DESIGN

```
module FullAdder(A, B, Cin, Sum, Cout);
input A, B, Cin;
output Sum, Cout;

wire w1, w2, w3;

xor U1 (Sum, A, B, Cin);
and U2 (w1, B, A);
and U3 (w2, A, Cin);
and U4 (w3, Cin, B);
or U5 (Cout, w1, w2, w3);
endmodule

module Mux4 (S, D, Y);
input [1:0] S; //select input that is 2-bit wide
input [3:0] D;
output Y;

wire w1, w2, w3, w4, w5, w6;

not U1 (w1, S[0]);
not U2 (w2, S[1]);
and U3 (w1, W2, w3, D[0]);
and U4 (w4, W2, S[0], D[1]);
and U5 (w5, w1, S[1], D[2]);
and U6 (W6, s1, S[0], D[3]);
or U7 (Y, w3, w4, w5, w6);

endmodule

module Majority3 (a,b,c,x);

input a,b,c;
output x;

wire w1,w2, w3;

and U1 (w1,a,b);
and U2 (w2,a,c);
and U3 (w3,b,c);
or U4 (x,w1,w2,w3);

endmodule

module EvenParity (data_in, data_out);
input [7:0] data_in;
output [8:0] data_out;
      assign data_out [8:1] =data_in;
      assign data_out[0] =^data_in; //store parity bit into LSB
endmodule
```

```
module FullAdder8 (A,B,Cin,S,Cout);
input [7:0] A,B;
input Cin;
output [7:0] S;
output Cout;
//add 2 8-bit numbers together along with Carry-in and store to 9 bit result of sum and carry-out
assign {Cout,S}=A+B+Cin;

endmodule

module Mux4_DF(S,D,Y);
input [3:0] D;
input [1:0] S;
output Y;

assign Y=(~S[1]) & (~S[0]) & (D[0]) |
         (~S[1]) & (S[0]) & (D[1]) |
         (S[1]) & (~S[0]) & (D[2]) |
         (S[1]) & (S[0]) & (D[3]);

endmodule

module SR4_NB(din, dout, clk); //4-bit shift register non-blocking
input din, clk;
output [3:0] dout;

reg [3:0] Q;

initial
begin
Q=4'b0; //intialize shift register to be clear
end

always @ (posedge clk) //model the shifting within the shift register
begin
    Q[0]<=din;//serial inputs
    Q[1]<=Q[0];
    Q[2]<=Q[1];
    Q[3]<=Q[2];
end

assign dout=Q; //serial output updates when Q[3] value changes

endmodule

module SR4_B(din, dout, clk); //4-bit shift register non-blocking
input din, clk;
output [3:0] dout;

reg [3:0] Q;
```

```
initial
begin
Q=4'b0; //intialize shift register to be clear
end

always @ (posedge clk) //model the shifting within the shift register
begin
	Q[0]=din;//serial inputs
	Q[1]=Q[0];
	Q[2]=Q[1];
	Q[3]=Q[2];
end

assign dout=Q; //serial output updates when Q[3] value changes

endmodule

module decoder_38( in_data, oeb, out_data);
input wire [2:0] in_data;
input oeb;
output [7:0] out_data;

//use for loop
reg [7:0] out_data;
integer cnt;

always @ (in_data,oeb)
begin
	if (oeb ==1'b0)//output is enabled
	begin
		for(cnt=0; cnt<8; cnt=cnt+1) //set the value for each bit of the output
		begin
			if(cnt==in_data) //check if current bit location equals to the input value
			begin
				out_data[cnt]=1'b1;
			end
			else
			begin
				out_data[cnt]=1'b0;
			end
		end
	end
end

////use continuous assignment
//assign out_data=(oeb==1'b0) ? (8'b0000_0001<<in_data):(8'b0000_0000);

//use case statement with procedural block
//reg [7:0] out_data;
//always @(in_data,oeb)
//begin
//	if(oeb==1'b0)
```

```
//      begin
//          case(in_data)
//              3'b000: out_data=8'b0000_0001;
//              3'b001: out_data=8'b0000_0010;
//              3'b010: out_data=8'b0000_0100;
//              3'b011: out_data=8'b0000_1000;
//              3'b100: out_data=8'b0001_0000;
