MICROPROCESSOR LAB ASSIGNMENT

1. Design a 4-bit equality comparator circuit that has two 4-bit binary inputs (A and B) and outputs a logic-1 if both inputs are equal?

```
module equality_comparator(
  input [3:0] A,    // First 4-bit input
  input [3:0] B,    // Second 4-bit input
  output reg equal  // Output: 1 if A equals B, 0 otherwise
);

  // Compare the two 4-bit numbers and set the output
  always @(*) begin
    if (A == B)
      equal = 1'b1;
    else
      equal = 1'b0;
  end

endmodule

// Testbench
module equality_comparator_tb;
  reg [3:0] A;
  reg [3:0] B;
  wire equal;

  // Instantiate the comparator
  equality_comparator uut (
```

```
    .A(A),
    .B(B),
    .equal(equal)
);

// Test stimulus
initial begin
    // Test case 1: Equal values
    A = 4'b0000; B = 4'b0000;
    #10;  //this #10 is delay

    // Test case 2: Different values
    A = 4'b0101; B = 4'b1010;
    #10;

    // Test case 3: Equal values
    A = 4'b1111; B = 4'b1111;
    #10;

    // Test case 4: Different values
    A = 4'b1100; B = 4'b1101;
    #10;

    $finish;
end

// Optional: Monitor changes
```

```
    initial begin
        $monitor("Time=%0t A=%b B=%b equal=%b", $time, A, B, equal);
    end

endmodule
```

2. Design a 4-bit adder?

```
        // 1-bit Full Adder
module full_adder(
    input a,
    input b,
    input cin,
    output sum,
    output cout
);
    assign sum = a ^ b ^ cin;
    assign cout = (a & b) | (cin & (a ^ b));
endmodule

// 4-bit Adder
module four_bit_adder(
    input [3:0] a,
    input [3:0] b,
    input cin,
    output [3:0] sum,
    output cout
);
    wire c1, c2, c3;  // Internal carry wires

    // Instantiate four 1-bit full adders
    full_adder fa0(
        .a(a[0]),
        .b(b[0]),
        .cin(cin),
        .sum(sum[0]),
        .cout(c1)
    );
```

```
    full_adder fa1(
        .a(a[1]),
        .b(b[1]),
        .cin(c1),
        .sum(sum[1]),
        .cout(c2)
    );

    full_adder fa2(
        .a(a[2]),
        .b(b[2]),
        .cin(c2),
        .sum(sum[2]),
        .cout(c3)
    );

    full_adder fa3(
        .a(a[3]),
        .b(b[3]),
        .cin(c3),
        .sum(sum[3]),
        .cout(cout)
    );
endmodule

// Testbench
module four_bit_adder_tb;
    reg [3:0] a;
    reg [3:0] b;
    reg cin;
    wire [3:0] sum;
    wire cout;

    // Instantiate the 4-bit adder
    four_bit_adder uut(
        .a(a),
        .b(b),
        .cin(cin),
        .sum(sum),
        .cout(cout)
```

```
    );

    // Test stimulus
    initial begin
        // Test case 1: Simple addition
        a = 4'b0001; b = 4'b0001; cin = 0;
        #10;

        // Test case 2: Addition with carry
        a = 4'b0011; b = 4'b0001; cin = 0;
        #10;

        // Test case 3: Addition resulting in carry out
        a = 4'b1111; b = 4'b0001; cin = 0;
        #10;

        // Test case 4: Addition with carry in
        a = 4'b0111; b = 4'b0001; cin = 1;
        #10;

        // Test case 5: Maximum value addition
        a = 4'b1111; b = 4'b1111; cin = 0;
        #10;

        $finish;
    end

    // Monitor changes
    initial begin
        $monitor("Time=%0d a=%b b=%b cin=%b sum=%b cout=%b",
                 $time, a, b, cin, sum, cout);
    end
endmodule
```

3. Design the circuits for multiplication?

```
// Module for 4-bit multiplier
module multiplier_4bit(
    input [3:0] a,        // First input
    input [3:0] b,        // Second input
```

```
    output reg [7:0] prod   // Product output (8 bits to hold full result)
);

    // Behavioral description of multiplication
    always @(*) begin
        prod = a * b;    // Direct multiplication
    end

endmodule

// Testbench for verification
module multiplier_4bit_tb;
    // Test signals
    reg [3:0] a;
    reg [3:0] b;
    wire [7:0] prod;

    // Instantiate the multiplier
    multiplier_4bit uut (
        .a(a),
        .b(b),
        .prod(prod)
    );

    // Test stimulus
    initial begin
        // Display header
        $display("Time\tA\tB\tProduct");
        $display("----\t-\t-\t-------");

        // Test case 1: 2 x 3
        a = 4'd2; b = 4'd3;
        #10;
        $display("%0d\t%0d\t%0d\t%0d", $time, a, b, prod);

        // Test case 2: 5 x 4
        a = 4'd5; b = 4'd4;
        #10;
        $display("%0d\t%0d\t%0d\t%0d", $time, a, b, prod);

        // Test case 3: 15 x 15 (maximum value)
```

```
        a = 4'd15; b = 4'd15;
        #10;
        $display("%0d\t%0d\t%0d\t%0d", $time, a, b, prod);

        // Test case 4: 0 x 8
        a = 4'd0; b = 4'd8;
        #10;
        $display("%0d\t%0d\t%0d\t%0d", $time, a, b, prod);

        $finish;
    end

endmodule
```

