Lab Report 3

Dr. Aly

Hector Garibay,

Justin Wong

Due Date: July 2, 2025

Introduction:

The goal of this lab was to implement a 16x1 multiplexer in Verilog using gate-level 2x1 multiplexers. We will be able to change the value of the MUX select lines through the pushbuttons on the board, which will require debouncing and toggle logic. Once we set a value for the select lines and the inputs via the switches, we will see the output of the selected input on LED0.

Code and Explanation:

```
module mux2x1(

input a, b,

input sel,

output y

);

// gate level logic: y = a~sel + bsel

wire nsel, a1, b1; // declare wires for ~sel, input 1, and input 2

not (nsel, sel); // ~sel

and (a1, a, nsel); // a1 = a~sel

and (b1, b, sel); // b1 = bsel

or (y, a1, b1); // y = a1 + b1

endmodule
```

For the implementation of the 2x1 mux, gate level logic was used. From the truth table and kmap of the 2x1 mux, we can derive the expression y = asel' + bsel. Using and, not, and or gates, we constructed the Boolean expression for the 2x1 mux to build our 16x1 mux.

```
module mux16x1(
input [15:0] in,
input [3:0] sel,
output out
);
```

```
module mux16x1(
input [15:0] in,
input [3:0] sel,
output out
);
/* 16x1 mux requires 4 levels of 2x1 muxes: lvl 1 : 8, lvl 2: 4, lvl 3: 2, lvl 4: 1*/
wire [15:0] level1;
wire [7:0] level2;
wire [3:0] level3;
genvar i;
generate
for (i = 0; i < 8; i = i + 1)
mux2x1 m1 (.a(in[2*i]), .b(in[2*i+1]), .sel(sel[0]), .y(level1[i])); // instantiate 8 2x1 mux for level 1
for (i = 0; i < 4; i = i + 1)
mux2x1 m2 (.a(level1[2*i]), .b(level1[2*i+1]), .sel(sel[1]), .y(level2[i])); // instantiate 4 2x1 mux for level 2
for (i = 0; i < 2; i = i + 1)
mux2x1 m3 (.a(level2[2*i]), .b(level2[2*i+1]), .sel(sel[2]), .y(level3[i])); // instantiate 2 2x1 mux for level 3

mux2x1 m4 (.a(level3[0]), .b(level3[1]), .sel(sel[3]), .y(out)); // instantiate 1 2x1 mux for last level
endgenerate

endmodule
```

Generating a 16x1 mux requires 15 2x1 mux. A mux has 2^n inputs where n is the number of select lines; this means that a 16x1 mux will require 4 select lines. There will be 4 levels of 2x1 mux, level 1 with 8, level 2 with 4, level 3 with 2, and level 4 with 1. Each will have their own select line. Here, we used a generate for loop with loop variable I to instantiate multiple 2x1 mux for each level.

```
module debounce (
input clk,
input btn_in,
output reg btn_clean
);
reg [2:0] shift_reg; // 3 bit shift register
always @(posedge clk) begin
shift_reg <= {shift_reg[1:0], btn_in}; // on posedge of clock, feed button input to lsb of shift reg
if (shift_reg == 3'b111) btn_clean <= 1; // once shift reg is 111, button input is clean and set to 1
else if (shift_reg == 3'b000) btn_clean <= 0; // once shift reg is 000, button input is set to 0
end
endmodule
```

We need the debounce module for the pushbuttons since they do not generate clean signals; they bounce between 0 and 1 for a short time causing unintended triggers. This module creates a 3 bit shift register, with the button input being fed into the lsb of the shift register on every positive clock edge. Once the bits are either all 0 or all 1, the button is clean and is registered.

```
module toggle_switch (
input clk, // System clock
input rst, // Reset signal
input btn_raw, // Raw button input
output reg state // Output toggle state
);

wire btn_clean; // Debounced button signal
reg btn_prev; // Stores previous button state

// Debounce the raw button input
debounce db (
.clk(clk),
.btn_in(btn_raw),
.btn_clean(btn_clean)
);
always @(posedge clk) begin
if (rst) begin
state <= 0; // Reset the output state
btn_prev <= 0; // Clear previous button state
end else begin
// If button is pressed now and wasn't pressed before
if (btn_clean && !btn_prev)
state <= ~state; // Toggle the state
btn_prev <= btn_clean; // Update previous button state
end
end

endmodule
```

To implement toggle logic, we created a module that models a toggle flip flop. It takes in a clock, a reset signal, and the button input. In order to toggle the state, we need to check if the button is pressed, as well as if the button was not previously pressed. If these two conditions are met, then we toggle the state and update the previous state.

