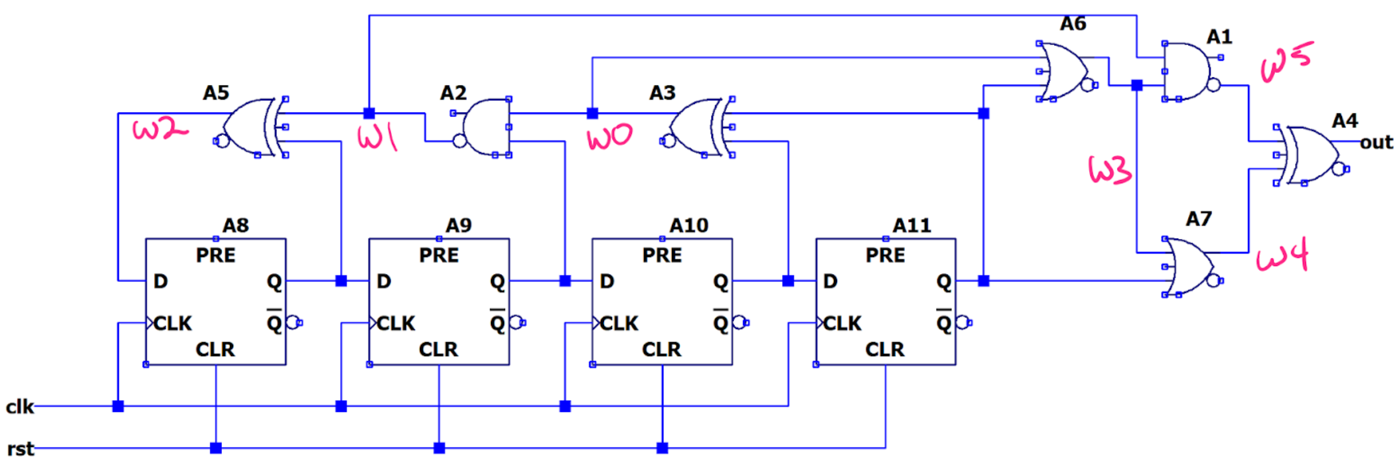
ECE 581 Project 2

7 November 2021

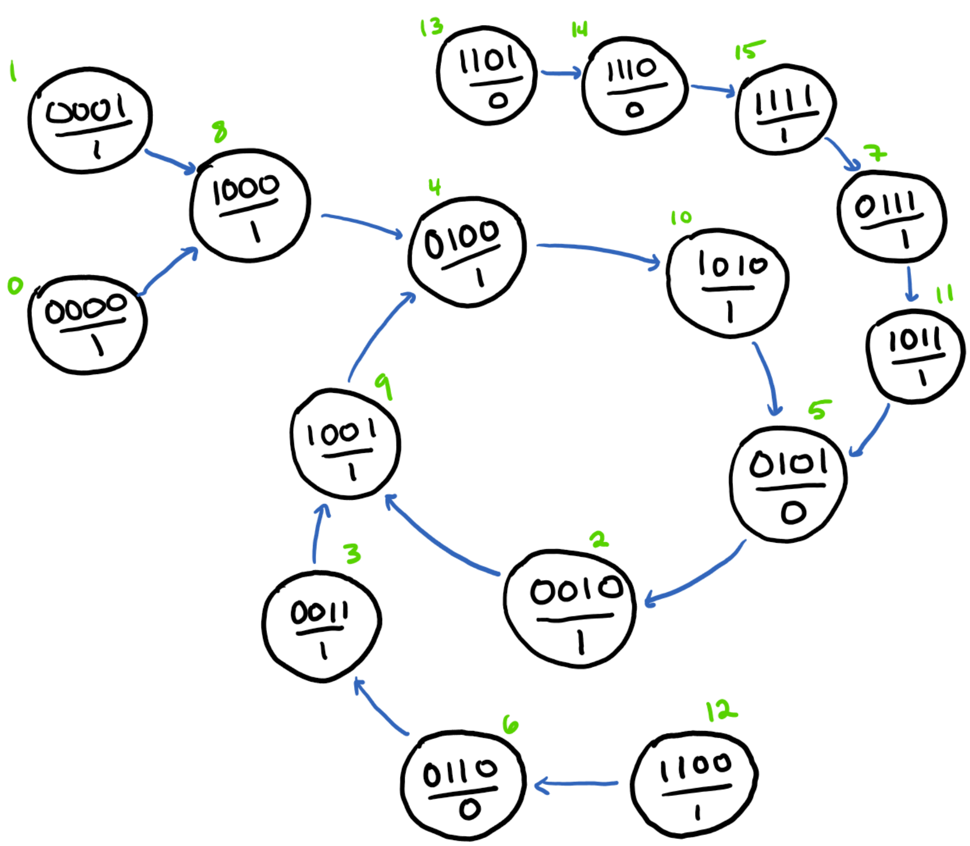
Chuck Faber (cfaber@pdx.edu)
Chris Mersman (cmersman@pdx.edu)

Problem 1

Circuit S1



S1 Transition Diagram



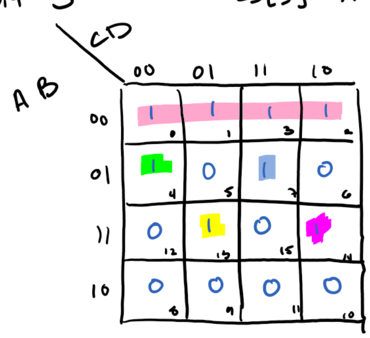
State Transition Diagram Truth Table

	Current State				Next State				
#	bit 3	bit 2	bit 1	bit 0	bit 3	bit 2	bit 1	bit 0	out
0	0	0	0	0	1	0	0	0	1
1	0	0	0	1	1	0	0	0	1
2	0	0	1	0	1	0	0	1	1
3	0	0	1	1	1	0	0	1	1
4	0	1	0	0	1	0	1	0	1
5	0	1	0	1	0	0	1	0	0
6	0	1	1	0	0	0	1	1	0
7	0	1	1	1	1	0	1	1	1
8	1	0	0	0	0	1	0	0	1
9	1	0	0	1	0	1	0	0	1
10	1	0	1	0	0	1	0	1	1
11	1	0	1	1	0	1	0	1	1
12	1	1	0	0	0	1	1	0	1
13	1	1	0	1	1	1	1	0	0
14	1	1	1	0	1	1	1	1	0
15	1	1	1	1	0	1	1	1	1