4. Design the circuits for division?

```
// Module for 8-bit divider
module divider(
    input wire clk,
    input wire rst,
    input wire start,
    input wire [7:0] dividend,   // Number to be divided
    input wire [7:0] divisor,    // Number to divide by
    output reg [7:0] quotient,   // Result of division
    output reg [7:0] remainder,  // Remainder
    output reg busy,             // Indicates calculation in progress
    output reg valid,            // Indicates result is valid
    output reg error             // Division by zero error
);

    // State definitions
    localparam IDLE = 2'b00;
    localparam CALC = 2'b01;
    localparam DONE = 2'b10;

    reg [1:0] state, next_state;
    reg [3:0] count;             // Counter for division steps
    reg [15:0] acc;              // Accumulator for division operation
    reg [7:0] divisor_reg;       // Store divisor
```

```
// State and output logic
always @(posedge clk or posedge rst) begin
   if (rst) begin
      state <= IDLE;
      busy <= 0;
      valid <= 0;
      error <= 0;
      quotient <= 0;
      remainder <= 0;
      count <= 0;
      acc <= 0;
      divisor_reg <= 0;
   end
   else begin
      case (state)
         IDLE: begin
            if (start) begin
               if (divisor == 0) begin
                  state <= DONE;
                  error <= 1;
                  valid <= 0;
               end
               else begin
                  state <= CALC;
                  busy <= 1;
                  valid <= 0;
                  error <= 0;
                  acc <= {8'b0, dividend};
                  divisor_reg <= divisor;
                  count <= 8;  // 8 bits to process
               end
            end
         end

         CALC: begin
            // Perform restoring division algorithm
            if (count > 0) begin
               // Shift left
               acc <= acc << 1;

               // Try to subtract
```

```
                if (acc[15:8] >= divisor_reg) begin
                    acc[15:8] <= acc[15:8] - divisor_reg;
                    acc[0] <= 1;
                end

                count <= count - 1;
            end
            else begin
                state <= DONE;
            end
        end

        DONE: begin
            busy <= 0;
            valid <= ~error;
            quotient <= acc[7:0];
            remainder <= acc[15:8];
            state <= IDLE;
        end

        default: state <= IDLE;
      endcase
    end
  end

endmodule

// Testbench
module divider_tb;
  reg clk, rst, start;
  reg [7:0] dividend, divisor;
  wire [7:0] quotient, remainder;
  wire busy, valid, error;

  // Clock generation
  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end

  // Instantiate divider
```

```
divider uut(
    .clk(clk),
    .rst(rst),
    .start(start),
    .dividend(dividend),
    .divisor(divisor),
    .quotient(quotient),
    .remainder(remainder),
    .busy(busy),
    .valid(valid),
    .error(error)
);

// Test stimulus
initial begin
    // Initialize
    rst = 1;
    start = 0;
    dividend = 0;
    divisor = 0;
    #10;
    rst = 0;

    // Test case 1: 20 ÷ 4 = 5 remainder 0
    #10;
    dividend = 20;
    divisor = 4;
    start = 1;
    #10;
    start = 0;
    wait(valid);
    $display("Test 1: %d ÷ %d = %d remainder %d",
            dividend, divisor, quotient, remainder);

    // Test case 2: 25 ÷ 3 = 8 remainder 1
    #10;
    dividend = 25;
    divisor = 3;
    start = 1;
    #10;
    start = 0;
```

```
        wait(valid);
        $display("Test 2: %d ÷ %d = %d remainder %d",
                dividend, divisor, quotient, remainder);

        // Test case 3: Division by zero
        #10;
        dividend = 15;
        divisor = 0;
        start = 1;
        #10;
        start = 0;
        wait(~busy);
        if (error)
          $display("Test 3: Division by zero detected correctly");

        // Test case 4: 255 ÷ 16 = 15 remainder 15
        #10;
        dividend = 255;
        divisor = 16;
        start = 1;
        #10;
        start = 0;
        wait(valid);
        $display("Test 4: %d ÷ %d = %d remainder %d",
                dividend, divisor, quotient, remainder);

        #50;
        $finish;
    end

    // Optional: Monitor changes
    initial begin
        $monitor("Time=%0t rst=%b start=%b busy=%b valid=%b error=%b",
                $time, rst, start, busy, valid, error);
    end

endmodule
```

