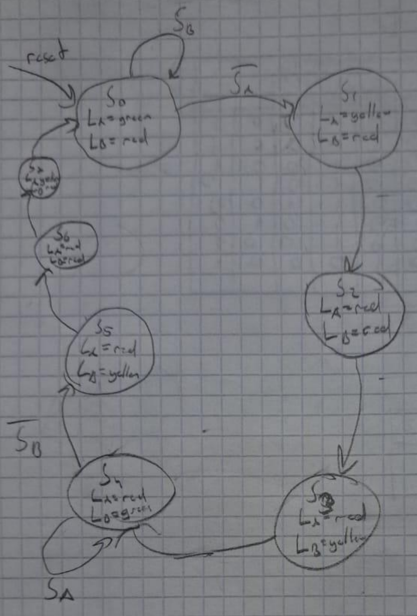
CS223 LAB5

Altay İlker Yiğitel
22203024
CS223 sec: 01


reset
S_B
$\overline{S_A}$
S_0 L_A=green L_B=red
S_1 L_A=yellow L_B=red
S_2 L_A=red L_B=red
S_3 L_A=red L_B=yellow
S_4 L_A=red L_B=green
S_5 L_A=red L_B=yellow
S_6 L_A=red L_B=red
S_7 L_A=yellow L_B=red
$\overline{S_B}$
S_A

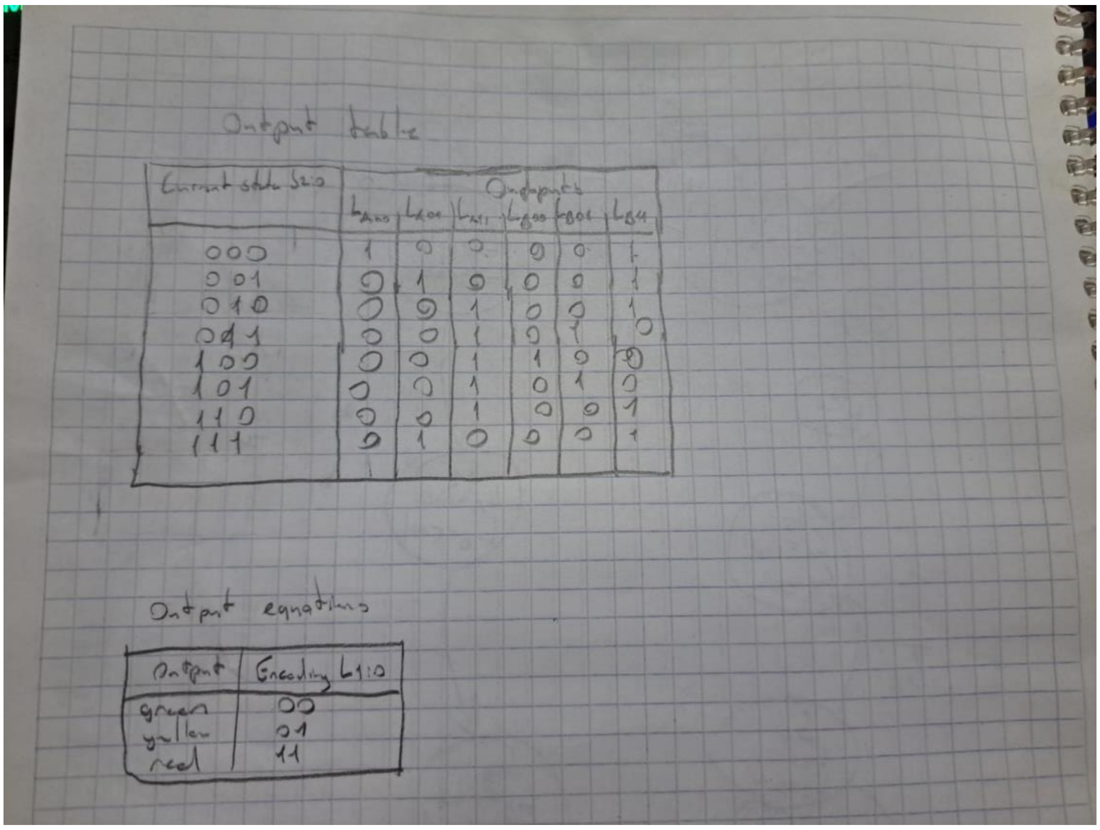
$\overline{S_a}$	$\wedge$	$\overline{S_b}$
1		1
1		0
0		1
0		0