Karnaugh Maps

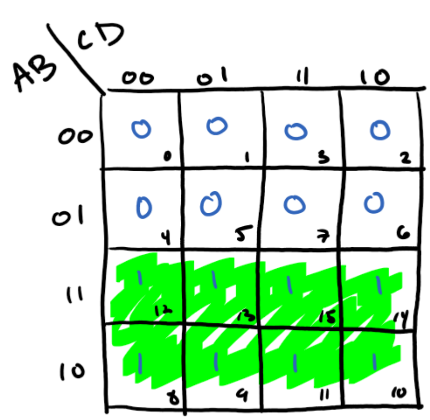
Bit 3

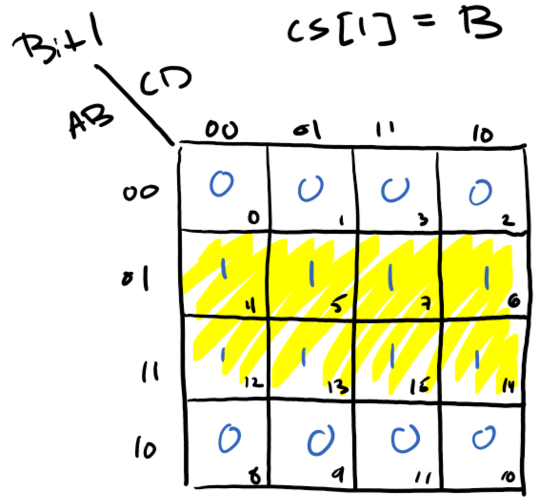
$$cs[3] = \bar{A}\bar{B} + \underbrace{\bar{A}B\bar{C}\bar{D} + \bar{A}BCD}_{\bar{A}B \,\&\, !(C \wedge D)} + \underbrace{AB\bar{C}D + ABC\bar{D}}_{AB \,\&\, (C \wedge D)}$$



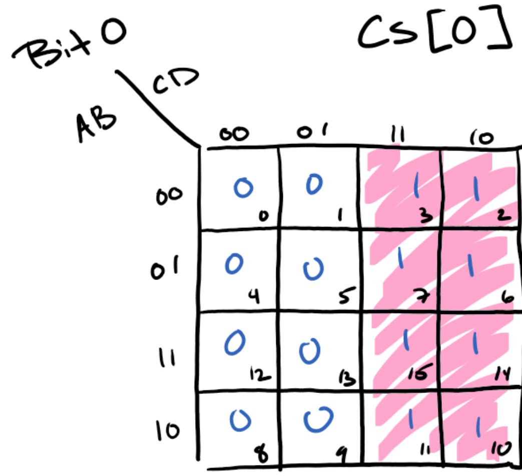
Bit 2

$$cs[2] = A$$

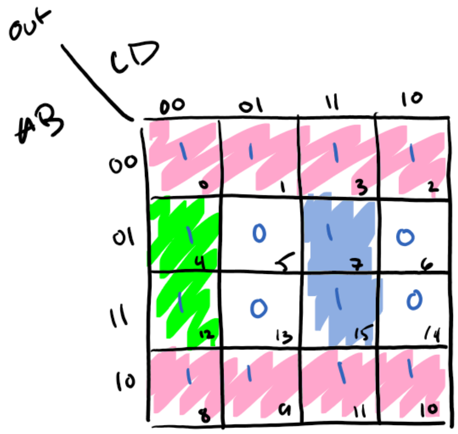




cs[1] = B



Cs[0] = C



$out = \bar{B} +$

$B\bar{C}\bar{D} +$ } B& !(C^D)

BCD

SystemVerilog Code

```
// Structural Design
module S1 (
    input clk, rst,
    output logic out
);

    logic [0:3] Q;
    logic [0:5] W;

    always_ff @(posedge clk, posedge rst) begin
        if (rst) Q <= 4'b0000;
        else begin
            Q <= {W[2], Q[0:2]};
        end
    end

    always_comb begin
        W[0] = Q[3] ^ Q[2];
        W[1] = ~(W[0] & Q[1]);
        W[2] = W[1] ^ Q[0];
```

```
        W[3] = W[0] | Q[3];
        W[4] = Q[3] | W[3];
        W[5] = ~(W[1] & W[3]);
        out = W[5] ^ W[4];
    end

endmodule
```

```
// FSM Design
module S1_FSM (
    input clk, rst,
    output logic out
);

    logic [3:0] cs, ns;

    always_ff @(posedge clk, posedge rst) begin: seq_logic
        if (rst) cs <= 4'b0000;
        else cs <= ns;
    end: seq_logic

    always_comb begin: next_state_logic
        case (cs)
            0:  ns = 8;
            1:  ns = 8;
            2:  ns = 9;
            3:  ns = 9;
            4:  ns = 10;
            5:  ns = 2;
            6:  ns = 3;
            7:  ns = 11;
            8:  ns = 4;
            9:  ns = 4;
            10: ns = 5;
            11: ns = 5;
            12: ns = 6;
            13: ns = 13;
            14: ns = 15;
            15: ns = 7;
            default: ns = 0;
        endcase
    end: next_state_logic

    always_comb begin: output_logic
        if ((cs == 5) || (cs == 6) || (cs == 13) || (cs == 14)) out = 1'b0;
        else out = 1'b1;
    end: output_logic
endmodule
```

```
// FSM Boolean Expression Design
module S1_BOOL_FSM (
    input clk, rst,
    output logic out
);
```

```
    logic [3:0] cs, ns;

    always_ff @(posedge clk, posedge rst) begin: seq_logic
        if (rst) cs <= 4'b0000;
        else cs <= ns;
    end: seq_logic

    always_comb begin: next_state_logic
        ns[3] = (!cs[3]&!cs[2]) | (!cs[3]&cs[2]&!(cs[1]^cs[0])) | (cs[3]&cs[2]&(cs[1]^cs[0]));
        ns[2] = cs[3];
        ns[1] = cs[2];
        ns[0] = cs[1];
    end: next_state_logic

    always_comb begin: output_logic
        out = !cs[2] | cs[2]&!(cs[1]^cs[0]);
    end: output_logic

endmodule
```