//              3'b101: out_data=8'b0010_0000;
//              3'b110: out_data=8'b0100_0000;
//              3'b111: out_data=8'b1000_0000;
//              default:out_data=8'b0000_0000;
//          endcase
//      end
//      else
//      begin
//          out_data=8'b0000_0000;
//      end
//end
endmodule

module comp4 (A, B, iAltB, iAeqB, iAgtB, oAltB, oAeqB, oAgtB);
input [3:0] A,B;
input iAltB, iAeqB, iAgtB;
output reg oAltB, oAeqB, oAgtB;

always @ (A,B,iAltB, iAeqB, iAgtB)
begin
    //reset all output regs
    oAltB=1'b0;
    oAeqB=1'b0;
    oAgtB=1'b0;
    if(A>B)
    begin
        oAgtB=1'b1;
    end
    else if (A<B)
    begin
        oAltB=1'b1;
    end
    else if (A==B)
    begin
        //check cascading input
        case({iAgtB, iAltB, iAeqB})
            3'b100: oAgtB=1'b1;
            3'b010: oAltB=1'b1;
            3'b001: oAeqB=1'b1;
            3'b000: begin
                        oAgtB=1'b1;
                        oAltB=1'b1;
                    end
            default:
                    begin
```

```
                        oAgtB=1'b0;
                        oAltB=1'b0;
                        oAeqB=1'b0;
                    end
        endcase
    end

end
endmodule


module top(pos, bin, out, an);
input [1:0] pos;
input [7:0] bin;
output reg [6:0] out;
output reg [3:0] an;

wire [3:0] ones, tens, hunds;
wire [6:0] seg7ones, seg7tens, seg7hunds;

bin2BCD U1 (.bin(bin), .ones(ones), .tens(tens), .hunds(hunds));
Seg7 U2 (.BCD(ones),.out(seg7ones));
Seg7 U3 (.BCD(tens),.out(seg7tens));
Seg7 U4 (.BCD(hunds),.out(seg7hunds));

always @ (pos,seg7ones, seg7tens, seg7hunds)
begin
    //implement mux for 7 segment display
    case(pos)
        2'b00://ones
            begin
                out=seg7ones;
                an=4'b1110;
            end
        2'b01://tens
            begin
                out=seg7tens;
                an=4'b1101;
            end
        2'b10://hunds
            begin
                out=seg7hunds;
                an=4'b1011;
            end
        2'b11://thousands
            begin
                out=7'b1111111;
                an=4'b0111;
            end
      endcase

end
```

```
endmodule


module Bin8ToBCD(clk, bin, out, an);
input clk;
input [7:0] bin;
output reg [6:0] out;
output reg [3:0] an;

reg [19:0] count; //20-bit counter to divide 100 MHz clock frequency for selecting 7-segment display

wire [1:0] pos;
wire [3:0] ones, tens, hunds;
wire [6:0] seg7ones, seg7tens, seg7hunds;

bin2BCD U1 (.bin(bin), .ones(ones), .tens(tens), .hunds(hunds));
Seg7 U2 (.BCD(ones),.out(seg7ones));
Seg7 U3 (.BCD(tens),.out(seg7tens));
Seg7 U4 (.BCD(hunds),.out(seg7hunds));

always @(posedge clk) //create a 20-bit free running binary counter
begin
	count<=count+1;
end

assign pos=count[19:18]; //use upper 2-bit of counter for mux select input, too fast would not show clear and too slow would blink

always @ (pos,seg7ones, seg7tens, seg7hunds)
begin
	//implement mux for 7 segment display
	case(pos)
		2'b00://ones
			begin
				out=seg7ones;
				an=4'b1110;
			end
		2'b01://tens
			begin
				out=seg7tens;
				an=4'b1101;
			end
		2'b10://hunds
			begin