state transition table

Current State S	Inputs S_A S_B	Next State S'
S_0	X 1	S_0
S_0	0 0	S_1
S_1	X X	S_2
S_2	X X	S_3
S_3	X X	S_4
S_4	1 X	S_4
S_4	0 0	S_5
S_5	X X	S_6
S_6	X X	S_7
S_7	X X	S_0

state encodings

State	Encoding $S_{2:0}$
S_0	000
S_1	001
S_2	010
S_3	011
S_4	100
S_5	101
S_6	110
S_7	111

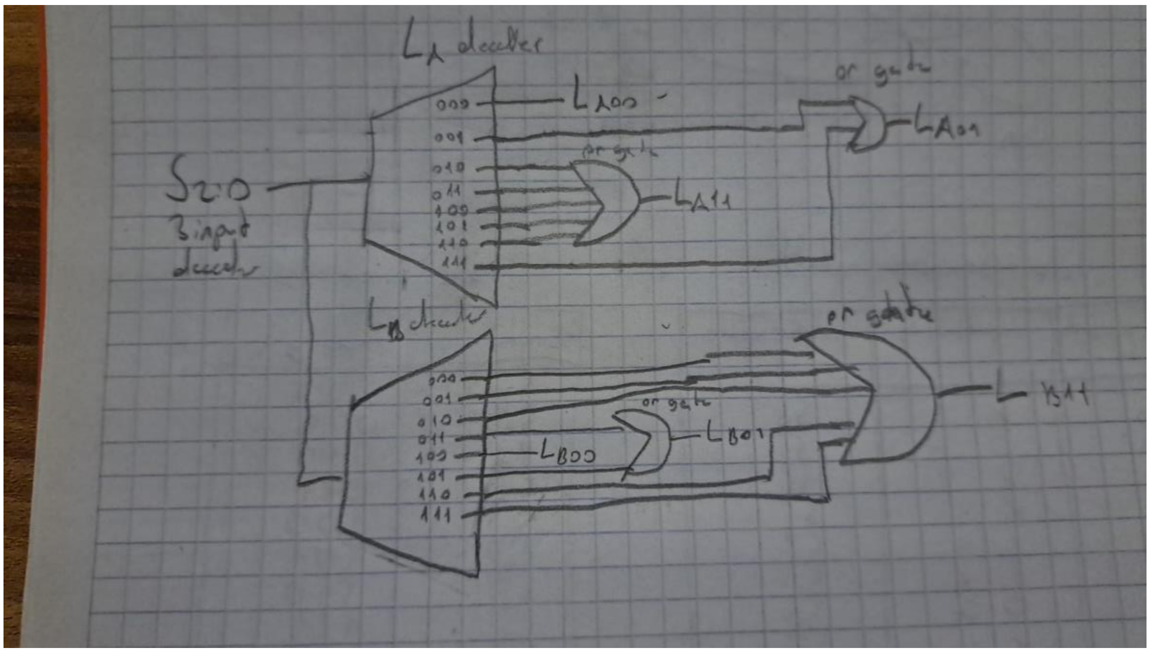

Output table

Current state S2:0	Outputs					
	L_{A00}	L_{A01}	L_{A11}	L_{B00}	L_{B01}	L_{B11}
000	1	0	0	0	0	1
001	0	1	0	0	0	1
010	0	0	1	0	0	1
011	0	0	1	0	1	0
100	0	0	1	1	0	0
101	0	0	1	0	1	0
110	0	0	1	0	0	1
111	0	1	0	0	0	1

Output equations

Output	Encoding L1:0
green	00
yellow	01
red	11

2 flip flops are needed as there are 2 important states La=green Lb=red and La=red and Lb=green



