CS 223 Lab Assignment II
Preliminary Work
Sıla Bozkurt
22401775
Section: 3
Instructor: Sinem Sav
November 1, 2025

1. 3x1 Multiplexer

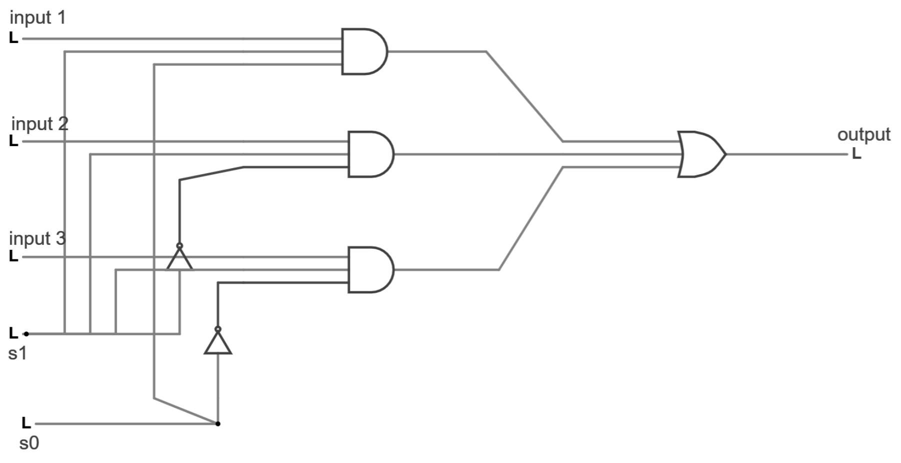
Boolean equation of a 3x1 multiplexer:

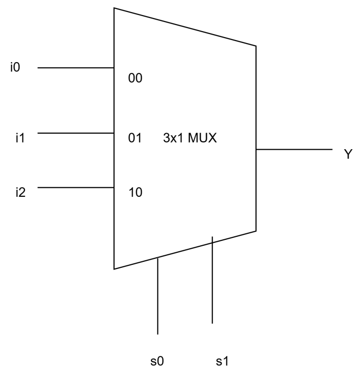
$$Y = \overline{S_1 S_0} D_0 + \overline{S_1} S_0 D_1 + S_1 \overline{S_0} D_2$$

D_0, D_1, D_2: inputs

S_0, S_1: select signals

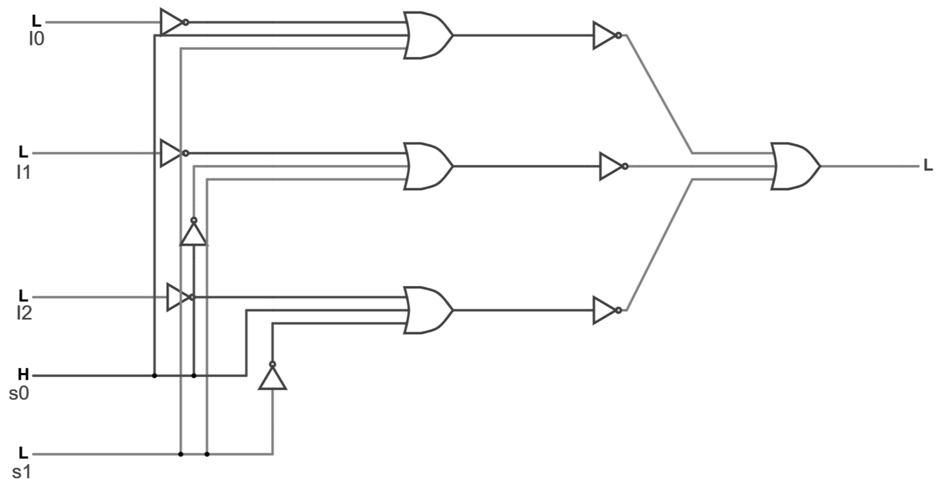
Y: output





Rewrite the equation without using the AND operator which can be found using **DeMorgan's Law**

$$Y = \overline{(S_1 + S_0 + \overline{D_0})} + \overline{(S_1 + \overline{S_0} + \overline{D_1})} + \overline{(\overline{S_1} + S_0 + \overline{D_2})}$$



1.1 Truth Table for 3x1 Multiplexer:

s0	s1	y
0	0	i0
0	1	i1
1	0	i2
1	1	X

1.2 Verilog module for 3x1 Multiplexer:

```
module threetoonemux(
   input i0,i1,i2,s0,s1,
   output y
   );
   assign y = (i0&~s0&~s1)|(i1&~s0&s1)|(i2&s0&~s1);
Endmodule
```

1.3 Testbench for 3x1 Multiplexer:

```
module tb_threetoonemux;

  logic i0, i1, i2;
  logic s0, s1;

  wire y;

  threetoonemux dut (
    .i0(i0),
    .i1(i1),
    .i2(i2),
    .s0(s0),
    .s1(s1),
    .y(y)
  );

  initial begin
    $monitor("Time=%0t | Select(s1,s0)=%b%b | Inputs(i2,i1,i0)=%b%b%b |
Output(y)=%b",
             $time, s1, s0, i2, i1, i0, y);
    i0 = 0; i1 = 0; i2 = 0;
    s0 = 0; s1 = 0;
    #10;

    $display("\n--- Testing Select i0 (s1=0, s0=0) ---");
    s1 = 0; s0 = 0;
    i0 = 1; i1 = 0; i2 = 0;
    #10;
    i0 = 0;
    #10;

    $display("\n--- Testing Select i2 (s1=0, s0=1) ---");
    s1 = 0; s0 = 1;
    i0 = 0; i1 = 0; i2 = 1;
    #10;
    i2 = 0;
    #10;

    $display("\n--- Testing Select i1 (s1=1, s0=0) ---");
    s1 = 1; s0 = 0;
    i0 = 0; i1 = 1; i2 = 0;
    #10;
    i1 = 0;
    #10;

    $display("\n--- Testing Unused State (s1=1, s0=1) ---");
    s1 = 1; s0 = 1;
    i0 = 1; i1 = 1; i2 = 1;
    #10;

    $display("\n--- Testbench Finished ---");
    $finish;
  end

Endmodule
```

