Lab 3: Multiplexers and Full-Adder

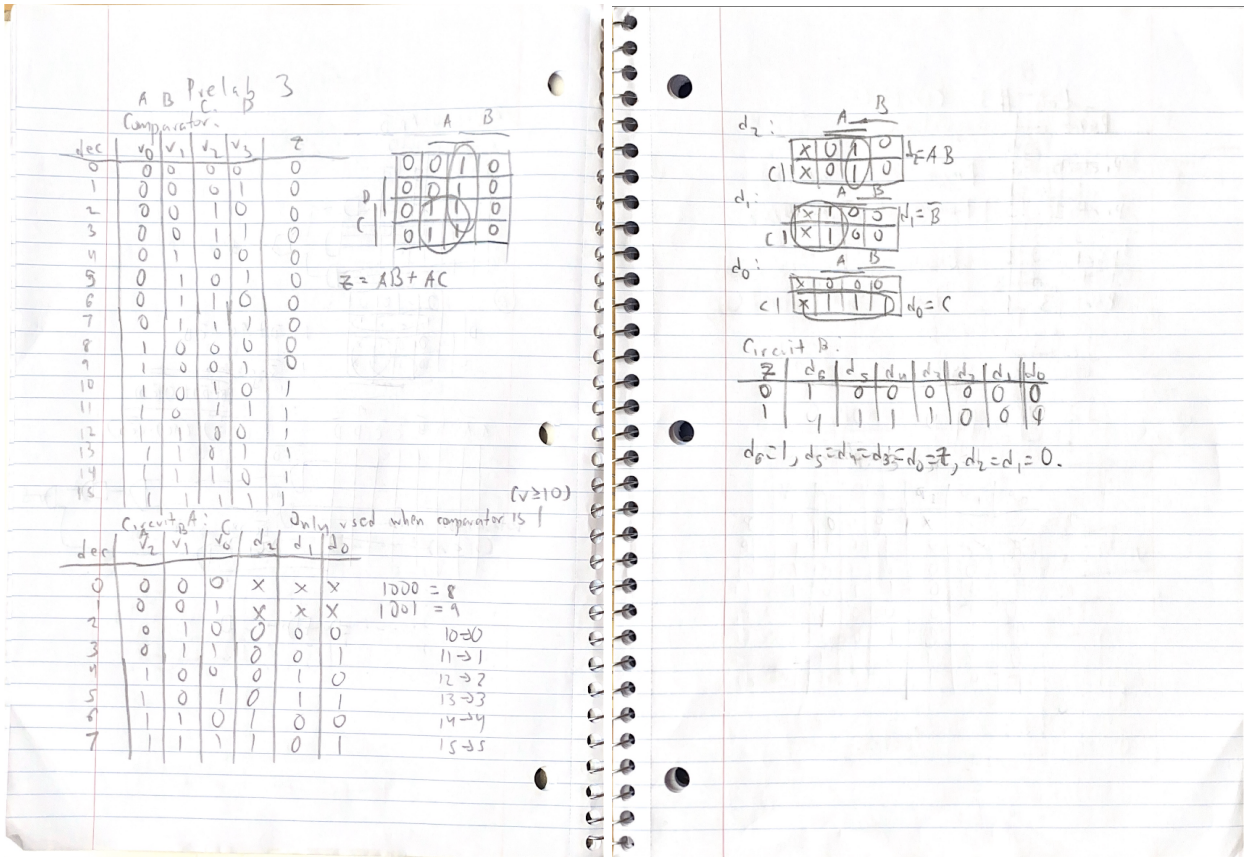
Eric Liu

Objectives

In this lab, our goal is to create a counter made of two 7-segment displays given a 4-bit binary number input. In part one, we will first create a comparator, “subtractor by 10”, and 0 or 1 7-segment display driver, and connect them together with multiplexers to create a working counter. In part two, we will create a cascading 3-bit full adder, and in part three, we will glue these parts together to create a counter that can take 2 3-bit numbers and a carry and display the result of them added together.