Testbench Code

```
module top ();
    logic clk, rst, out_0, out_1, out_2;

    initial begin
        clk = 1'b1;
        forever #50 clk = ~clk;
    end

    S1 s1_0 (
        .out(out_0),
        .*);
    S1_FSM s1_1 (
        .out(out_1),
        .*);
    S1_BOOL_FSM s1_2 (
        .out(out_2),
        .*);

    initial begin
        rst = 1'b1;
        #100;
        rst = 1'b0;
        $display("Q: %04b  Out: %01b\t\tFSM Q: %04b  Out: %01b\t\tBool FSM Q: %04b  Out: %01b", s1_0.Q,
out_0, s1_1.cs, out_1, s1_2.cs, out_2);
        #10

        repeat (50) @(negedge clk) begin
            $display("Q: %04b  Out: %01b\t\tFSM Q: %04b  Out: %01b\t\tBool FSM Q: %04b  Out: %01b", s1_0.Q,
out_0, s1_1.cs, out_1, s1_2.cs, out_2);
        end
        $finish();

    end
```

```
endmodule
```

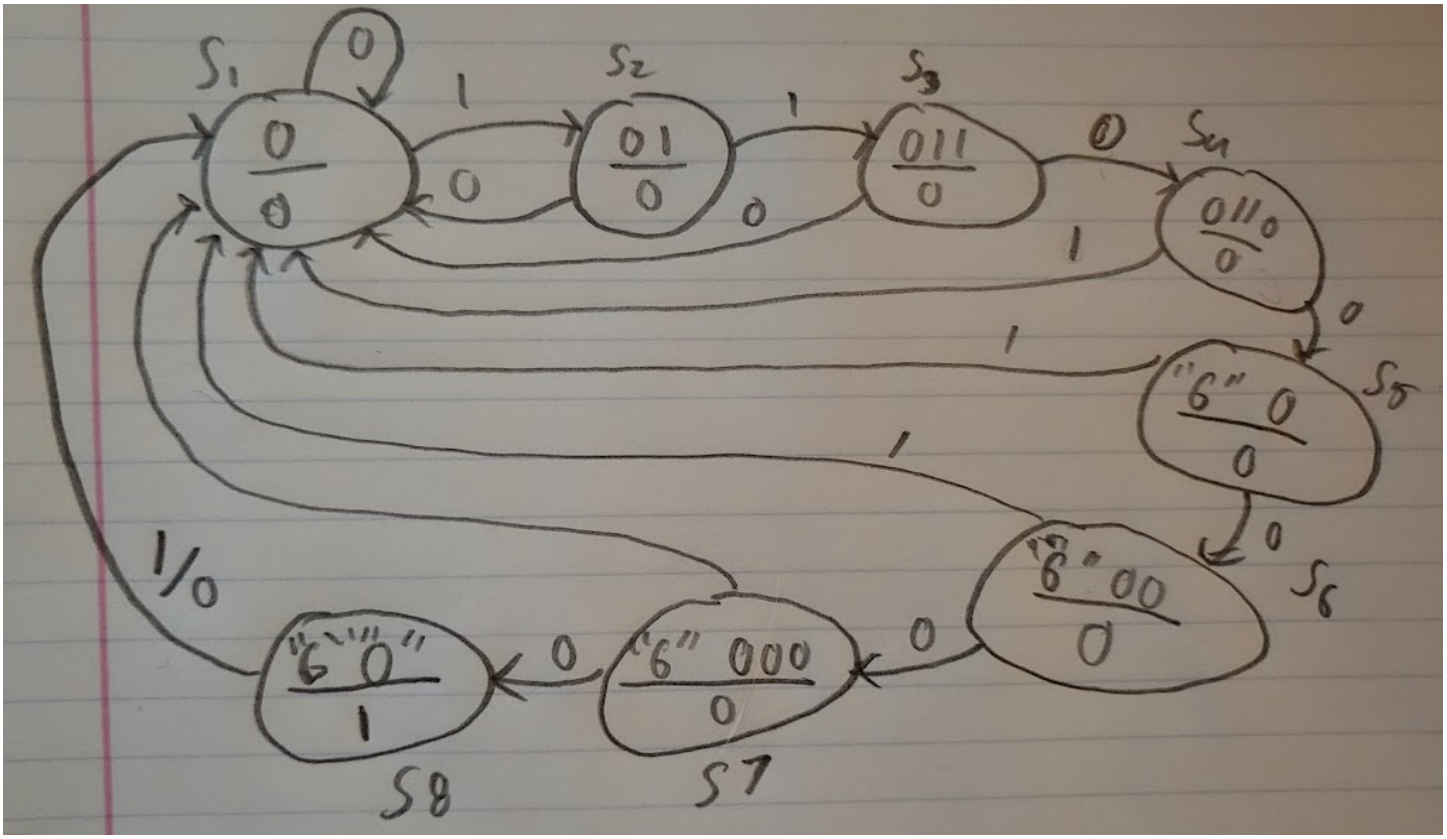
Transcript

```
VSIM 1> run -all
# Q: 0000  Out: 1                FSM Q: 0000  Out: 1                Bool FSM Q: 0000  Out: 1
# Q: 1000  Out: 1                FSM Q: 1000  Out: 1                Bool FSM Q: 1000  Out: 1
# Q: 0100  Out: 1                FSM Q: 0100  Out: 1                Bool FSM Q: 0100  Out: 1
# Q: 1010  Out: 1                FSM Q: 1010  Out: 1                Bool FSM Q: 1010  Out: 1
# Q: 0101  Out: 0                FSM Q: 0101  Out: 0                Bool FSM Q: 0101  Out: 0
# Q: 0010  Out: 1                FSM Q: 0010  Out: 1                Bool FSM Q: 0010  Out: 1
# Q: 1001  Out: 1                FSM Q: 1001  Out: 1                Bool FSM Q: 1001  Out: 1
# Q: 0100  Out: 1                FSM Q: 0100  Out: 1                Bool FSM Q: 0100  Out: 1
# Q: 1010  Out: 1                FSM Q: 1010  Out: 1                Bool FSM Q: 1010  Out: 1
# Q: 0101  Out: 0                FSM Q: 0101  Out: 0                Bool FSM Q: 0101  Out: 0
# Q: 0010  Out: 1                FSM Q: 0010  Out: 1                Bool FSM Q: 0010  Out: 1
# Q: 1001  Out: 1                FSM Q: 1001  Out: 1                Bool FSM Q: 1001  Out: 1
# Q: 0100  Out: 1                FSM Q: 0100  Out: 1                Bool FSM Q: 0100  Out: 1
# Q: 1010  Out: 1                FSM Q: 1010  Out: 1                Bool FSM Q: 1010  Out: 1
# Q: 0101  Out: 0                FSM Q: 0101  Out: 0                Bool FSM Q: 0101  Out: 0
# Q: 0010  Out: 1                FSM Q: 0010  Out: 1                Bool FSM Q: 0010  Out: 1
# Q: 1001  Out: 1                FSM Q: 1001  Out: 1                Bool FSM Q: 1001  Out: 1
# Q: 0100  Out: 1                FSM Q: 0100  Out: 1                Bool FSM Q: 0100  Out: 1
# Q: 1010  Out: 1                FSM Q: 1010  Out: 1                Bool FSM Q: 1010  Out: 1
# Q: 0101  Out: 0                FSM Q: 0101  Out: 0                Bool FSM Q: 0101  Out: 0
# Q: 0010  Out: 1                FSM Q: 0010  Out: 1                Bool FSM Q: 0010  Out: 1
# Q: 1001  Out: 1                FSM Q: 1001  Out: 1                Bool FSM Q: 1001  Out: 1
# Q: 0100  