2. 4x1 multilexer

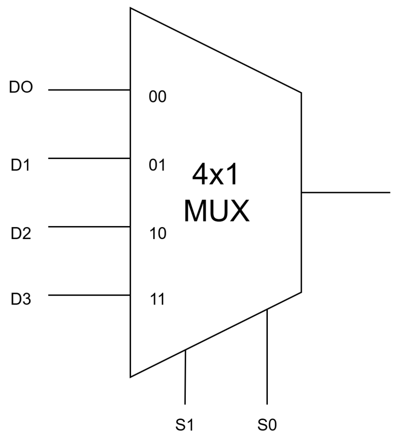
Boolean equation for a 4x1 multiplexer:

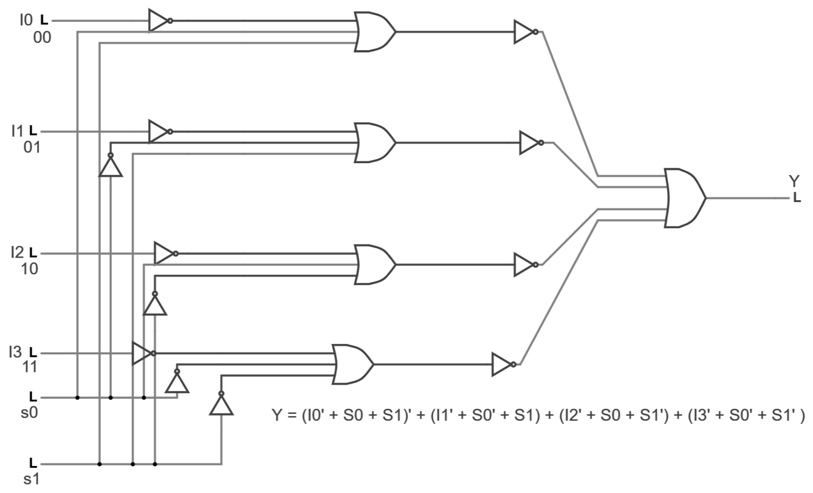
$$Y = \overline{S_1 S_0} D_0 + \overline{S_1} S_0 D_1 + S_1 \overline{S_0} D_2 + S_1 S_0 D_3$$

D_0, D_1, D_2, D_3: inputs

S_0, S_1: select signals

Y: output





I0 L
00
I1 L
01
I2 L
10
I3 L
11
L
s0
L
s1
Y
L
Y = (I0' + S0 + S1)' + (I1' + S0' + S1) + (I2' + S0 + S1') + (I3' + S0' + S1')

2.1 Truth Table for 4:1 Multiplexer:

s0	s1	y
0	0	i0
0	1	i1
1	0	i2
1	1	i3

2.2 Module for 4x1 multiplexer:

```
// This is a behavioral module for a 4-to-1 MUX
// It meets the requirement from
module fourtoonemux_behavioral(
    input  logic s0, s1,        // Select lines
    input  logic i0, i1, i2, i3, // Data inputs
    output logic y          // Output
);

    // Use always_comb for combinational logic
    always_comb begin
        // Use a case statement to select the output
        // based on the concatenated select lines {s1, s0}
        case ({s1, s0})
            2'b00: y = i0; // If s1=0, s0=0, select i0
            2'b01: y = i1; // If s1=0, s0=1, select i1
            2'b10: y = i2; // If s1=1, s0=0, select i2
            2'b11: y = i3; // If s1=1, s0=1, select i3
            default: y = 1'bx; // Assign 'x' (unknown) for any other state
        endcase
    end

endmodule
```

2.2.2 Behavioural:

```
module fourtoonemux_behavioral(
    input  logic s0, s1,
    input  logic i0, i1, i2, i3,
    output logic y          // Output
);

    always_comb begin
        case ({s1, s0})
            2'b00: y = i0; // If s1=0, s0=0, select i0
            2'b01: y = i1; // If s1=0, s0=1, select i1
            2'b10: y = i2; // If s1=1, s0=0, select i2
            2'b11: y = i3; // If s1=1, s0=1, select i3
            default: y = 1'bx; // Assign 'x' (unknown) for any other state
        endcase
    end

endmodule
```

2.3 Module for 2x1 multiplexer:

```
module twotoonemux(
    input s, i0, i1,
    output y
    );
    assign y = (~s & i0) | (s & i1);
endmodule
```

2.4 Testbench for 4x1 Multiplexer:

```
module tb_fourtoonemux;

  logic s0, s1;
  logic i0, i1, i2, i3;

  wire y;

  fourtoonemux dut (
    .s0(s0),
    .s1(s1),
    .i0(i0),
    .i1(i1),
    .i2(i2),
    .i3(i3),
    .y(y)
  );

  initial begin
    $monitor("Time=%0t | Select(s1,s0)=%b%b | Inputs(i3,i2,i1,i0)=%b%b%b%b | 
Output(y)=%b",
             $time, s1, s0, i3, i2, i1, i0, y);

    i0 = 0; i1 = 0; i2 = 0; i3 = 0;
    s0 = 0; s1 = 0;
    #10;

    $display("\n--- Testing Select i0 (s1=0, s0=0) ---");
    s1 = 0; s0 = 0;
    i0 = 1; i1 = 0; i2 = 0; i3 = 0;
    #10;
    i0 = 0;
    #10;

    $display("\n--- Testing Select i1 (s1=0, s0=1) ---");
    s1 = 0; s0 = 1;
    i0 = 0; i1 = 1; i2 = 0; i3 = 0;
    #10;
    i1 = 0;
    #10;

    $display("\n--- Testing Select i2 (s1=1, s0=0) ---");
    s1 = 1; s0 = 0;
```

```
        i0 = 0; i1 = 0; i2 = 1; i3 = 0;
        #10;
        i2 = 0;
        #10;

        $display("\n--- Testing Select i3 (s1=1, s0=1) ---");
        s1 = 1; s0 = 1;
        i0 = 0; i1 = 0; i2 = 0; i3 = 1;
        #10;
        i3 = 0;
        #10;

        $display("\n--- Testbench Finished ---");
        $finish;
      end

    endmodule
```