5. Implement an Arithmetic Logic Unit (ALU).

Required: Develop an ALU that takes two 8-bit inputs A and B, and executes the

following six instructions: **add, sub, and, or, xor, nor**.The ALU generates an 8-bit output that we call ‘Result’ and an additional 1-bit flag
‘Zero’ that will be set to ‘logic-1’ if all the bits of ‘Result’ are 0?

```
// 8-bit ALU with 6 operations (add, sub, and, or, xor, nor)
module alu(
   input [7:0] A,         // 8-bit input A
   input [7:0] B,         // 8-bit input B
   input [2:0] opcode,    // Operation selection
   output reg [7:0] Result, // 8-bit output
   output Zero            // Zero flag
);

   // Operation codes
   parameter ADD = 3'b000;
   parameter SUB = 3'b001;
   parameter AND = 3'b010;
   parameter OR  = 3'b011;
   parameter XOR = 3'b100;
   parameter NOR = 3'b101;

   // Zero flag assignment
   assign Zero = (Result == 8'b0) ? 1'b1 : 1'b0;

   // ALU operation logic
   always @(*) begin
      case (opcode)
         ADD: Result = A + B;
         SUB: Result = A - B;
         AND: Result = A & B;
         OR:  Result = A | B;
         XOR: Result = A ^ B;
         NOR: Result = ~(A | B);
         default: Result = 8'b0;
      endcase
   end

endmodule

// Testbench for verification
```

```
module alu_tb;
    reg [7:0] A, B;
    reg [2:0] opcode;
    wire [7:0] Result;
    wire Zero;

    // Instantiate the ALU
    alu uut(
        .A(A),
        .B(B),
        .opcode(opcode),
        .Result(Result),
        .Zero(Zero)
    );

    // Test stimulus
    initial begin
        // Test case 1: Addition
        A = 8'h05; B = 8'h03; opcode = 3'b000;
        #10;

        // Test case 2: Subtraction
        A = 8'h05; B = 8'h03; opcode = 3'b001;
        #10;

        // Test case 3: AND
        A = 8'hFF; B = 8'h0F; opcode = 3'b010;
        #10;

        // Test case 4: OR
        A = 8'hF0; B = 8'h0F; opcode = 3'b011;
        #10;

        // Test case 5: XOR
        A = 8'hFF; B = 8'hF0; opcode = 3'b100;
        #10;

        // Test case 6: NOR
        A = 8'hF0; B = 8'h0F; opcode = 3'b101;
        #10;
```

```
        // Test case 7: Zero flag test
        A = 8'h00; B = 8'h00; opcode = 3'b000;
        #10;

        $finish;
    end

    // Monitor changes
    initial begin
        $monitor("Time=%0d A=%h B=%h opcode=%b Result=%h Zero=%b",
                 $time, A, B, opcode, Result, Zero);
    end

endmodule
```

6. Design an in-order processor, in which the main execution unit , that is , ALU is at least able to perform functions listed in Part 2 of the assignment.

```
// 8-bit ALU with 6 operations (add, sub, and, or, xor, nor)
module alu(
    input [7:0] A,        // 8-bit input A
    input [7:0] B,        // 8-bit input B
    input [2:0] opcode,   // Operation selection
    output reg [7:0] Result, // 8-bit output
    output Zero           // Zero flag
);

    // Operation codes
    parameter ADD = 3'b000;
    parameter SUB = 3'b001;
    parameter AND = 3'b010;
    parameter OR  = 3'b011;
    parameter XOR = 3'b100;
```

```
    parameter NOR = 3'b101;

    // Zero flag assignment
    assign Zero = (Result == 8'b0) ? 1'b1 : 1'b0;

    // ALU operation logic
    always @(*) begin
        case (opcode)
            ADD: Result = A + B;
            SUB: Result = A - B;
            AND: Result = A & B;
            OR:  Result = A | B;
            XOR: Result = A ^ B;
            NOR: Result = ~(A | B);
            default: Result = 8'b0;
        endcase
    end

endmodule

// Testbench for verification
module alu_tb;
    reg [7:0] A, B;
    reg [2:0] opcode;
    wire [7:0] Result;
    wire Zero;
```

```
// Instantiate the ALU
alu uut(
    .A(A),
    .B(B),
    .opcode(opcode),
    .Result(Result),
    .Zero(Zero)
);

// Test stimulus
initial begin
    // Test case 1: Addition
    A = 8'h05; B = 8'h03; opcode = 3'b000;
    #10;

    // Test case 2: Subtraction
    A = 8'h05; B = 8'h03; opcode = 3'b001;
    #10;

    // Test case 3: AND
    A = 8'hFF; B = 8'h0F; opcode = 3'b010;
    #10;

    // Test case 4: OR
    A = 8'hF0; B = 8'h0F; opcode = 3'b011;
    #10;
```

```
        // Test case 5: XOR
        A = 8'hFF; B = 8'hF0; opcode = 3'b100;
        #10;

        // Test case 6: NOR
        A = 8'hF0; B = 8'h0F; opcode = 3'b101;
        #10;

        // Test case 7: Zero flag test
        A = 8'h00; B = 8'h00; opcode = 3'b000;
        #10;

        $finish;
    end

    // Monitor changes
    initial begin
        $monitor("Time=%0d A=%h B=%h opcode=%b Result=%h Zero=%b",
                 $time, A, B, opcode, Result, Zero);
    end

endmodule
```