CMPE 124 LAB 1 Multiplexers

Muhammad Anas

# Introduction

Abstract— This lab required us to use primitive logic gates to build multiplexers using EDAPlayground. There were two parts to this lab. Part one consisted of 2-1 MUX. Part two consisted of 3-1 MUX.

# Description of the circuit schematics

## Part 1

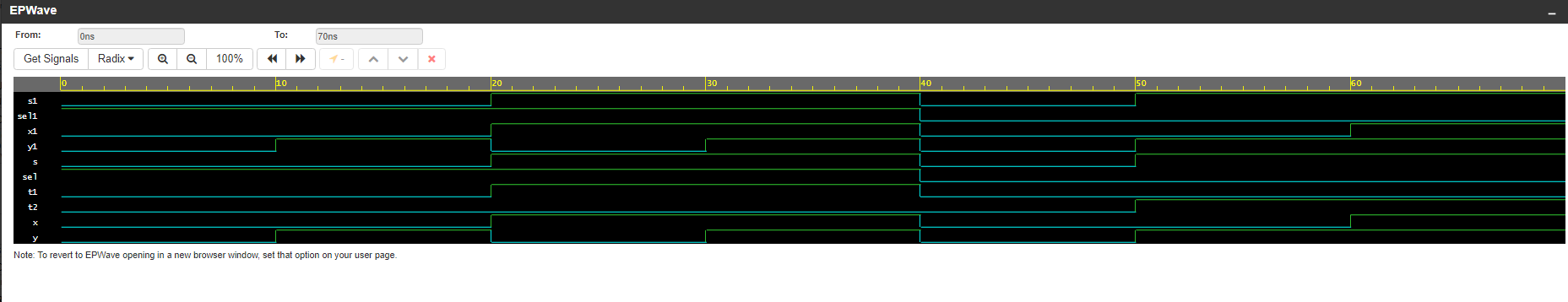
In part 1 of the lab, we had to design a multiplexer using 2 input AND and OR gates. For this we first apply logic levels (logic 1/logic 0) to the select, A and B inputs, therefore satisfying every entry of the truth table, we then further recorded the output values.

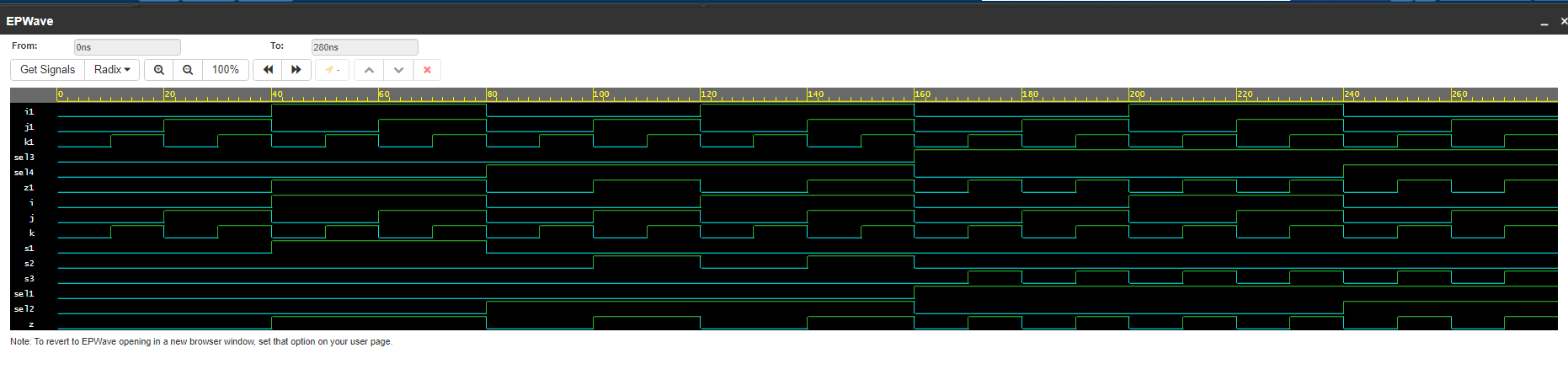
## Part 2

In part 2 of this lab we did the same thing as part 1, however, this time we had to design 3-1 MUX using 2 and 3-input AND and OR gates. There are three inputs, A, B, C, and it contains two select lines, which are sel1 and sel2.

# Timing diagram(s)

Part 1: (Zoom in to view)



Part 2: (Zoom in to view)

# Verification Table:



# IV.Verification Table (Cont.):



# DISCUSSION OF THE RESULTS

If the output produced were different from the produced output this would mean that we made a mistake in one of our values for our truth table. We would have to go back and go through the values to verify they are the same. This will then produce the same output as the expected output.