2.4.2 For behavioural

```
module tb_fourtoonemux;

  logic s0, s1;
  logic i0, i1, i2, i3;
  wire y;

  fourtoonemux dut (
    .s0(s0),
    .s1(s1),
    .i0(i0),
    .i1(i1),
    .i2(i2),
    .i3(i3),
    .y(y)
  );

  initial begin
    $display("Time | s1 s0 | i3 i2 i1 i0 |  y (Out)");
    $monitor("%4t | %b  %b  | %b  %b  %b  %b  |     %b",
             $time, s1, s0, i3, i2, i1, i0, y);

    i0=0; i1=1; i2=0; i3=1;
```

```
        {s1, s0} = 2'b00; #10;
        {s1, s0} = 2'b01; #10;
        {s1, s0} = 2'b10; #10;
        {s1, s0} = 2'b11; #10;

        i0=1; i1=0; i2=1; i3=0;

        {s1, s0} = 2'b00; #10;
        {s1, s0} = 2'b01; #10;
        {s1, s0} = 2'b10; #10;
        {s1, s0} = 2'b11; #10;

        $display("\nTestbench Finished.");
        $finish;
    end

endmodule
```

3. Functions

w1	w2	w3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

3.1 Function f

3.1.1 Which operation is realized by the function f given in Table1 a?
First in order to apply Sum-of-products Form, all the combinations where f = 1 is highlighted:

$$f = \Sigma(0, 3, 5, 6)$$

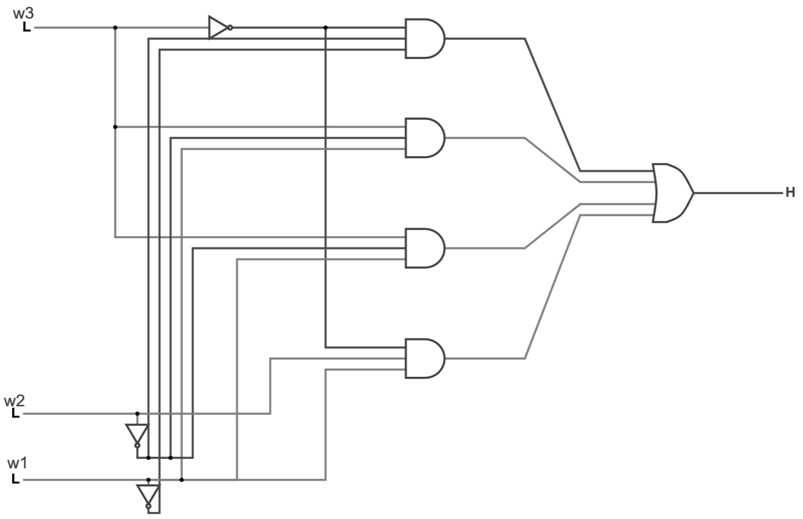
From expanding:

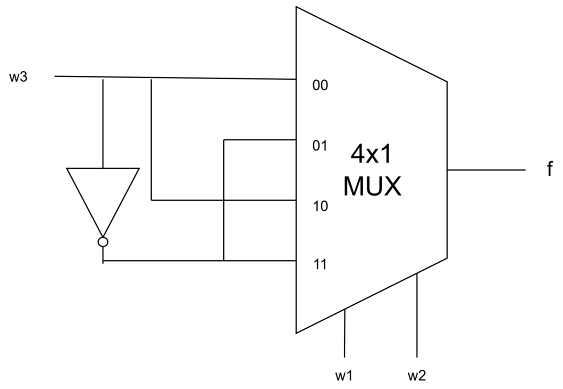
$$f = \overline{w1\,w2}\,\overline{w3} + \overline{w1}\,w2\,w3 + w1\,\overline{w2}\,w3 + w1\,w2\,\overline{w3}$$

Which can be simplified into a 3 input **XNOR** gate.