Out: 1                FSM Q: 0100  Out: 1                Bool FSM Q: 0100  Out: 1
# Q: 1010  Out: 1                FSM Q: 1010  Out: 1                Bool FSM Q: 1010  Out: 1
# Q: 0101  Out: 0                FSM Q: 0101  Out: 0                Bool FSM Q: 0101  Out: 0
# Q: 0010  Out: 1                FSM Q: 0010  Out: 1                Bool FSM Q: 0010  Out: 1
# Q: 1001  Out: 1                FSM Q: 1001  Out: 1                Bool FSM Q: 1001  Out: 1
# Q: 0100  Out: 1                FSM Q: 0100  Out: 1                Bool FSM Q: 0100  Out: 1
# Q: 1010  Out: 1                FSM Q: 1010  Out: 1                Bool FSM Q: 1010  Out: 1
# Q: 0101  Out: 0                FSM Q: 0101  Out: 0                Bool FSM Q: 0101  Out: 0
# Q: 0010  Out: 1                FSM Q: 0010  Out: 1                Bool FSM Q: 0010  Out: 1
# Q: 1001  Out: 1                FSM Q: 1001  Out: 1                Bool FSM Q: 1001  Out: 1
# Q: 0100  Out: 1                FSM Q: 0100  Out: 1                Bool FSM Q: 0100  Out: 1
# Q: 1010  Out: 1                FSM Q: 1010  Out: 1                Bool FSM Q: 1010  Out: 1
# Q: 0101  Out: 0                FSM Q: 0101  Out: 0                Bool FSM Q: 0101  Out: 0
# Q: 0010  Out: 1                FSM Q: 0010  Out: 1                Bool FSM Q: 0010  Out: 1
# Q: 1001  Out: 1                FSM Q: 1001  Out: 1                Bool FSM Q: 1001  Out: 1
# Q: 0100  Out: 1                FSM Q: 0100  Out: 1                Bool FSM Q: 0100  Out: 1
# Q: 1010  Out: 1                FSM Q: 1010  Out: 1                Bool FSM Q: 1010  Out: 1
# Q: 0101  Out: 0                FSM Q: 0101  Out: 0                Bool FSM Q: 0101  Out: 0
# Q: 0010  Out: 1                FSM Q: 0010  Out: 1                Bool FSM Q: 0010  Out: 1
# Q: 1001  Out: 1                FSM Q: 1001  Out: 1                Bool FSM Q: 1001  Out: 1
# Q: 0100  Out: 1                FSM Q: 0100  Out: 1                Bool FSM Q: 0100  Out: 1
# Q: 1010  Out: 1                FSM Q: 1010  Out: 1                Bool FSM Q: 1010  Out: 1
# Q: 0101  Out: 0                FSM Q: 0101  Out: 0                Bool FSM Q: 0101  Out: 0
# Q: 0010  Out: 1                FSM Q: 0010  Out: 1                Bool FSM Q: 0010  Out: 1
# Q: 1001  Out: 1                FSM Q: 1001  Out: 1                Bool FSM Q: 1001  Out: 1
# Q: 0100  Out: 1                FSM Q: 0100  Out: 1                Bool FSM Q: 0100  Out: 1
# Q: 1010  Out: 1                FSM Q: 1010  Out: 1                Bool FSM Q: 1010  Out: 1
# Q: 0101  Out: 0                FSM Q: 0101  Out: 0                Bool FSM Q: 0101  Out: 0
# Q: 0010  Out: 1                FSM Q: 0010  Out: 1                Bool FSM Q: 0010  Out: 1
```

```
# ** Note: $finish    : prob1.sv(127)
#    Time: 5050 ps  Iteration: 1  Instance: /top
# End time: 21:56:51 on Nov 06,2021, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0
```

Problem 2

Transition Diagram (encoded for "60")



