PROBLEM 1

```
// 2:1 MUX
module mux2_1(InA,InB,S,Out);
    // define inputs and outputs
    input InA,InB,S;
    output Out;
    // wires local to the module
    wire a1,a2,n1;

    // logic starts
    not1 n5(S,n1);
    nand2 n4(InA,n1,a1);
    nand2 n2(InB,S,a2);
    nand2 n3(a1,a2,out);

endmodule
// end of module

// 4:1 MUX using 2:1 MUX
module mux4_1 (InA,InB,InC,InD,S,Out);
    // define inputs and outputs
    input InA,InB,InC,InD;
    input [1:0] S;
    output Out;
    // wires local to module
    wire s1,s2;

    // logic starts
    mux2_1 m1 (.InA(InA),.InB(InB),.S(S[0]),.Out(s1));
    mux2_1 m2 (.InA(InC),.InB(InD),.S(S[0]),.Out(s2));
    mux2_1 m3 (.InA(s1),.InB(s2),.S(S[1]),.Out(out));

endmodule
// end of module

// quad mux
module quadmux4_1 (InA,InB,InC,InD,S,Out);
    // define inputs and outputs
    input [3:0] InA,InB,InC,InD;
    input [1:0] S;
    output [3:0] Out;

    // Logic starts
    mux4_1 m1(.InA(InA[3]),.InB(InB[3]),.InC(InC[3]),.InD(InD[3]),.S(S),.Out(Out[3]));
    mux4_1 m2(.InA(InA[2]),.InB(InB[2]),.InC(InC[2]),.InD(InD[2]),.S(S),.Out(Out[2]));
    mux4_1 m3(.InA(InA[1]),.InB(InB[1]),.InC(InC[1]),.InD(InD[1]),.S(S),.Out(Out[1]));
    mux4_1 m4(.InA(InA[0]),.InB(InB[0]),.InC(InC[0]),.InD(InD[0]),.S(S),.Out(Out[0]));

endmodule
// end of module
```

PROBLEM 2

```
// 1 bit full adder
module fulladder(A,B,Cin,Cout,S);
    // define inputs and outputs
    input Cin,A,B;
    output S,Cout;
    // define wires that are local to the module
    wire a1,a2,a3;

    // logic starts
    xor3 x1(.in1(A),.in2(B),.in3(Cin),.out(S));
    xor2 x2(.in1(A),.in2(B),.out(a1));
    nand2 n1(.in1(A),.in2(B),.out(a2));
    nand2 n2(.in1(a1),.in2(Cin),.out(a3));
    nand2 n3(.in1(a3),.in2(a2),.out(Cout));

endmodule
// end module

// 4-bit ripple carry adder
module rippleadder(A,B,SUM,CO,CI);
    // define inputs and outputs
    input [3:0] A,B;
    input CI;
    output CO;
    output [3:0] SUM;
    // define wires that re local to the module
    wire a1,a2,a3;

    // logic starts
    fulladder f1(.A(A[0]),.B(B[0]),.Cin(CI),.Cout(a1),.S(SUM[0]));
    fulladder f2(.A(A[1]),.B(B[1]),.Cin(a1),.Cout(a2),.S(SUM[1]));
    fulladder f3(.A(A[2]),.B(B[2]),.Cin(a2),.Cout(a3),.S(SUM[2]));
    fulladder f4(.A(A[3]),.B(B[3]),.Cin(a3),.Cout(CO),.S(SUM[3]));

endmodule
// end of module

// 16 bit ripple carry adder
module fulladder16(A,B,CI,SUM,CO);
    // define inputs and outputs
    input [15:0] A,B;
    output [15:0] SUM;
    input CI;
    output CO;
    // define wires that are local to the module
    wire [2:0] S;

    // logic starts
    rippleadder R1(.A(A[3:0]),.B(B[3:0]),.SUM(SUM[3:0]),.CI(CI),.CO(S[0]));
    rippleadder R2(.A(A[7:4]),.B(B[7:4]),.SUM(SUM[7:4]),.CI(S[0]),.CO(S[1]));
    rippleadder R3(.A(A[11:8]),.B(B[11:8]),.SUM(SUM[11:8]),.CI(S[1]),.CO(S[2]));
    rippleadder R4(.A(A[15:12]),.B(B[15:12]),.SUM(SUM[15:12]),.CI(S[2]),.CO(CO));

endmodule
// end of module
```

```
// testbench for 16-bit adder
module tb_fulladder;

    // testbench has no inputs and outputs
    reg [15:0] A;
    reg [15:0] B;
    reg [16:0] Sumcalc;
    wire [15:0] SUM;
    wire        CO;
    reg     CI;
    reg         Clk;

    // this statement is required if you want to use command line simulation in verilog
    initial $dumpvars;

    // Instantiate your module
    fulladder16 DUT (.A(A[15:0]), .B(B[15:0]), .CI(CI), .SUM(SUM), .CO(CO));

    // Always block for generating clock.