$$f = \overline{w1 \oplus w2 \oplus w3}$$

3.1.2 Logic for f:

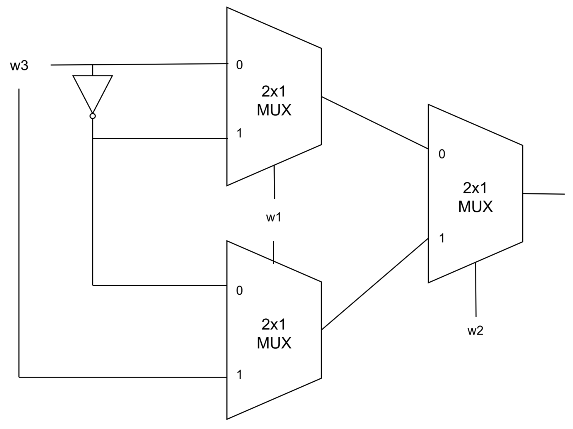


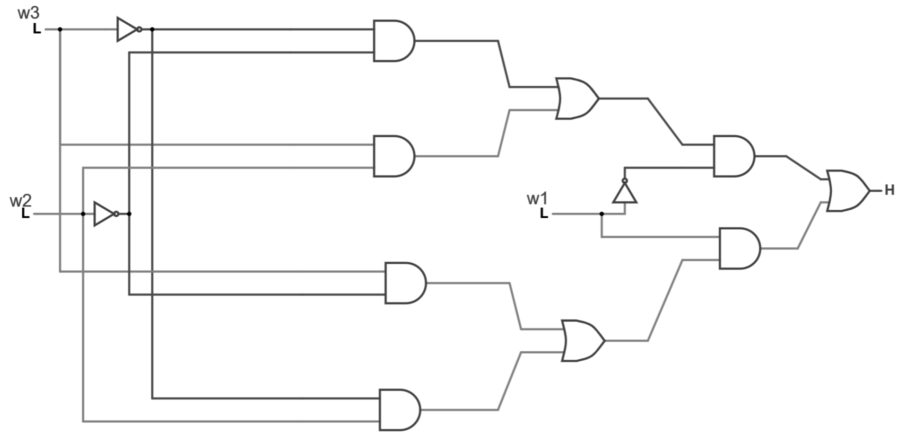


3.1.3 Here's the simple procedure:

1. The two inputs, w1 and w2, serve as the "select" lines for the multiplexer.
2. This splits the problem into four smaller cases (one for each data input: i0, i1, i2, i3).
3. For each case, the original truth table was examined to see how the third input (w3) affected the final answer.
4. Then, connected each data input of the MUX to w3 or its inverse (~w3) to match the logic for that specific case, ensuring the MUX always outputs the correct answer “f”.

3.1.4 Logic diagram for f:



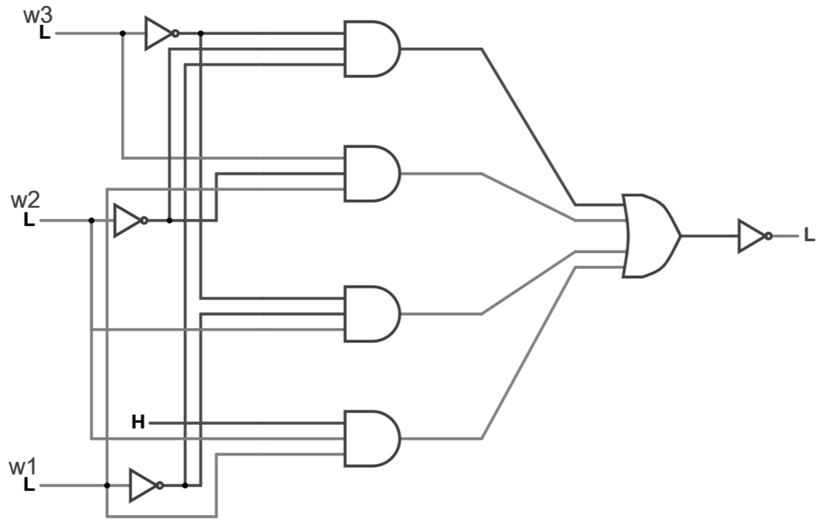


1) Implementing the 3-input function f using 2x1 multiplexers involves creating a two-level "tree" structure, a process formally known as Shannon expansion. The final stage consists of a 2x1 MUX (let's call it MUX C) that uses one input, w1, as its main select line; the output of this MUX produces the final function f. The two data inputs of this MUX, I0 and I1, must then be set to solve the remaining logic based on the value of w1.
2) The I0 input must provide the correct answer for the function *assuming w1 is 0* (which, based on the truth table, is the function w2 XNOR w3), while the I1 input must provide the answer *assuming w1 is 1* (the function w2 XOR w3). To supply these two "sub-functions," a first stage of MUXes is utilized. A second MUX (MUX A) builds the w2 XNOR w3 function by using w2 as its select line, with ~w3 connected to its I0 input and w3 to its I1 input.
3) A third MUX (MUX B) builds the w2 XOR w3 function, also using w2 as its select line, but with w3 connected to its I0 and ~w3 to its I1. Finally, the output of MUX A is wired to the I0 input of MUX C, and the output of MUX B is wired to the I1 input of MUX C, completing the full logic implementation.

3.2 Function g

w1	w2	w3	g
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

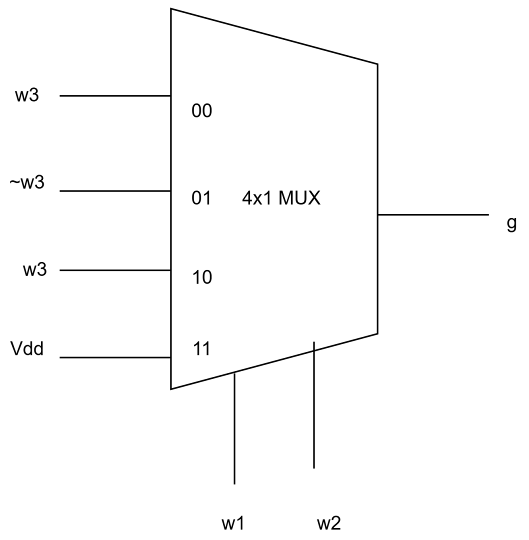
3.2.1 Logic for g:



s1 (w2)	s0 (w1)	MUX Input	Function Rows (w1, w2, w3)	Required Output g	Data Input Connection
0	0	I0	(0, 0, 0) (0, 0, 1)	0 1	g = w3
0	1	I1	(1, 0, 0) (1, 0, 1)	0 1	g = w3
1	0	I2	(0, 1, 0) (0, 1, 1)	1 0	g = ~w3
1	1	I3	(1, 1, 0) (1, 1, 1)	1 1	g = 1 (High)