SystemVerilog Code

```
// Project 2 problem 2 sequence detector
// this code designs a FSM designed to find the sequence "60" in BCD
// which is  0110 0000, it tests one bit at a time if a bit is not a match
// the output will be zero and the FSM will reset, stating not matched.
// if the input matches the correct sequence the output will be a match.
// By chris Mersman and Chuck Faber

module Sequence_Detector(s_in, clk, rst, d_out);
input clk; // clk signal
input rst; // rst input
input s_in; // binary input
output logic d_out; // output of the sequence detector

enum logic [7:0] {s1='h00, s2='h01, s3='h03, s4='h06, s5='h0c, s6='h18, s7='h30, s8='h60} cs, ns;

always_ff @(posedge clk, posedge rst) begin // handle reset
  if(rst==1)
    cs <= s1;
```

```
  else
    cs <= ns;
  end

    always_ff @(cs, s_in) begin // go to next state
        case(cs)
            s1: begin
                if(s_in==1) ns <= s2;
                else ns <= s1;
            end
            s2: begin
                if(s_in==1) ns <= s3;
                else ns <= s1;
            end
            s3:begin
                if(s_in==0) ns <= s4;
                else ns <= s1;
            end
            s4:begin
                if(s_in==0) ns <= s5;
                else ns <= s1;
            end
            s5:begin
                if(s_in==0) ns <= s6;
                else ns <= s1;
            end
            s6:begin
                if(s_in==0) ns <= s7;
                else ns <= s1;
            end
            s7:begin
                if(s_in==0) ns <= s8;
                else ns <= s1;
            end
            s8:begin
                ns <= s1;
            end
            default:ns <= s1;
        endcase
    end

    always @(cs) begin
        case(cs)
        s1:        d_out = 0;
        s2:        d_out = 0;
        s3:        d_out = 0;
        s4:        d_out = 0;
        s5:        d_out = 0;
        s6:        d_out = 0;
        s7:        d_out = 0;
        s8:        d_out = 1;
        endcase
    end
endmodule
```

Testbench Code

```
module top();

    parameter N = 10;

    logic s_in, clk, rst, d_out;
    logic [7:0] test_val;
    logic [N-1:0][7:0] test_stream;

    Sequence_Detector uut (.*);

    initial begin
        clk = 1'b0;
        forever #50 clk = ~clk;
    end

    initial begin

        for (int i = 0; i < N; i++) begin
            test_stream[i] = $random() & 8'hFF;
        end
        test_stream[4] = 8'b0110_0000;

        s_in = 1'b0;
        rst = 1'b1;
        repeat (2) @(negedge clk);
        rst = 1'b0;
        repeat (2) @(negedge clk);

        test_val = 8'b0110_0000;

        foreach(test_val[i]) begin
            s_in = test_val[i];
            repeat (1) @(negedge clk);
            $display("%0t\ts_in:%0b\td_out:%0b\t%0s", $time, s_in, d_out, d_out ? "Matched" : "Unmatched");
        end

        rst = 1'b1;
        repeat (2) @(negedge clk);
        rst = 1'b0;
        repeat (2) @(negedge clk);

        $display("\n\nTesting with stream of bits %0b", test_stream);

        foreach(test_stream[j]) begin
            for (int k=7; k >= 0; k--) begin
                s_in = test_stream[j][k];
                repeat (1) @(negedge clk);
                $display("%0t\ts_in:%0b\td_out:%0b\t%0s", $time, s_in, d_out, d_out ? "Matched" :
"Unmatched");
            end
        end


        $finish();
    end
endmodule
```

Transcript

```
VSIM 1> run -all
# 500   s_in:0  d_out:0 Unmatched
# 600   s_in:1  d_out:0 Unmatched
# 700   s_in:1  d_out:0 Unmatched
# 800   s_in:0  d_out:0 Unmatched
# 900   s_in:0  d_out:0 Unmatched
# 1000  s_in:0  d_out:0 Unmatched
# 1100  s_in:0  d_out:0 Unmatched
# 1200  s_in:0  d_out:1 Matched
#
#
# Testing with stream of bits 1101000000010001001001100101100011010110000001100011000010011000000100100100
# 1700  s_in:0  d_out:0 Unmatched
# 1800  s_in:0  d_out:0 Unmatched
# 1900  s_in:0  d_out:0 Unmatched
# 2000  s_in:0  d_out:0 Unmatched
# 2100  s_in:1  d_out:0 Unmatched
# 2200  s_in:1  d_out:0 Unmatched
# 2300  s_in:0  d_out:0 Unmatched
# 2400  s_in:1  d_out:0 Unmatched
# 2500  s_in:0  d_out:0 Unmatched
# 2600  s_in:0  d_out:0 Unmatched
# 2700  s_in:0  d_out:0 Unmatched
# 2800  s_in:0  d_out:0 Unmatched
# 2900  s_in:0  d_out:0 Unmatched
# 3000  s_in:0  d_out:0 Unmatched
# 3100  s_in:0  d_out:0 Unmatched
# 3200  s_in:1  d_out:0 Unmatched
# 3300  s_in:0  d_out:0 Unmatched
# 3400  s_in:0  d_out:0 Unmatched
# 3500  s_in:0  d_out:0 Unmatched
# 3600  s_in:1  d_out:0 Unmatched
# 3700  s_in:0  d_out:0 Unmatched
# 3800  s_in:0  d_out:0 Unmatched
# 3900  s_in:1  d_out:0 Unmatched
# 4000  s_in:0  d_out:0 Unmatched
# 4100  s_in:0  d_out:0 Unmatched
# 4200  s_in:1  d_out:0 Unmatched
# 4300  s_in:1  d_out:0 Unmatched
# 4400  s_in:0  d_out:0 Unmatched
# 4500  s_in:0  d_out:0 Unmatched
# 4600  s_in:1  d_out:0 Unmatched
# 4700  s_in:0  d_out:0 Unmatched
# 4800  s_in:1  d_out:0 Unmatched
# 4900  s_in:1  d_out:0 Unmatched
# 5000  s_in:0  d_out:0 Unmatched
# 5100  s_in:0  d_out:0 Unmatched
# 5200  s_in:0  d_out:0 Unmatched
# 5300  s_in:1  d_out:0 Unmatched
# 5400  s_in:1  d_out:0 Unmatched
# 5500  s_in:0  d_out:0 Unmatched
# 5600  s_in:1  d_out:0 Unmatched
# 5700  s_in:0  d_out:0 Unmatched
# 5800  s_in:1  d_out:0 Unmatched
# 5900  s_in:1  d_out:0 Unmatched
```

```
# 6000  s_in:0  d_out:0 Unmatched
# 6100  s_in:0  d_out:0 Unmatched
# 6200  s_in:0  d_out:0 Unmatched
# 6300  s_in:0  d_out:0 Unmatched
# 6400  s_in:0  d_out:1 Matched
# 6500  s_in:0  d_out:0 Unmatched
# 6600  s_in:1  d_out:0 Unmatched
# 6700  s_in:1  d_out:0 Unmatched
# 6800  s_in:0  d_out:0 Unmatched
# 6900  s_in:0  d_out:0 Unmatched
# 7000  s_in:0  d_out:0 Unmatched
# 7100  s_in:1  d_out:0 Unmatched
# 7200  s_in:1  d_out:0 Unmatched
# 7300  s_in:0  d_out:0 Unmatched
# 7400  s_in:0  d_out:0 Unmatched
# 7500  s_in:0  d_out:0 Unmatched
# 7600  s_in:0  d_out:0 Unmatched
# 7700  s_in:1  d_out:0 Unmatched
# 7800  s_in:0  d_out:0 Unmatched
# 7900  s_in:0  d_out:0 Unmatched
# 8000  s_in:1  d_out:0 Unmatched
# 8100  s_in:1  d_out:0 Unmatched
# 8200  s_in:0  d_out:0 Unmatched
# 8300  s_in:0  d_out:0 Unmatched
# 8400  s_in:0  d_out:0 Unmatched
# 8500  s_in:0  d_out:0 Unmatched
# 8600  s_in:0  d_out:1 Matched
# 8700  s_in:0  d_out:0 Unmatched
# 8800  s_in:1  d_out:0 Unmatched
# 8900  s_in:0  d_out:0 Unmatched
# 9000  s_in:0  d_out:0 Unmatched
# 9100  s_in:1  d_out:0 Unmatched
# 9200  s_in:0  d_out:0 Unmatched
# 9300  s_in:0  d_out:0 Unmatched
# 9400  s_in:1  d_out:0 Unmatched
# 9500  s_in:0  d_out:0 Unmatched
# 9600  s_in:0  d_out:0 Unmatched
# ** Note: $finish    : prob2_tb.sv(126)
#    Time: 9600 ps  Iteration: 1  Instance: /top
# End time: 22:28:51 on Nov 07,2021, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0
```