    // I want to see outputs at negative clock edge -> not required for this problem
    always
            #10 Clk = ~ Clk;

    // Define initial values for A, B and CI
    initial
            begin
            A = 16'b0000_0000_0000_0000;
            B = 16'b0000_0000_0000_0000;
            CI = 1'b0;
            Clk = 1'b0;
            #400 $finish;
            end

    // Assign random values for A, B and CI inputs
    always@(posedge Clk)
            begin
            A[15:0] = $random;
            B[15:0] = $random;
            CI = $random;
            end

    // Sum should be A+B+CI; of course you cannot use this in your source code
    always@(negedge Clk)
            begin
            Sumcalc = A+B+CI;
            $display("A : %x, B%x, Sum %x", A, B, SUM);

            // if the sum and CO returned by your module does not match results; throw an error
            if (Sumcalc[15:0] != SUM) $display ("ERRORCHECK Sum error");
            if (Sumcalc[16] != CO) $display ("ERRORCHECK CO error");
            end

endmodule
// tb_fulladder
-------------------------------------------------------------------------------------------
```

PROBLEM 3

```
// Top level module for detecting 46 (0100 0110)
module seqdec_42(InA,Clk,Reset,Out);

    // inputs and outputs of the module
    input Clk,Reset,InA;
    output Out;
    // wires that are local to the module for tracking present state and next state
    wire [2:0] pres_state,next_state;

    // logic start -> combinational logic and sequential logic instantiated
    comb_logic c1(.in(InA),.present_state(pres_state),.next_state(next_state),.out(Out));
    seq_logic s1(.clk(Clk),.rst(Reset),.present_state(pres_state),.next_state(next_state));

endmodule
// end of module

// This module is the combinational logic part that takes care of the transitions
module comb_logic(in,present_state,next_state,out);

    input in;
    input [2:0]present_state;
    output [2:0]next_state;
    output out;

    reg [2:0] next_state;
    reg out;

    // transition logic start
    always@ (*)
    case ({present_state,in})
        // if present state is 000 and the input is 0/1, what is the next state and output
        4'b0000:begin next_state=3'b001; out=1'b0; end
        4'b0001:begin next_state=3'b000; out=1'b0; end
        // if present state is 001 and the input is 0/1, what is the next state and output
        4'b0010:begin next_state=3'b001; out=1'b0; end
        4'b0011:begin next_state=3'b010; out=1'b0; end
        // if present state is 010 and the input is 0/1, what is the next state and output
        4'b0100:begin next_state=3'b011; out=1'b0; end
        4'b0101:begin next_state=3'b000; out=1'b0; end
        // if present state is 011 and the input is 0/1, what is the next state and output
        4'b0110:begin next_state=3'b100; out=1'b0; end
        4'b0111:begin next_state=3'b010; out=1'b0; end
        // if present state is 100 and the input is 0/1, what is the next state and output
        4'b1000:begin next_state=3'b101; out=1'b0; end
        4'b1001:begin next_state=3'b010; out=1'b0; end
        // if present state is 101 and the input is 0/1, what is the next state and output
        4'b1010:begin next_state=3'b001; out=1'b0; end
        4'b1011:begin next_state=3'b110; out=1'b0; end
        // if present state is 110 and the input is 0/1, what is the next state and output
        4'b1100:begin next_state=3'b011; out=1'b0; end
        4'b1101:begin next_state=3'b111; out=1'b0; end
        // if present state is 111 and the input is 0/1, what is the next state and output
        4'b1110:begin next_state=3'b001; out=1'b1; end
        4'b1111:begin next_state=3'b000; out=1'b0; end
        // By default, my next_state is 000 and output is a 0
```

```
        default:begin next_state=3'b000; out=1'b0; end
    endcase

endmodule
// end of module

// sequential logic with D flip flops
module seq_logic(clk,rst,present_state,next_state);

    // D-flip flop inputs
    input clk,rst;
    input [2:0]next_state;
    output [2:0]present_state;

    // instantiate 3 D flip flops
    dff d1 [2:0] (.q(present_state),.d(next_state),.clk(clk),.rst(rst));

endmodule
// end of module

// A simple testbench for the sequence detector
module tb_seqdec_46;

    // Testbench will not have any inputs/outputs
    reg rst,in,clk;
    wire out;

    // Instatntiate your sequence detector here
    seqdec_46 DUT(.Clk(clk),.Reset(rst),.InA(in),.Out(out));

    // Monitor statement will monitor all signals in its argument and will be called when one
    of them changes
    // Here I am printing the simulation time, output value, input value, present state and
    next state
    initial $monitor("%t  out:%b  in:%b  rst:%b pres_state:%b  next_state:%b ",
    $time,out,in,rst,DUT.pres_state,DUT.next_state);
    // After 1000ns simulation time; call $finish simulation task to stop simulation
    initial #1000 $finish;

    // This initial block defines clock behavior
    // Start with clk = 0; at every 5th timestep clk is the inverse of its previous value
    // the result of this will be a periodic clock signal. An always block can be used to
    define clk as well
    initial
    begin
            clk=0;
            forever
            #5 clk= ~clk;
    end

    // Pass values to the input signal and see if sequence detector works
    initial
    begin
        rst=1'b1;
        in=1'b1;
        #10 rst=1'b0;
```

```
        #10 in=1'b0;
        #10 in=1'b1;
        #10 in=1'b0;
        #10 in=1'b0;
        #10 in=1'b0;
        #10 in=1'b1;
        #10 in=1'b1;
        #10 in=1'b0;
        #10 in=1'b1;
        #10 in=1'b0;
        #10 in=1'b0;
        #10 in=1'b0;
        #10 in=1'b1;
        #10 in=1'b1;
        #10 in=1'b0;
        #10 in=1'b1;
        #10 in=1'b0;
        #10 in=1'b0;
        #10 in=1'b0;
        #10 in=1'b1;
        #10 in=1'b0;
        #10 in=1'b0;
        #10 in=1'b0;
        #10 in=1'b1;
        #10 in=1'b1;
        #10 in=1'b0;
    end

endmodule
// end of testbench
-------------------------------------------------------------------------------------------------
```