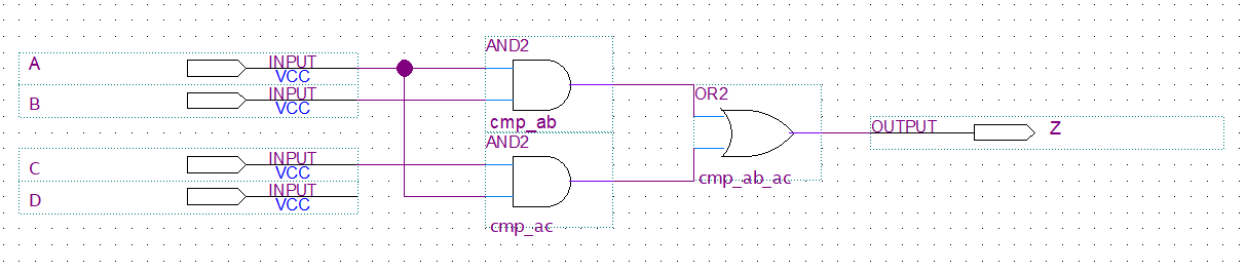
Design and Test Procedure

Prelab:



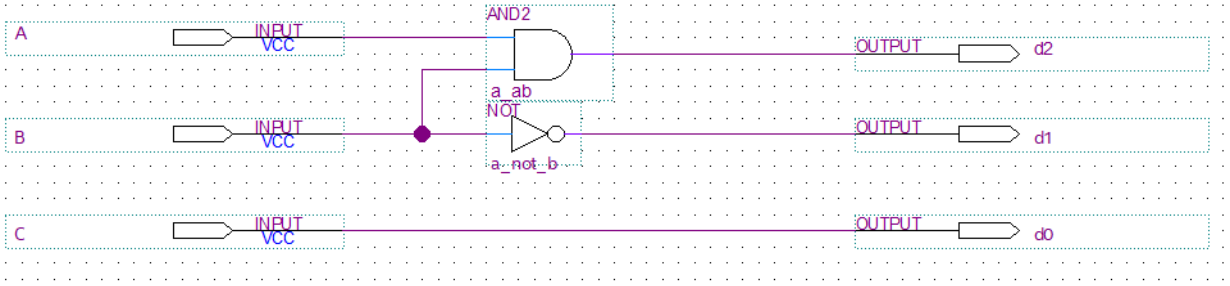
This details the formulas used to create the schematics in Quartus below. Explanations for the purposes of each circuit are also explained below the schematics.

Comparator:



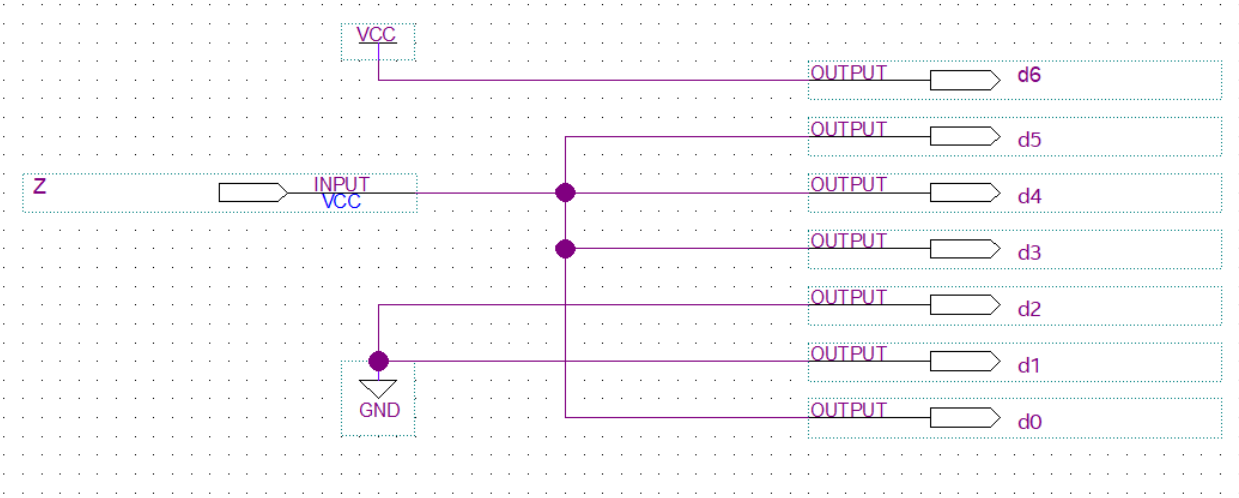
This circuit simply takes a 4-bit binary number and checks if it's greater than 9. We will use this in the top level as a mux input as well as the input to circuit B, which will effectively toggle on circuit A and switch the "tens place" 7-segment display to 1.2