System verilog code

Traffic Light System

module traffic(

```
    input SA,
    input SB,
    output LA_green,
    output LA_yellow,
    output LA_red,
    output LB_green,
    output LB_yellow,
    output LB_red,
    input clk
);

typedef enum logic [3:0] {
    BOTH_RED1,
    BOTH_RED2,
    LA_GREEN,
    LB_GREEN,
    LA_YELLOW1,
    LB_YELLOW1,
    LA_YELLOW2,
    LB_YELLOW2
} state_t;

state_t current_state, next_state;
logic [1:0] counter;



initial begin
    current_state = LA_GREEN;
    counter = 0;
    LA_green = 1; LA_yellow = 0; LA_red = 0;
    LB_green = 0; LB_yellow = 0; LB_red = 1;
end

always @(posedge clk) begin
    case (current_state)
        LA_GREEN: begin
            if (!(SA && SB)) begin
                next_state = LA_GREEN;
            end else begin
                next_state = LA_YELLOW1;
            end
        end
        LB_GREEN: begin
            if (!(SA && SB)) begin
                next_state = LB_GREEN;
            end else begin
                next_state = LB_YELLOW1;
            end
        end
        BOTH_RED1: begin
            if (counter == 3) begin
                next_state = LA_YELLOW2;
            end else begin
                next_state = BOTH_RED1;
            end
        end
        BOTH_RED2: begin
            if (counter == 3) begin
                next_state = LB_YELLOW2;
            end else begin
                next_state = BOTH_RED2;
            end
        end
        LA_YELLOW1: begin
            if (counter == 3) begin
                next_state = BOTH_RED1;
            end else begin
                next_state = LA_YELLOW1;
            end
        end
        LB_YELLOW1: begin
            if (counter == 3) begin
                next_state = BOTH_RED2;
            end else begin
                next_state = LB_YELLOW1;
            end
        end
        LA_YELLOW2: begin
            if (counter == 3) begin
                next_state = LB_GREEN;
            end else begin
                next_state = LA_YELLOW2;
            end
        end
        LB_YELLOW2: begin
            if (counter == 3) begin
                next_state = LA_GREEN;
```

```
            end else begin
                next_state = LB_YELLOW2;
            end
        end
    endcase
end

always @(posedge clk) begin
    if (counter == 3) begin
        counter <= 0;
    end else begin
        counter <= counter + 1;
    end

    case (next_state)
        BOTH_RED: begin
            LA_green = 0; LA_yellow = 0; LA_red = 1;
            LB_green = 0; LB_yellow = 0; LB_red = 1;
        end
        BOTH_YELLOW: begin
            LA_green = 0; LA_yellow = 1; LA_red = 0;
            LB_green = 0; LB_yellow = 1; LB_red = 0;
        end
        LA_GREEN: begin
            LA_green = 1; LA_yellow = 0; LA_red = 0;
            LB_green = 0; LB_yellow = 0; LB_red = 1;
        end
        LB_GREEN: begin
            LA_green = 0; LA_yellow = 0; LA_red = 1;
            LB_green = 1; LB_yellow = 0; LB_red = 0;
        end
        LA_YELLOW: begin
            LA_green = 0; LA_yellow = 1; LA_red = 0;
            LB_green = 0; LB_yellow = 0; LB_red = 1;
        end
        LB_YELLOW: begin
            LA_green = 0; LA_yellow = 0; LA_red = 1;
            LB_green = 0; LB_yellow = 1; LB_red = 0;
        end
    endcase

    current_state <= next_state;
end

endmodule
```

Testbench

```
module traffic_tb;

    input SA;
    input SB;
    input clk;

    output LA_green;
    output LA_yellow;
    output LA_red;
    output LB_green;

    output LB_yellow;

    output LB_red;


    traffic dut (

        .SA(SA),

        .SB(SB),

        .LA_green(LA_green),

        .LA_yellow(LA_yellow),

        .LA_red(LA_red),

        .LB_green(LB_green),

        .LB_yellow(LB_yellow),

        .LB_red(LB_red),

        .clk(clk)

    );


    always #5 clk = ~clk;


    initial begin

        SA = 0;

        SB = 0;


        #10;


        assert(LA_green && LB_red);


        SA = 1;

        #10;

        assert(LA_yellow && LB_red);
```

```
        SA = 0;
        SB = 1;
        #10;
        assert(LB_green && LA_red);

        SA = 1;
        #10;
        assert(LA_red && LB_red);

        $finish;
    end

endmodule
```