Select Lines	Binary Value	Data Input Selected
S1 = 0, S0 = 0	00	D0 (or i0)
S1 = 0, S0 = 1	01	D2 (or i1)
S1 = 1, S0 = 0	10	D2 (or i2)
S1 = 1, S0 = 1	11	D3 (or i3)

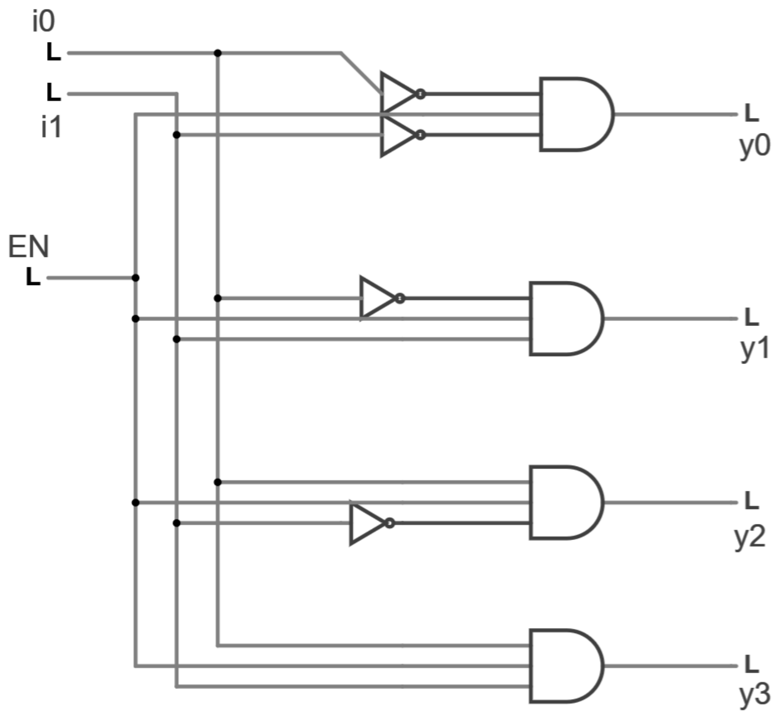
```
module function_g_2to1mux (
    input  logic w1, w2, w3,
    output logic g
);

    logic i0, i1;

    assign i0 = w3;
    assign i1 = w1 | (~w3);
    assign g = (~w2 & i0) | (w2 & i1);

Endmodule

module tb_function_g_2to1mux;

    logic w1, w2, w3;

    wire g;

    function_g_2to1mux dut (
        .w1(w1),
        .w2(w2),
        .w3(w3),
        .g(g)
    );

    logic [2:0] inputs;
    logic expected_g;

    initial begin
        $display("--- Testing Function g (2-to-1 MUX) ---");
        $display("Time | w1 | w2 | w3 | g (Output) | Expected");
        $monitor("%4t | %b  | %b  | %b  |     %b      |     %b",
                 $time, w1, w2, w3, g, expected_g);

        for (int i = 0; i < 8; i = i + 1) begin
            inputs = i;
            {w1, w2, w3} = inputs;
        case (inputs)
            3'b000: expected_g = 1'b0;
            3'b001: expected_g = 1'b1;
            3'b010: expected_g = 1'b1;
            3'b011: expected_g = 1'b0; // Official value from lab doc
            3'b100: expected_g = 1'b0;
            3'b101: expected_g = 1'b1;
            3'b110: expected_g = 1'b1;
            3'b111: expected_g = 1'b1;
            default: expected_g = 1'bx;
        endcase

        #10; // Wait 10 time units
    end

    $display("\n--- Testbench Finished ---");
    $finish;
  end

endmodule
```

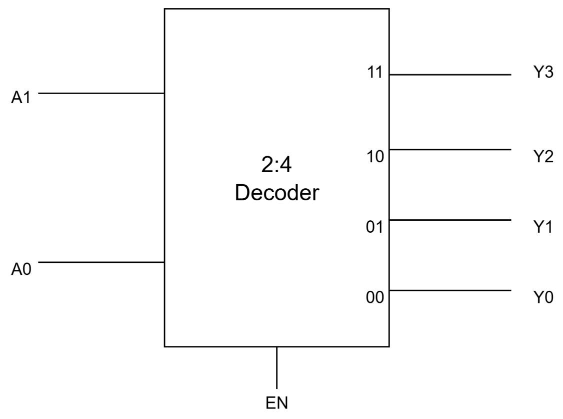


4. Decoder

4.1. 2:4 Decoder with EN signal:

4.1.1 Logic of 2:4 Decoder with EN signal





4.1.2 Truth Table for 2:4 Decoder with EN signal

EN	$A_1(i_1)$	$A_0(i_0)$	Y_3	Y_2	Y_1	Y_0
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