Part 1:
Circuit A:



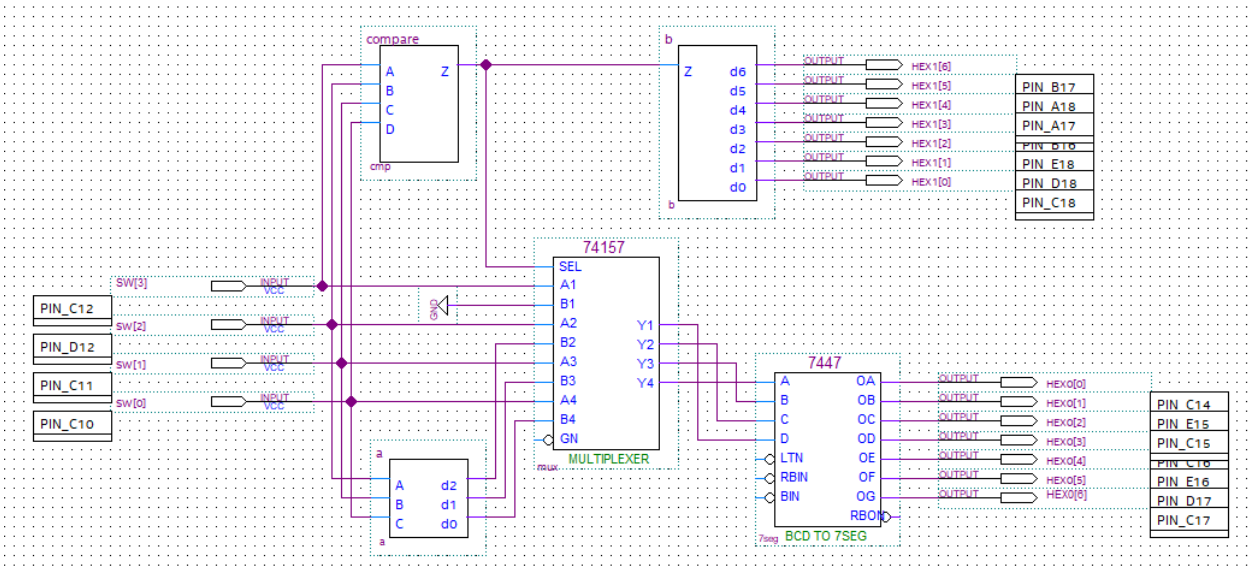
This circuit is turned on in the top-level design when the comparator sees the input number is greater than 9. It effectively translates numbers from the range [10, 15] to [0, 5] so that the "ones place" 7-segment display circuit can display a reasonable value. If this translation doesn't happen, the ones place circuit will display gibberish and the counter won't work.

Circuit B:



This circuit simply displays the input value as a boolean, 0 or 1. It takes input from the comparator, which means that numbers greater than or equal to 10 will have 1 in the tens place and numbers that aren't will have a 0, which makes sense logically.

Top level:



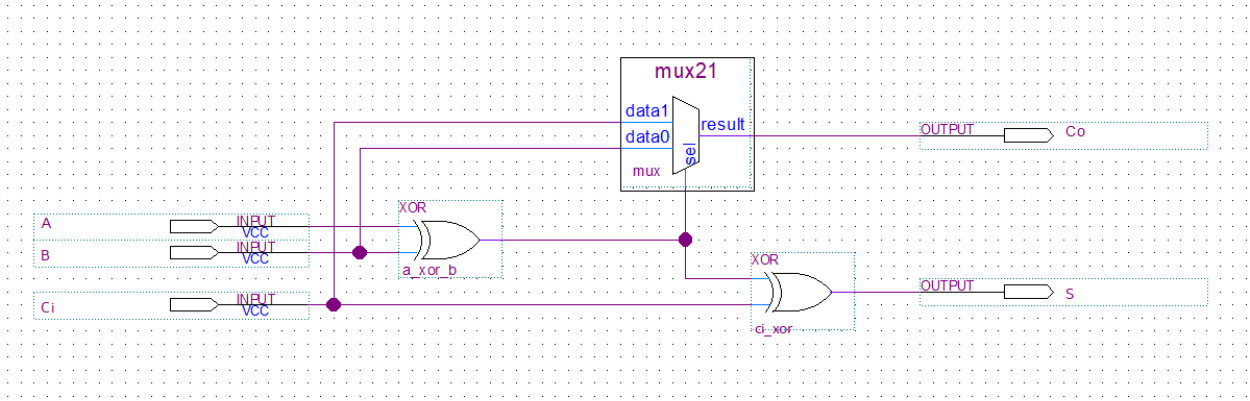
This simply combines all the components together. As stated before, the comparator output is used as a mux input to effectively toggle circuit A, which maps the number range [10, 15] to [0, 5] if needed. Otherwise, the number is directly fed to the ones place circuit and our additions effectively disable themselves.