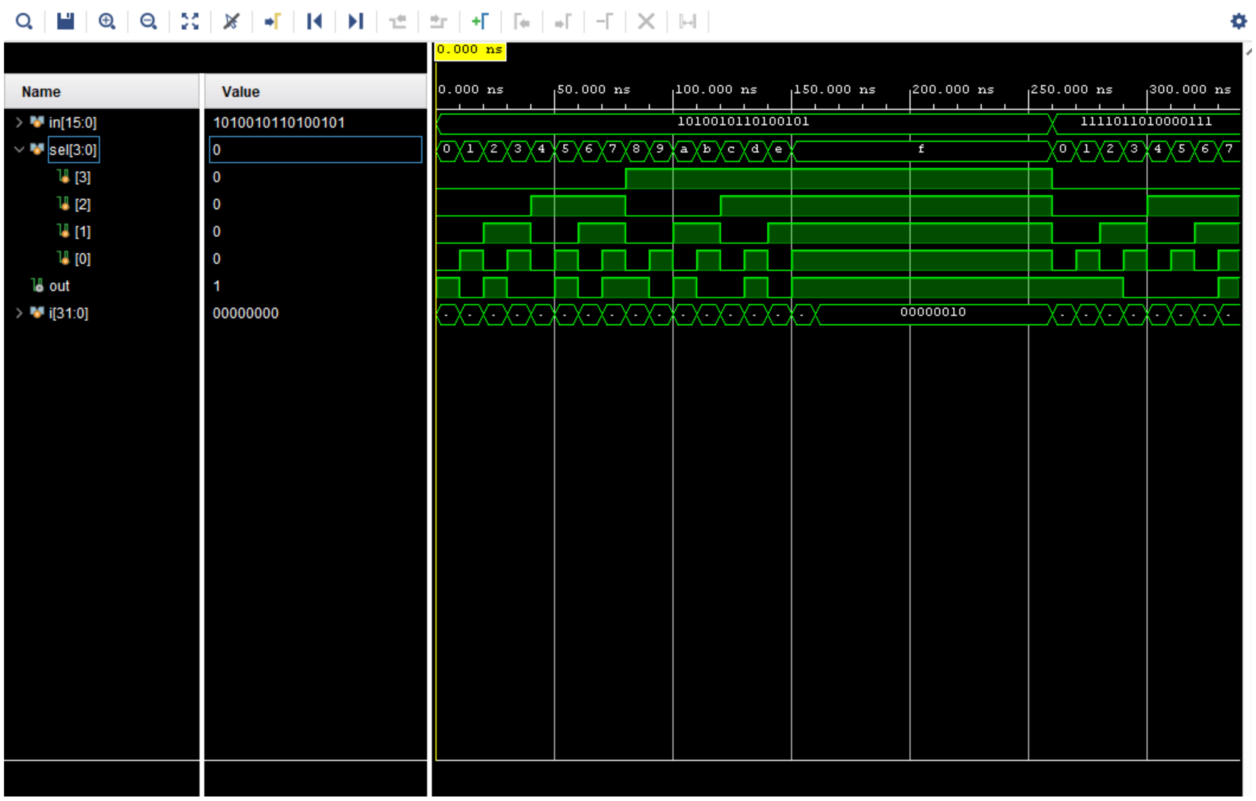
```
module top_mux_lab3 (
input clk,
input rst,
input [15:0] SW,
input btnU, btnD, btnL, btnR,
output LED0
);
wire [3:0] sel; // select lines
```

```
// Toggle switches to control each bit of the select line using pushbuttons
toggle_switch t0 (.clk(clk), .rst(rst), .btn_raw(btnD), .state(sel[0]));
toggle_switch t1 (.clk(clk), .rst(rst), .btn_raw(btnR), .state(sel[1]));
toggle_switch t2 (.clk(clk), .rst(rst), .btn_raw(btnL), .state(sel[2]));
toggle_switch t3 (.clk(clk), .rst(rst), .btn_raw(btnU), .state(sel[3]));
// 16x1 multiplexer: selects one of the 16 switches based on sel value
mux16x1 mux (.in(SW), .sel(sel), .out(LED0));
endmodule
```

This is the top module for lab 3. Here we instantiate the toggle switches within the top module where button U is the msb, and button D being the lsb. These are to control the select lines and get our desired output. We instantiate our 16x1 mux module and set our output to LED0.

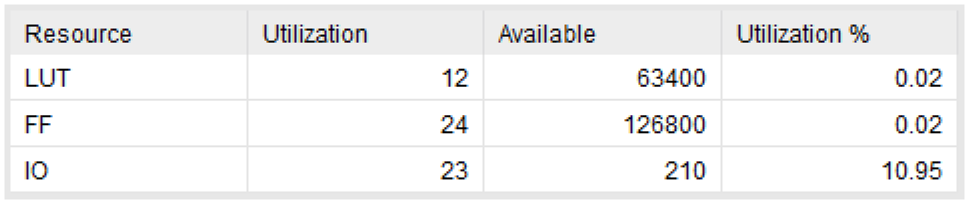
```
module tb_mux16x1;
reg [15:0] in;
reg [3:0] sel;
wire out;
integer i;

mux16x1 test (.in(in), .sel(sel), .out(out));
initial begin
in = 16'b1010_0101_1010_0101; // case 1: A5A5
for (i = 0; i<16; i = i+1)
begin
sel = i; // loop 0-15; set select line to i each iteration
#10; // 10 ns delay
end
#100
in = 16'b1111_0110_1000_0111; // case 2: F687
for (i = 0; i<16; i = i+1)
begin
sel = i;
#10;
end
#100
in = 16'b0110_1100_0101_1110; // case 3: 6C5E
for (i = 0; i<16; i = i+1)
begin
sel = i;
#10;
end
$finish;
end
endmodule
```

This is our testbench module where we tried three inputs for the 16x1 mux. Once instantiating our 16x1 mux module, we tested values A5A5, F687, and 6C5E. By using a for loop and setting the select lines to I, we were able to iterate through all 15 select line values and view the output at LED0. The figure below shows the inputs and outputs for the first test case:



Name	Value
in[15:0]	1010010110100101
sel[3:0]	0
[3]	0
[2]	0
[1]	0
[0]	0
out	1
i[31:0]	00000000

Utilizations:



Resource	Utilization	Available	Utilization %
LUT	12	63400	0.02
FF	24	126800	0.02
IO	23	210	10.95

Contributions:

Hector Garibay: Code/Report 50%

Justin Wong: Code/Report 50%

Video Link:

https://youtu.be/Mj6DLHa6oic