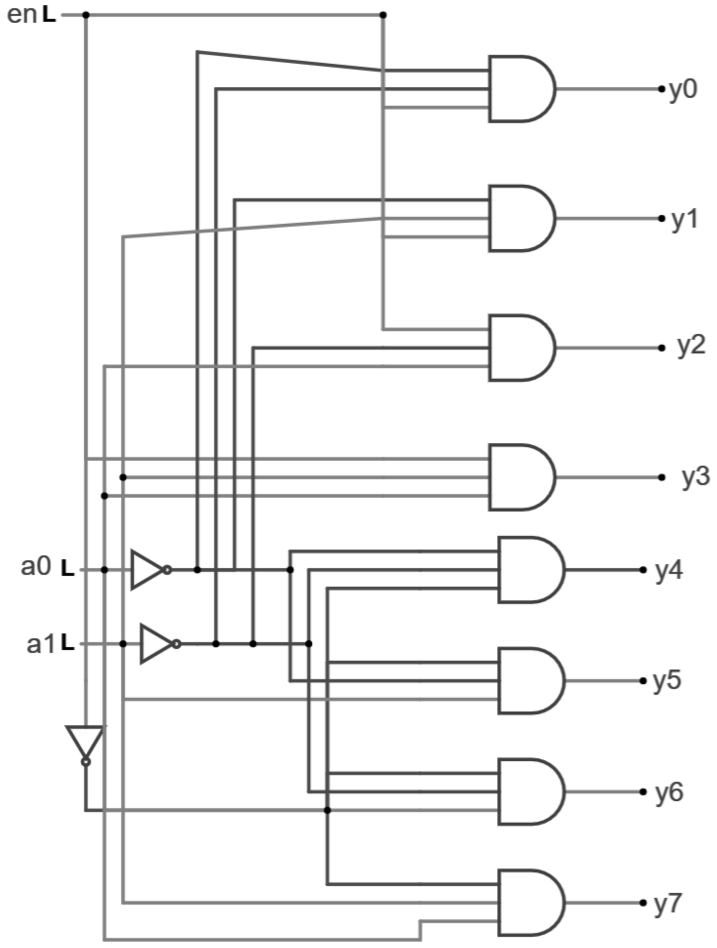
4.1.3.SystemVerilog Code for 2:4 Decoder:

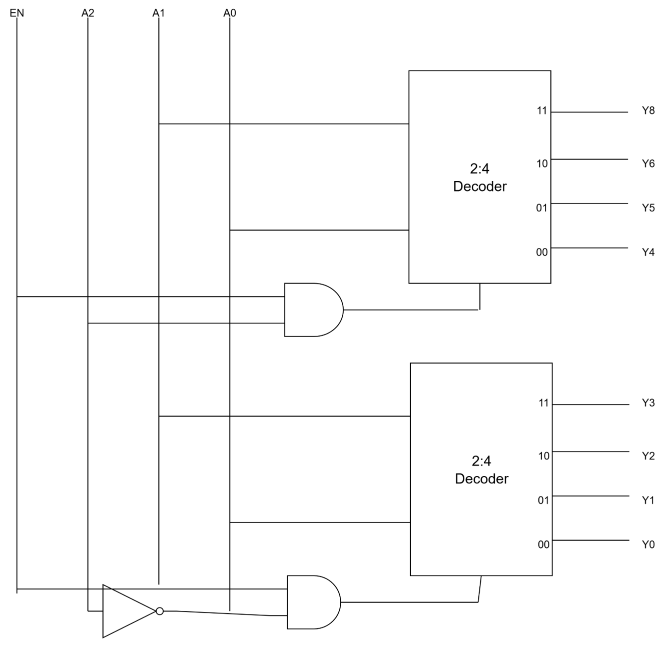
```
module decoder_2to4 (
    input  logic [1:0] in,
    input  logic       en,
    output logic [3:0] y_out
);

    always_comb begin
        if (en) begin
            case (in)
                2'b00:    y_out = 4'b0001; // Select y_out[0]
                2'b01:    y_out = 4'b0010; // Select y_out[1]
                2'b10:    y_out = 4'b0100; // Select y_out[2]
                2'b11:    y_out = 4'b1000; // Select y_out[3]
                default:  y_out = 4'b0000; // Handle 'x' or 'z' inputs
            endcase
        end else begin
            y_out = 4'b0000;
        end
    end

endmodule
```

4.1.4. Testbench for 2:4 decoder:

```
`timescale 1ns / 1ps
module tb_decoder_2to4;

    logic [1:0] in_tb;
    logic       en_tb;


    wire  [3:0] y_out_tb;

    decoder_2to4 dut (
        .in(in_tb),
        .en(en_tb),
        .y_out(y_out_tb)
    );


    initial begin
        $display("--- 2:4 Decoder Testbench ---");
        $display("Time | En | In | Y_Out");
        $display("----------------------------");

        $display("--- Test Case 1: Enable OFF ---");
        en_tb = 1'b0;
        in_tb = 2'b00;
        #10;
        in_tb = 2'b10;
        #10;
        in_tb = 2'b11;
        #10;
        $display("--- Test Case 2: Enable ON ---");
        en_tb = 1'b1;

        for (int i = 0; i < 4; i = i + 1) begin
            in_tb = i;
            #10;
        end

        $display("----------------------------");
        $display("--- Testbench Finished ---");
        $finish;
    end

endmodule
```

4.2. 3:8 Decoder using 2:4 Decoders:
4.2.1 Logic





4.2.2 SystemVerilog code for 3:8 decoder using 2:4 decoders:

```
module decoder_3to8 (
   input  logic [1:0] in,
   input  logic       en,
   output logic [7:0] y_out
);
   logic en_low;
   logic en_high;
   logic [3:0] y_low;
   logic [3:0] y_high;

   assign en_low  = en & ~in[2];
```