Problem 3

SystemVerilog Code

```
// Bubble Sort Algorithm

module bubbleSort #(parameter N=3) (
    input clk, rst, we, sort,
    input [31:0] d,
    input [N-1:0] a,
    output logic [31:0] q,
    output logic done
    );
```

```
    logic [31:0] mem [0:2**N-1];
    logic [N-1:0] i, j;

    // Inferred RAM
    always_ff @(posedge clk or posedge rst) begin
        if (rst) begin
            done <= 1'b0;
            q <= '0;
            i <= 2**N-1;
            j <= '0;
        end else if (we) begin
            // New values written to RAM, restart sorting variables
            done <= 1'b0;
            i <= 2**N-1;
            j <= '0;
            // Write d value to memory
            mem[a] <= d;
        end
        q <= mem[a];
    end

    // Sorting
    always_ff @(posedge clk) begin
        if (sort && !done) begin
            if (mem[j] > mem[j+1]) begin // swap
                mem[j] <= mem[j+1];
                mem[j+1] <= mem[j];
            end
            if (j > 2**N-2) begin
                j <= '0;
                i <= i-1;
            end else i <= i;
            if (i < 1) begin
                done <= 1'b1;
                i <= 2**N-1;
            end
            j <= j+1;
        end
    end

endmodule
```

Testbench Code

```
module top();

    parameter N = 3;

    logic clk, rst, we, sort, done;
    logic [31:0] d, q;
    logic [N-1:0] a;

    bubbleSort #(N) bSort (.*);

    initial begin
        clk = 1'b0;
```

```
        forever #50 clk = ~clk;
    end

    initial begin
        rst = 1'b1;
        repeat (3) @(negedge clk);
        rst = 1'b0;

        // Testing Writing to RAM
        $display("Testing Writing to RAM.");
        for (int i = 0; i < 2**N; i++) begin
            d = i;
            a = i;
            we = 1'b1;
            repeat (1) @(negedge clk);
        end
        we = 1'b0;

        // Testing Reading from RAM
        $display("Testing Reading from RAM");
        for (int i = 0; i < 2**N; i++) begin
            a = i;
            repeat (1) @(negedge clk);
            $display("Read %0d from location %0d.", q, a);
        end

        rst = 1'b1;
        repeat (3) @(negedge clk);
        rst = 1'b0;

        $display("\n\n");
        // Writing random values to RAM
        $display("Writing random values to RAM");
        for (int i = 0; i < 2**N; i++) begin
            d = $urandom() % 10;
            a = i;
            $display("Writing %0d to location %0d.", d, a);
            we = 1'b1;
            repeat (1) @(negedge clk);
        end
        we = 1'b0;

        $display("\n\nSorting.");
        while (!done) begin
            sort = 1'b1;
            repeat (1) @(negedge clk);
        end
        sort = 1'b0;

        // Reading from RAM
        $display("\n\nReading Sorted Values from RAM");
        for (int i = 0; i < 2**N; i++) begin
            a = i;
            repeat (1) @(negedge clk);
            $display("Read %0d from location %0d.", q, a);
        end

        $finish();
```

```
    end

endmodule
```

