ssd = 7'b1111001; // 1
            4'd2: ssd = 7'b0100100; // 2
            4'd3: ssd = 7'b0110000; // 3
            4'd4: ssd = 7'b0011001; // 4
            4'd5: ssd = 7'b0010010; // 5
            4'd6: ssd = 7'b0000010; // 6
            4'd7: ssd = 7'b1111000; // 7
            4'd8: ssd = 7'b0000000; // 8
            4'd9: ssd = 7'b0010000; // 9
            default: ssd = 7'b1111111; // Turn off all segments
        endcase
    end

endmodule
```

```
wsl –install
```

```
curl -fsSL https://ollama.com/install.sh | sh
```

```
ollama pull llama3.2:1b
```

```
ollama run llama3.2:1b "What is the capital of France?"
```