# OVerall learning experience

This lab was interesting as we got the chance to implement logic and see it function practically. The difficult part was definitely creating the truth tables, especially for the second part. Writing out each line, line by line and verifying for correctness took a lot of time so it was a bit tedious.

# Verilog code of the module(s)

Part1:

**Design:**

module mux(s, x, y, sel);

output s;

input x,y,sel;

wire t1, t2;

and(t1,x,sel);

and(t2,y,~sel);

or(s,t1, t2);

endmodule

**Testbench:**

module mux\_tb;

wire s1;

reg x1, y1, sel1;

mux p1 (.s(s1), .x(x1), .y(y1), .sel(sel1));

initial begin

x1=0; y1=0; sel1=1;

#10 x1=0; y1=1; sel1=1;

#10 x1=1; y1=0; sel1=1;

#10 x1=1; y1=1; sel1=1;

#10 x1=0; y1=0; sel1=0;

#10 x1=0; y1=1; sel1=0;

#10 x1=1; y1=1; sel1=0;

#10 x1=1; y1=0; sel1=0;

end

initial begin

$dumpfile("new.vcd");

$dumpvars;

end

endmodule

Part 2:

**Design:**

module mux2(z, i, j, k, sel1, sel2);

output z;

input i,j,k, sel1, sel2;

wire s1, s2, s3;

and(s1, ~sel1, ~sel2, i);

and(s2,~sel1, sel2, j);

and(s3,sel1, k);

or(z, s1, s2, s3);

endmodule

**Testbench:**

module mux2\_tb;

wire z1;

reg i1, j1, k1, sel3, sel4;

mux2 t1 (.z(z1), .sel1(sel3), .sel2(sel4), .i(i1), .j(j1), .k(k1));

initial begin

sel3=0; sel4=0; i1=0; j1=0; k1=0;

#10 sel3=0; sel4=0; i1=0; j1=0; k1=1;

#10 sel3=0; sel4=0; i1=0; j1=1; k1=0;

#10 sel3=0; sel4=0; i1=0; j1=1; k1=1;

#10 sel3=0; sel4=0; i1=1; j1=0; k1=0;

#10 sel3=0; sel4=0; i1=1; j1=0; k1=1;

#10 sel3=0; sel4=0; i1=1; j1=1; k1=0;

#10 sel3=0; sel4=0; i1=1; j1=1; k1=1;

#10 sel3=0; sel4=1; i1=0; j1=0; k1=0;

#10 sel3=0; sel4=1; i1=0; j1=0; k1=1;

#10 sel3=0; sel4=1; i1=0; j1=1; k1=0;

#10 sel3=0; sel4=1; i1=0; j1=1; k1=1;

#10 sel3=0; sel4=1; i1=1; j1=0; k1=0;

#10 sel3=0; sel4=1; i1=1; j1=0; k1=1;

#10 sel3=0; sel4=1; i1=1; j1=1; k1=0;

#10 sel3=0; sel4=1; i1=1; j1=1; k1=1;

#10 sel3=1; sel4=0; i1=0; j1=0; k1=0;

#10 sel3=1; sel4=0; i1=0; j1=0; k1=1;

#10 sel3=1; sel4=0; i1=0; j1=1; k1=0;

#10 sel3=1; sel4=0; i1=0; j1=1; k1=1;

#10 sel3=1; sel4=0; i1=1; j1=0; k1=0;

#10 sel3=1; sel4=0; i1=1; j1=0; k1=1;

#10 sel3=1; sel4=0; i1=1; j1=1; k1=0;

#10 sel3=1; sel4=0; i1=1; j1=1; k1=1;

#10 sel3=1; sel4=1; i1=0; j1=0; k1=0;

#10 sel3=1; sel4=1; i1=0; j1=0; k1=1;

#10 sel3=1; sel4=1; i1=0; j1=1; k1=0;

#10 sel3=1; sel4=1; i1=0; j1=1; k1=1;

#10 sel3=1; sel4=1; i1=1; j1=0; k1=0;

end

initial begin

$dumpfile("new.vcd");

$dumpvars;

end

endmodule

# Conclusion

Overall, I enjoyed this lab as it helped build a foundation for the other labs to come. I am still getting used to structural Verilog, but it was a bit different compared to other forms of programming languages. It was very interesting to see the primitive logic gates work practically as compared to when we just work them out by hand. I believe this is great steppingstone to understanding how to design complex logic.