Transcript

```
$ vsim -c top
Reading pref.tcl

# 2020.1

# vsim -c top
# Start time: 20:03:09 on Nov 07,2021
# Loading sv_std.std
# Loading work.top
# Loading work.bubbleSort
VSIM 1> run -all
# Testing Writing to RAM.
# Testing Reading from RAM
# Read 0 from location 0.
# Read 1 from location 1.
# Read 2 from location 2.
# Read 3 from location 3.
# Read 4 from location 4.
# Read 5 from location 5.
# Read 6 from location 6.
# Read 7 from location 7.
#
#
#
# Writing random values to RAM
# Writing 7 to location 0.
# Writing 8 to location 1.
# Writing 4 to location 2.
# Writing 0 to location 3.
# Writing 0 to location 4.
# Writing 9 to location 5.
# Writing 6 to location 6.
# Writing 0 to location 7.
#
#
# Sorting.
#
#
# Reading Sorted Values from RAM
# Read 0 from location 0.
# Read 0 from location 1.
# Read 0 from location 2.
# Read 4 from location 3.
# Read 6 from location 4.
# Read 7 from location 5.
# Read 8 from location 6.
# Read 9 from location 7.
# ** Note: $finish    : prob3.sv(124)
#    Time: 9500 ps  Iteration: 1  Instance: /top
# End time: 20:03:11 on Nov 07,2021, Elapsed time: 0:00:02
```

```
# Errors: 0, Warnings: 0
```

Problem 4 - Synchronous FIFO

SystemVerilog Code

```
module FIFO #(parameter N = 3, parameter W = 4) (
    input wr_en, rd_en, clk, rst,
    input [W-1:0] wr_data,
    output logic [W-1:0] rd_data,
    output logic full, empty
);

    logic [N:0] rd_ctr, wr_ctr;
    logic [N-1:0] rd_ptr, wr_ptr;
    logic [W-1:0] fifo_data [0:2**N-1];

    assign rd_ptr = rd_ctr[N-1:0];
    assign wr_ptr = wr_ctr[N-1:0];

    // FF write counter
    always_ff @(posedge clk or posedge rst) begin: write_counter
        if (rst) wr_ctr <= '0;
        else if (!full && wr_en) begin
            wr_ctr <= wr_ctr + 1;
        end else wr_ctr <= wr_ctr;
    end: write_counter

    // FF read pointer
    always_ff @(posedge clk or posedge rst) begin: read_counter
        if (rst) rd_ctr <= '0;
        else if (!empty && rd_en) begin
            rd_ctr <= rd_ctr + 1;
        end else rd_ctr <= rd_ctr;
    end: read_counter

    // Read data
    always_ff @(posedge clk or posedge rst) begin: read_data
        if (rst) rd_data <= '0;
        else if (!empty && rd_en) begin
            rd_data <= fifo_data[rd_ptr];
        end else rd_data <= rd_data;
    end: read_data

    // Write data
    always_ff @(posedge clk or posedge rst) begin: write_data
        if (!full && wr_en) begin
            fifo_data[wr_ptr] <= wr_data;
        end else fifo_data[wr_ptr] <= fifo_data[wr_ptr];
    end: write_data

    // full and empty logic
    always_comb begin: full_empty
        if (rd_ptr == wr_ptr) begin
            full = (wr_ctr[N] != rd_ctr[N]) ? 1'b1 : 1'b0;
            empty = (wr_ctr[N] == rd_ctr[N]) ? 1'b1 : 1'b0;
```

```
        end else begin
            full = 1'b0;
            empty = 1'b0;
        end
    end: full_empty
endmodule
```

Testbench Code

```
module top();
    parameter W = 4;
    parameter N = 3;

    logic clk, rst, wr_en, rd_en, full, empty;
    logic [W-1:0] rd_data, wr_data;
    int rdptr;

    FIFO #(.W(W), .N(N)) fifo_0 (.*);

    initial begin
        clk = 1'b0;
        forever #50 clk = ~clk;
    end

    initial begin
        rst = 1'b1;
        repeat (2) @(negedge clk);
        rst = 1'b0;
        repeat (2) @(negedge clk);

        $display("FIFO initial state. rd_data=%04b\tfull=%01b\tempty=%01b", rd_data, full, empty);

        // Test 1 write and 1 read
        $display("\n\nTest 1: 1 Write followed by 1 Read");
        wr_data = $random() & {W{1'b1}};
        $display("Writing %0d to location %0d.", wr_data, fifo_0.wr_ptr);
        wr_en = 1'b1;
        repeat(1) @(negedge clk);
        wr_en = 1'b0;
        $display("FIFO state. rd_data=%04b\tfull=%01b\tempty=%01b", rd_data, full, empty);
        $display("Reading data from location %0d.", fifo_0.rd_ptr);
        rd_en = 1'b1;
        repeat (1) @(negedge clk);
        rd_en = 1'b0;
        $display("FIFO state. rd_data=%04b\tfull=%01b\tempty=%01b", rd_data, full, empty);


        $display("\n\nTest 2: 6 writes followed by 6 reads");
        // 6 writes
        repeat(2) @(negedge clk);
        $display("FIFO state. rd_data=%04b\tfull=%01b\tempty=%01b", rd_data, full, empty);
        wr_en = 1'b1;
        for (int i = 0; i < 6; i++) begin
            wr_data = i & {W{1'b1}};
            $display("Writing %0d to location %0d.", wr_data, fifo_0.wr_ptr);
            repeat(1) @(negedge clk);
```

```
    end
    wr_en = 1'b0;
    $display("FIFO state. rd_data=%04b\tfull=%01b\tempty=%01b", rd_data, full, empty);

    // 6 reads
    repeat (2) @(negedge clk);
    rd_en = 1'b1;
    for (int i = 0; i < 6; i++) begin
        rdptr = fifo_0.rd_ptr;
        repeat (1) @(negedge clk);
        $display("Read %0d from location %0d", rd_data, rdptr);
    end
    rd_en = 1'b0;
    $display("FIFO state. rd_data=%04b\tfull=%01b\tempty=%01b", rd_data, full, empty);

    // Test 3 - write till full.
    $display("\n\nTest 3: Write until full.");
    repeat(2) @(negedge clk);
    $display("FIFO state. rd_data=%04b\tfull=%01b\tempty=%01b", rd_data, full, empty);
    wr_en = 1'b1;
    for (int i = 0; i < 2**N; i++) begin
        wr_data = i & {W{1'b1}};
        $display("Writing %0d to location %0d.", wr_data, fifo_0.wr_ptr);
        repeat(1) @(negedge clk);
    end
    wr_en = 1'b0;
    $display("FIFO state. rd_data=%04b\tfull=%01b\tempty=%01b", rd_data, full, empty);

    // Test 4a - Attempt to overflow.
    $display("\n\nTest 4a: Test overflow.");
    repeat(2) @(negedge clk);
    $display("FIFO state. rd_data=%04b\tfull=%01b\tempty=%01b", rd_data, full, empty);
    wr_en = 1'b1;
    for (int i = 0; i < 2**N; i++) begin
        wr_data = $random() & {W{1'b1}};
        $display("Writing %0d to location %0d.", wr_data, fifo_0.wr_ptr);
        repeat(1) @(negedge clk);
    end
    wr_en = 1'b0;
    $display("FIFO state. rd_data=%04b\tfull=%01b\tempty=%01b", rd_data, full, empty);

    // Test 4b - Read until Empty and check overflow
    $display("\n\nTest 4b: Read until empty. Check that random values were not written.");
    repeat (2) @(negedge clk);
    rd_en = 1'b1;
    for (int i = 0; i < 2**N; i++) begin
        rdptr = fifo_0.rd_ptr;
        repeat (1) @(negedge clk);
        $display("Read %0d from location %0d", rd_data, rdptr);
    end
    rd_en = 1'b0;
    $display("FIFO state. rd_data=%04b\tfull=%01b\tempty=%01b", rd_data, full, empty);

    // Test 4c - Attempt to underflow
    $display("\n\nTest 4c: Read while empty. Verify that the rd_data value doesn't change.");
    repeat (2) @(negedge clk);
    rd_en = 1'b1;
    for (int i = 0; i < 2**N; i++) begin
```

```
            rdptr = fifo_0.rd_ptr;
            repeat (1) @(negedge clk);
            $display("Read %0d from location %0d", rd_data, rdptr);
        end
        rd_en = 1'b0;
        $display("FIFO state. rd_data=%04b\tfull=%01b\tempty=%01b", rd_data, full, empty);

        $finish();
    end
endmodule
```