To test this circuit, I used the following Tcl script in ModelSim:

```
for {set i 0} {$i < 16} {incr i} {
    force -drive {sim:/part_a /SW} 10#$i 0
    run 20 ns
    # Wait for input so I can show the TA before proceeding
    gets stdin
}
```

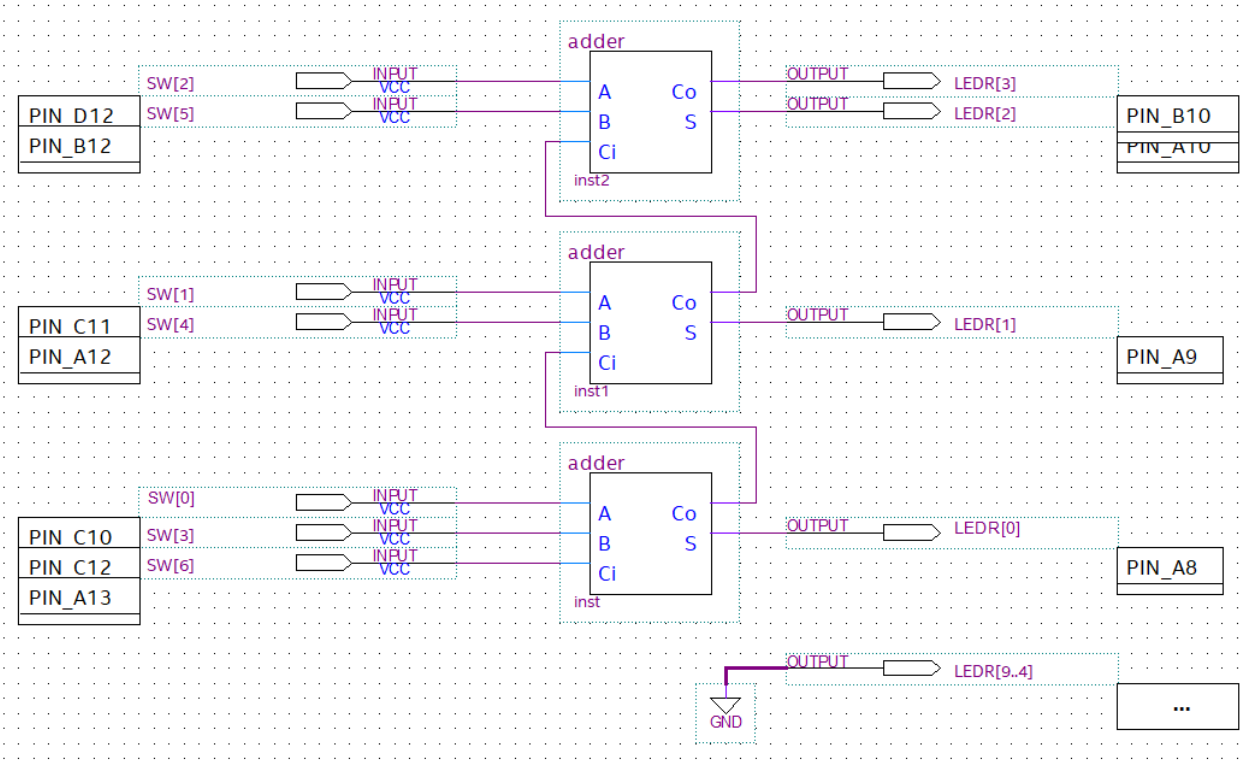
This simply loops over the whole input range (0 to 15) and pauses each time so I can analyze the output. Results are in the Results section.

Part 2:
Full adder:



This is a full adder implemented with a multiplexer. Not much to be said.

Top level:



This just feeds 3 full adders into each other to achieve a 3-bit full adder with carry in and out. Not much to be said either.