Counter

```
module BCD_Counter(
    input clk,
    input reset,
    input enable,
    input load,
    input [3:0] D,
    output [3:0] Q
);

always @(posedge clk) begin
    if (reset) begin
        Q <= 4'b0000;
    end
     if (enable) begin
        if (load) begin
            Q <= D;
        end else begin
            if (Q == 4'b1001) begin
                Q <= 4'b0000;
            end else begin
                Q <= Q + 1;
            end
        end
    end
end

endmodule
```

testbench

```
module BCD_Counter_TB;

    input clk;
    input reset;
    input enable;
    input load;
    input [3:0] D;
    output [3:0] Q;

    BCD_Counter counter(
        .clk(clk),
        .reset(reset),
        .enable(enable),
        .load(load),
        .D(D),
        .Q(Q)
    );

    always #((CLK_PERIOD)/2) clk = ~clk;

    initial begin
        clk = 0;
        reset = 0;
        enable = 0;
        load = 0;
        D = 0;
        #20 reset = 1;
        #20 reset = 0;
        #10 enable = 1;

        #20;
        assert(Q == 4'b0001) else ;

        #30;
        assert(Q == 4'b0000) ;

        D = 4'b1100;
        load = 1;
        #20;
        load =0;
        assert(Q == 4'b1100) ;
```

```
        #20;
        assert(Q == 4'b1101) ;
    $finish;
    end

endmodule
```

BCD Counter

```
module BCD_Counter(
    input clk,
    input reset,
    input enable,
    output [3:0] BCD
);


always @(posedge clk) begin
    if (reset) begin
        BCD <= 4'b0000;
    end else if (enable) begin
        if (BCD == 4'b1001) begin
            BCD <= 4'b0000;
        end else begin
            BCD <= BCD+1;
        end
    end
end

endmodule
```

```
module BCD_Counter_Reset_TB;


    input clk;
    inpt reset;
    input enable;
    output [3:0] BCD;

    BCD_Counter counter(
        .clk(clk),
        .reset(reset),
        .enable(enable),
        .BCD(BCD)
    );

    always #((CLK_PERIOD)/2) clk = ~clk;

    initial begin
        clk = 0;
        reset = 0;
        enable = 0;
        #20 reset = 1;
        #20 reset = 0;
        #10 enable = 1;

        #20;
        assert(BCD == 4'b0001) ;
        #30;

        assert(BCD == 4'b0000) ;


        #30;

        assert(BCD == 4'b0001) ;


        #200;

        assert(BCD == 4'b1001) ;


        #30;

        assert(BCD == 4'b0000) ;


        #30;

        assert(BCD == 4'b0001);

         $finish;

    end


endmodule
```

Two digit

```
module TwoDigit_BCD_Counter(

    input clk,

    input reset,

    output [3:0]  ones,
    output [3:0]  tens


);



always @(posedge clk) begin
    if(ones ==9) begin
ones <=0;
if(tens ==9) begin
tens <=0;
end
else begin
                    tens <= tens+1;
          end
end
else begin
ones <=ones+1;
end
endmodule
```

Testbench

```
module TwoDigit_BCD_Counter_TB;

    reg clk;
    reg reset;
    wire [3:0] ones;
    wire [3:0] tens;
    TwoDigit_BCD_Counter counter(
        .clk(clk),
        .reset(reset),
        .ones(ones),
        .tens(tens),
    );

    always #((CLK_PERIOD)/2) clk = ~clk;

    initial begin
        clk = 0;
        reset = 0;
        #20 reset = 1;
        #20 reset = 0;



        #10;
        assert(ones == 4'b0000 && tens 4'b0000);


         $finish;
    end

endmodule
```