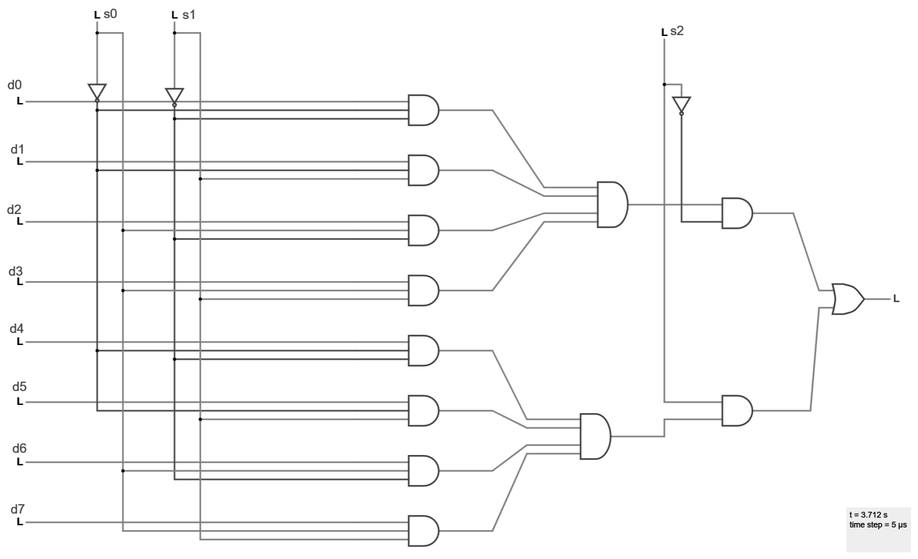
```
    assign en_high = en &  in[2];


    decoder_2to4 dec_low (
        .in(in[1:0]),
        .en(en_low),
        .y_out(y_low)
    );


    decoder_2to4 dec_high (
        .in(in[1:0]),
        .en(en_high),
        .y_out(y_high)
    );
    assign y_out = {y_high, y_low};

endmodule
```

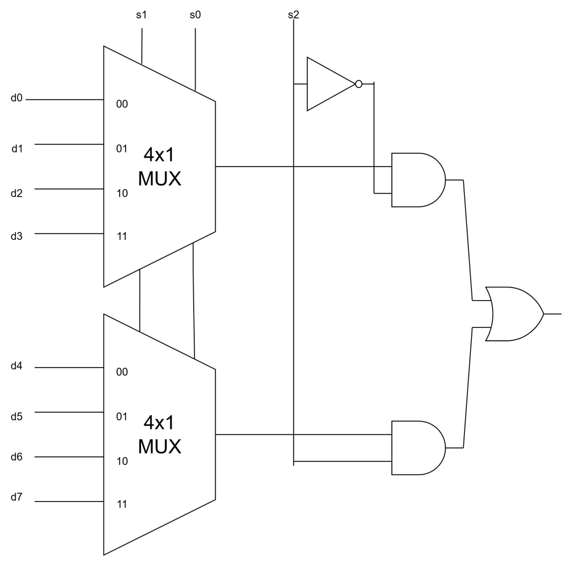
4.2.3 Testbench for 3:8 multiplexer:

```
module tb_decoder_3to8;

    logic [2:0] in_tb;
    logic       en_tb;
    wire  [7:0] y_out_tb;

    decoder_3to8 dut (
        .in(in_tb),
        .en(en_tb),
        .y_out(y_out_tb)
    );

    initial begin
        $display("--- 3-to-8 Decoder Testbench ---");
        $display("Time | En | In  | Y_Out");
        $display("---------------------------------------");


        $monitor("%4t | %b  | %b | %b", $time, en_tb, in_tb, y_out_tb);
        $display("--- Test Case 1: Enable OFF ---");
        en_tb = 1'b0;


        in_tb = 3'b000;
        #10;
        in_tb = 3'b101;
        #10;
        in_tb = 3'b111;
        #10;

        $display("--- Test Case 2: Enable ON ---");
        en_tb = 1'b1;


        for (int i = 0; i < 8; i = i + 1) begin
            in_tb = i;
            #10;
        end

        $display("--------------------------------------");
        $display("--- Testbench Finished ---");
        $finish;
    end

endmodule
```

5. Logic Function:

5.1 8x1 Multiplexer:





5.1.1 SystemVerilog module for 4x1 multiplexer:

```
module mux_4to1 (
   input logic [1:0] sel,
   input logic d0, d1, d2, d3,
   output logic y
);

   assign y = sel[1] ? (sel[0] ? d3 : d2) : (sel[0] ? d1 : d0);

endmodule
```

5.1.2 SystemVerilog module for 8x1 multiplexer:

```
module mux_8to1 (
   input logic s2, s1, s0,
   input logic d0, d1, d2, d3, d4, d5, d6, d7,
   output logic y
);

   logic [1:0] sel_lsb = {s1, s0};
   logic y_mux_top;
   logic y_mux_bottom;
   logic s2_n;
   logic y_and_top;
   logic y_and_bottom;
```

```
    assign s2_n = ~s2;

    mux_4to1 top_mux (
        .sel(sel_lsb),
        .d0(d0), .d1(d1), .d2(d2), .d3(d3),
        .y(y_mux_top)
    );

    mux_4to1 bottom_mux (
        .sel(sel_lsb),
        .d0(d4), .d1(d5), .d2(d6), .d3(d7),
        .y(y_mux_bottom)
    );

    assign y_and_top = y_mux_top & s2_n;

    assign y_and_bottom = y_mux_bottom & s2;

    assign y = y_and_top | y_and_bottom;

endmodule
```

5.1.3 Testbench for the 8x1 multiplexer module:

```
module mux_8to1_tb;

    logic s2, s1, s0;
    logic d0, d1, d2, d3, d4, d5, d6, d7;
    logic y;

    mux_8to1 mux (.*);

    initial begin
        s2 = 0; s1 = 0; s0 = 0;
        d0 = 1; d1 = 0; d2 = 1; d3 = 0;
        d4 = 0; d5 = 1; d6 = 0; d7 = 1;

        for (int i = 0; i < 8; i++) begin
            {s2, s1, s0} = i;
            #10;
            logic expected_y;
            case (i)
                0: expected_y = d0;
                1: expected_y = d1;
                2: expected_y = d2;
                3: expected_y = d3;
                4: expected_y = d4;
                5: expected_y = d5;
                6: expected_y = d6;
                7: expected_y = d7;
            endcase
        end
        $finish;

    end

endmodule
```
[