Transcript

```
VSIM 1> run -all
# FIFO initial state. rd_data=0000      full=0  empty=1
#
#
# Test 1: 1 Write followed by 1 Read
# Writing 4 to location 0.
# FIFO state. rd_data=0000      full=0  empty=0
# Reading data from location 0.
# FIFO state. rd_data=0100      full=0  empty=1
#
#
# Test 2: 6 writes followed by 6 reads
# FIFO state. rd_data=0100      full=0  empty=1
# Writing 0 to location 1.
# Writing 1 to location 2.
# Writing 2 to location 3.
# Writing 3 to location 4.
# Writing 4 to location 5.
# Writing 5 to location 6.
# FIFO state. rd_data=0100      full=0  empty=0
# Read 0 from location 1
# Read 1 from location 2
# Read 2 from location 3
# Read 3 from location 4
# Read 4 from location 5
# Read 5 from location 6
# FIFO state. rd_data=0101      full=0  empty=1
#
#
# Test 3: Write until full.
# FIFO state. rd_data=0101      full=0  empty=1
# Writing 0 to location 7.
# Writing 1 to location 0.
# Writing 2 to location 1.
# Writing 3 to location 2.
# Writing 4 to location 3.
# Writing 5 to location 4.
# Writing 6 to location 5.
# Writing 7 to location 6.
# FIFO state. rd_data=0101      full=1  empty=0
#
#
# Test 4a: Test overflow.
```

```
# FIFO state. rd_data=0101      full=1  empty=0
# Writing 1 to location 7.
# Writing 9 to location 7.
# Writing 3 to location 7.
# Writing 13 to location 7.
# Writing 13 to location 7.
# Writing 5 to location 7.
# Writing 2 to location 7.
# Writing 1 to location 7.
# FIFO state. rd_data=0101      full=1  empty=0
#
#
# Test 4b: Read until empty. Check that random values were not written.
# Read 0 from location 7
# Read 1 from location 0
# Read 2 from location 1
# Read 3 from location 2
# Read 4 from location 3
# Read 5 from location 4
# Read 6 from location 5
# Read 7 from location 6
# FIFO state. rd_data=0111      full=0  empty=1
#
#
# Test 4c: Read while empty. Verify that the rd_data value doesn't change.
# Read 7 from location 7
# Read 7 from location 7
# Read 7 from location 7
# Read 7 from location 7
# Read 7 from location 7
# Read 7 from location 7
# FIFO state. rd_data=0111      full=0  empty=1
# ** Note: $finish    : prob4.sv(185)
#    Time: 6200 ps  Iteration: 1  Instance: /top
# End time: 20:10:10 on Nov 06,2021, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
```

Method

For this design I used the (N+1) bit binary counters to represent an N-bit FIFO. I utilize N+1 counters using the MSB comparison when the least significant bits are equal to determine the state of the full and empty signal. The least significant bits are used to index into the FIFO data structure which is an unpacked array. When the FIFO is in overflow (the FIFO is full) attempts to write to the FIFO will be unsuccessful and it will retain the data it already has in the FIFO. When the FIFO is in underflow, (the FIFO is empty), attempts to read from the FIFO will just produce the last read value, and the read pointer will not increment.

FIFO Signals		
Signal	**Direction**	**Description**
clk	input	Clock signal.
rst	input	Reset signal.

wr_en	input	Write-enable. Set high during a clock cycle, and the data on the wr_data lines will be ingested into the FIFO.
rd_en	input	Read-enable. Set high during a clock cycle, and the data at the current read pointer will be output on the rd_data lines.
wr_data	input	Set data to write to FIFO on these lines.
rd_data	output	When rd_en is set, data will be output on these lines.
full	output	Signal indicating the FIFO is full.
empty	output	Signal indicating the FIFO is empty.
rd_ctr	internal	Read counter. Increments when rd_en is high and FIFO is not empty.
wr_ctr	internal	Write counter. Increments when wr_en is high and FIFO is not full.
fifo_data	internal	Array structure holding FIFO data