				out=seg7hunds;
				an=4'b1011;
			end
		2'b11://thousands
			begin
				out=7'b1111111;
				an=4'b0111;
			end
```

```
        endcase

end

endmodule

module bin2BCD(bin, ones, tens, hunds);
input [7:0] bin;
output [3:0] ones, tens, hunds;

reg [19:0] temp; //temporary register to hold 8-bit binary input and BCD output during shifting and add 3 process

integer cnt; //control variable for thr for loop

always @ (bin)
begin
    temp={12'b0, bin}; //concatinate 12 bits of 0 with binary input and store to temp register

    //perform the shift and add 3 algorithm 8 times
    for (cnt=0; cnt<8; cnt=cnt+1)
    begin
        temp = temp << 1;//shift temp register to left by 1
        if(cnt<7)//first shifts
        begin
            //test each BCD output >=5
            if(temp[11:8]>=5)//if ones digit >=5
            begin
                temp[11:8]=temp[11:8]+3;
            end
            if(temp[15:12]>=5)//if tens digit >=5
            begin
                temp[15:12]=temp[15:12]+3;
            end
            if(temp[19:16]>=5)//if hunds digit >=5
            begin
                temp[19:16]=temp[19:16]+3;
            end
        end
     end
end

assign ones=temp[11:8];
assign tens=temp[15:12];
assign ones=temp[19:16];

endmodule

module Seg7(BCD, out);
input [3:0] BCD;
output reg [6:0] out;

always @ (BCD)
```

```
begin
    case(BCD) //care about the lsb and msb
        4'b0000:out=7'b1000000;
        4'b0001:out=7'b1111001;
        4'b0010:out=7'b0100100;
        4'b0011:out=7'b0110000;
        4'b0100:out=7'b0011001;
        4'b0101:out=7'b0010010;
        4'b0110:out=7'b0000010;
        4'b0111:out=7'b1111000;
        4'b1000:out=7'b0000000;
        4'b1001:out=7'b0010000;
        default: out=7'b1111111; //turn off display if invalid number
        endcase
end

endmodule

module DFF (clk, D, clr, pre, Q, Qnot);// clr, pre are active high

input clk, D, clr, pre;
output reg Q;
output Qnot;

always @ (posedge clk, posedge clr, posedge pre)
begin
    //first look at asyschronous inputs
    if(clr==1'b1)
    begin
        Q<=1'b0;
    end
    else if (pre==1'b1)
    begin
        Q<=1'b1;
    end
    else
    begin
        Q<=D;
    end
end

assign Qnot=~Q;
endmodule


module shiftreg (data_in, clk, clr, load, sc, load_data, data_out);//clr and load are asynchronous inputs
input data_in, clk,clr, load, sc;
input [3:0] load_data;
output data_out;

reg [3:0] R;
```

```
always @ (posedge clk, posedge clr, posedge load)
begin
	if (clr==1'b1)
	begin
		R<=0;
	end
	else if(load==1'b1)
	begin
		R<=load_data;
	end
	else
	begin
		if(sc==1'b1)//shift right
		begin
			R<={data_in, R[3:1]};
		end
		else//shift left
		begin
			R<={R[2:0],data_in};
		end
	end
end

assign data_out=(sc==1'b1)?R[0]:R[3];

endmodule

module LFSR(clk, en, seed, seed_data, LFSR_data, Repeat);
parameter n=4; //default width of LFSR is 4-bit
input clk, en, seed;
input [n-1:0] seed_data;
output Repeat;
output [n-1:0] LFSR_data;

reg [n-1:0] LFSR;
reg XOR; //output from linear function

always @ (posedge clk, posedge en)
begin
	if(en==1'b1)//LFSR is enabled
	begin
		if(seed==1'b1)//seeding the LFSR
		begin