5.2 Decoder and Multiplexer Function:

$$F = (A, B, C, D, E) = \Sigma\,(0, 1, 2, 4, 1, 11, 16, 17, 18, 19, 25, 28, 29)$$

We are given function in the sum of products for. Via Karnaugh maps the equation can be simplified to:

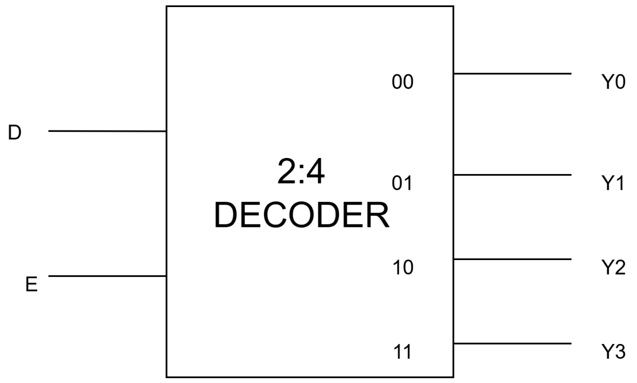
$$F = \overline{B\,C} + \overline{A}\,\overline{C}\,D + A\,\overline{C}\,\overline{D}\,E + A\,B\,C\,\overline{D} + \overline{A}\,\overline{B}\,\overline{D}\,\overline{E}$$

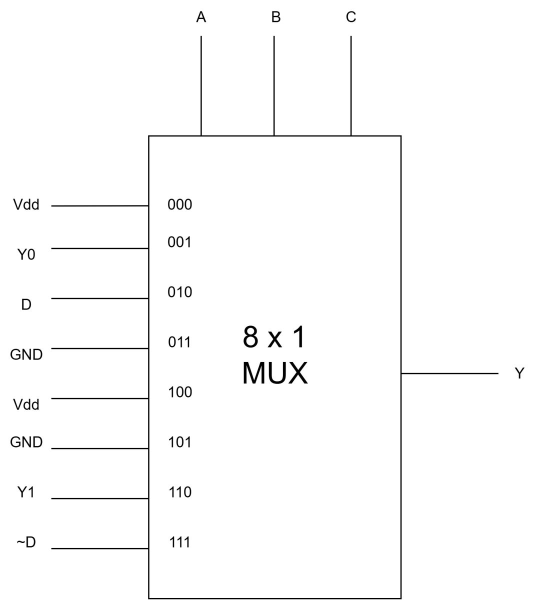
An 8-to-1 MUX requires 3 select lines, and a 2-to-4 Decoder requires 2 select lines. Since we have 5 input variables: A, B, C, D, E

- We'll use three variables as the select lines S_0, S_1 for the 8-to-1 MUX. Let's choose the three most significant bits (MSBs): A, B, C
- The remaining two variables D, E will be used to control the 2-to-4 Decoder.

We implemented the 5-variable logic function $F = (A, B, C, D, E)$ using only a single 8-to-1 Multiplexer (MUX) and a single 2-to-4 Decoder. We used the three MSBs A, B, C as the select lines for the MUX. This left the two LSBs, D and E. We connected D and E to the select lines of the 2-to-4 Decoder to generate the four minterms ($\overline{DE}$, $\overline{D}E$, $D\overline{E}$, DE). By analyzing the original function's minterms grouped by A, B, C, we determined that each MUX data input I0 through I7 could be connected directly to one of the Decoder outputs or one of the available inputs (D, $\overline{D}$, 0, 1), thus satisfying the constraint of using no additional logic gates.

5.2.1 Logic Diagrams for F





I_K	ABC	Min	$DE = 00$	$DE = 01$	$DE = 10$	$DE = 11$	Required Logic
I_0	0	0,1,2,3	1	1	1	1	1
I_1	1	4	1	0	0	0	$\overline{DE}$
I_2	10	10,11	0	0	1	1	D
I_3	11	X	0	0	0	0	0
I_4	100	16,17,18,19	1	1	1	1	1
I_5	101	25	0	1	0	0	$\overline{D}E$
I_6	110	28	0	0	1	0	$D\overline{E}$
I_7	111	29	0	0	0	1	DE

5.2.2 System Verilog Codes

```
module decoder_2to4 (
   input logic S1, S0,
   output logic [3:0] Y
);
   assign Y[0] = ~S1 & ~S0;
   assign Y[1] = ~S1 & S0;
```

```
    assign Y[2] = S1 & ~S0;
    assign Y[3] = S1 & S0;
endmodule

module mux_8to1 (
    input logic [2:0] S,
    input logic [7:0] I,
    output logic F
);
    always_comb begin
        case (S)
            3'b000: F = I[0];
            3'b001: F = I[1];
            3'b010: F = I[2];
            3'b011: F = I[3];
            3'b100: F = I[4];
            3'b101: F = I[5];
            3'b110: F = I[6];
            3'b111: F = I[7];
            default: F = 1'bx;
        endcase
    end
endmodule

module F_function (
    input logic A, B, C, D, E,
    output logic F
);
    logic [3:0] decoder_out;
    logic [7:0] mux_in;

    logic [2:0] mux_select = {A, B, C};


    decoder_2to4 decoder (
        .S1(D),
        .S0(E),
        .Y(decoder_out)
    );
    assign mux_in[0] = 1'b1;

    assign mux_in[1] = decoder_out[0];

    assign mux_in[2] = D;

    assign mux_in[3] = 1'b0;

    assign mux_in[4] = 1'b1;

    assign mux_in[5] = decoder_out[1];

    assign mux_in[6] = decoder_out[2];

    assign mux_in[7] = decoder_out[3];

    mux_8to1 mux (
        .S(mux_select),
        .I(mux_in),
        .F(F)
    );

endmodule
```