			LFSR<=seed_data;//parallel load the LFSR register
		end
		else//not sedding, just shifting
		begin
			LFSR<={LFSR[n-1:0],XOR};//shift LFSR with linear function (XOR) output
		end
	end
	else //LFSR is not enabled
	begin
```

```
        LFSR<=0;//clear LFSR when not enabled
    end
end

//create the linear function for feedback of XOR value
always @ *
begin
    case(n)
        3:XOR=LFSR[0]^LFSR[2];
        4:XOR=LFSR[0]^LFSR[3];
        5:XOR=LFSR[1]^LFSR[4];
        6:XOR=LFSR[0]^LFSR[5];
        7:XOR=LFSR[0]^LFSR[6];
        8:XOR=LFSR[1]^LFSR[2]^LFSR[3]^LFSR[7];
        9:XOR=LFSR[3]^LFSR[8];
        10:XOR=LFSR[2]^LFSR[9];
        11:XOR=LFSR[1]^LFSR[10];
        12:XOR=LFSR[0]^LFSR[3]^LFSR[5]^LFSR[11];
        13:XOR=LFSR[0]^LFSR[2]^LFSR[3]^LFSR[12];
        14:XOR=LFSR[0]^LFSR[2]^LFSR[4]^LFSR[13];
        15:XOR=LFSR[0]^LFSR[14];
        16:XOR=LFSR[1]^LFSR[2]^LFSR[4]^LFSR[15];
     endcase
end
assign LFSR_data=LFSR;
assign Repeat={LFSR==seed_data}?1'b1:1'b0;//set Repeat to 1 whenever LFSR register equals the initial seed data (repeats)
endmodule

module MooreEdgeDetect(clk, in, reset, f_e, r_e);
localparam RESET=0, LOW=1, HIGH=2, RISING=3, FALLING=4; //or change it to parameter and upon the module
input clk, in, reset;
output reg f_e, r_e;

reg [2:0] state, nextstate;
always @ (posedge clk)
begin
    if(reset==1'b1)
    begin
        state<=RESET;
    end
    else
    begin
        state<=nextstate;
    end
end

always @ (*)//next state and output functions
begin
    nextstate=state;//default set next state to current state
    r_e=1'b0;
    f_e=1'b0;
```

```
    case(state)
        RESET:begin
                    if (in==1'b0)
                    begin
                        nextstate<=LOW;
                    end
                    else
                    begin
                        nextstate<=HIGH;
                    end
                end
        LOW:begin
                    if (in==1'b1)
                    begin
                        nextstate<=RISING;
                    end
                end
        HIGH:begin
                    if (in==1'b0)
                    begin
                        nextstate<=FALLING;
                    end
                end
        RISING:begin
                    r_e=1'b1; //set the rising edge flag to 1
                    if (in==1'b0)
                    begin
                        nextstate<=FALLING;
                    end
                    else
                    begin
                        nextstate<=HIGH;
                    end
                end
        FALLING:begin
                    f_e=1'b1; //set the falling edge flag to 1
                    if (in==1'b0)
                    begin
                        nextstate<=LOW;
                    end
                    else
                    begin
                        nextstate<=RISING;
                    end
                end
        default:begin
                    nextstate=RESET;
                    end
    endcase
end
endmodule
```

```
module MealyEdgeDetect(clk, in, reset, f_e, r_e);
localparam RESET=0, LOW=1, HIGH=2; //or change it to parameter and upon the module
input clk, in, reset;
output reg f_e, r_e;

reg [1:0] state, nextstate;
always @ (posedge clk)
begin
	if(reset==1'b1)
	begin
		state<=RESET;
	end
	else
	begin
		state<=nextstate;
	end
end

always @ (*)//next state and output functions
begin
	nextstate=state;//default set next state to current state
	r_e=1'b0;
	f_e=1'b0;

	case(state)
		RESET:begin
				if (in==1'b0)
				begin
					nextstate<=LOW;
				end
				else
				begin
					nextstate<=HIGH;
				end
			end
		LOW:begin
				if (in==1'b1)
				begin
					nextstate<=HIGH;
					r_e=1'b1; //set rising edge flag
				end
			end
		HIGH:begin
				if (in==1'b0)
				begin
					nextstate<=LOW;
					f_e=1'b1;
				end
			end

		default:begin
				nextstate=RESET;
```

```
            end
endcase
```

Test

```
module Tester();//testbench to test the non-blocking SR4
reg din, clk;
wire [3:0] dout, dout2;
SR4_NB UUT (.din(din), .clk(clk), .dout(dout));
SR4_B UUT2 (.din(din), .clk(clk), .dout(dout2));
initial
begin
    clk=1'b0;
    din=1'b1;
end
always//run continuously to generate clokck signal
begin
     #5;
     clk=~clk;
end
endmodule


//module Tester();//testbench to test the 8-bit full adder
//reg [7:0] a,b;
//reg cin;
//wire [7:0] sum;
//wire cout;
//FullAdder8 UUT (.A(a), .B(b), .Cin(cin), .S(sum), .Cout(cout));
//initial
//begin
//    a=8'b10101010;
//    b=8'b10101010;
//    cin=1'b0;
//    #10;
//    a=8'b11110000;
//    b=8'b00001111;
//    cin=1'b0;
//    #10;
//    a=8'b11001100;
//    b=8'b10101010;
//    cin=1'b0;
//    #10;
//    a=8'b10110110;
//    b=8'b00110010;
//    cin=1'b0;
//    #10;
//end
//endmodule


//module Tester();//testbench to test the EvenParity module
```

```
//wire [8:0] d_out;
//reg [7:0] d_in;
//EvenParity UUT(.data_in(d_in),.data_out(d_out));
//initial
//begin
//      d_in=8'b10101010;
//      #10;
//      d_in=8'b10101000;
//      #10;
//end
//endmodule


//module Tester(      );//testbench to test Majority3 module
//wire X;
//reg A,B,C;
//Majority3 UUT (A,B,C,X); //create instance of Majority3 module for testing
//initial
//begin
//      A=0;
//      B=0;
//      C=0;
//      #10 //simulate 10 units of delay
//      A=0;
//      B=0;
//      C=1;
//      #10 //simulate 10 units of delay
//      A=0;
//      B=1;
//      C=0;
//      #10 //simulate 10 units of delay
//      A=0;
//      B=1;
//      C=1;
//      #10 //simulate 10 units of delay
//      A=1;
//      B=0;
//      C=0;
//      #10 //simulate 10 units of delay
//      A=1;
//      B=0;
//      C=1;
//      #10 //simulate 10 units of delay
//      A=1;
//      B=1;
//      C=0;
//      #10 //simulate 10 units of delay
//      A=1;
//      B=1;
//      C=1;
//end
//endmodule
```

```
//module Tester();    //create n 8-bit LFSR instance
//parameter N=3;
//reg CLK, EN, SEED;
//reg [N-1:0] SEED_DATA;
//wire [N-1:0] LFSR_DATA;
//wire REPEAT;
//LFSR #(.n(N)) UUT(.clk(CLK), .en(EN), .seed(SEED), .seed_data(SEED_DATA), .LFSR_data(LFSR_DATA), .Repeat(REPEAT));
//initial
//begin
//      CLK=0;
//      EN=0;
//      SEED=0;
//      SEED_DATA=3'b100;
//      #10;
//      EN=1;
//      SEED=1;
//      #20;
//      SEED=0;
//end
//always
//begin
//      #5;
//      CLK=~CLK;
//end
//endmodule
```