To test this circuit, I used the following Tcl script in ModelSim:

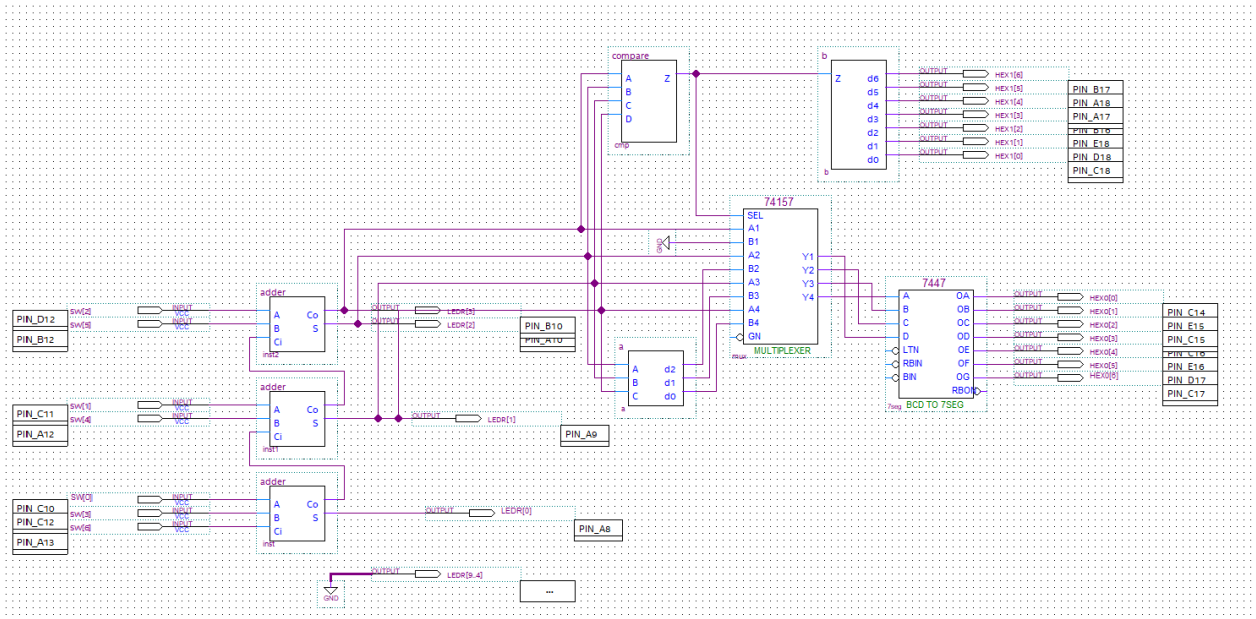
```
proc toBinStr {num width} {
    binary scan [binary format "I" $num] "B*" binval
    return [string range $binval end-$width end]
}

for {set i 0} {$i < 16} {incr i} {
    # Range is 2^3 - 1 = 7
    set a [expr {int(rand() * 7)}]
    set b [expr {int(rand() * 7)}]
    # Range is 0 or 1
    set c [expr {int(rand() * 1.999)}]
    set str $c
    append str [toBinStr $b 2]
    append str [toBinStr $a 2]

    puts [format "%s (a) + %s (b) + %s (c) = %x?" $a $b $c [expr {$a + $b + $c}]]
    puts $str
    force -drive {sim:/part_b /SW} 2#$str 0
    run 20 ns
    # Wait for input so I can show the TA before proceeding
    gets stdin
}
```

This simply chooses 2 random 3-bit binary numbers, along with a random carry bit, and then drives the SW inputs appropriately. Then it prints the expected output and waits for user input. Results are shown in the Results section.

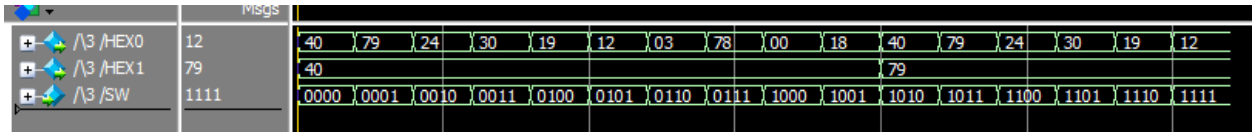
Part 3:
Top level:



This is simply a combination of the first two parts. We now take 2 3-bit numbers and a carry-in, and output the sum of the 3 inputs on our counter.

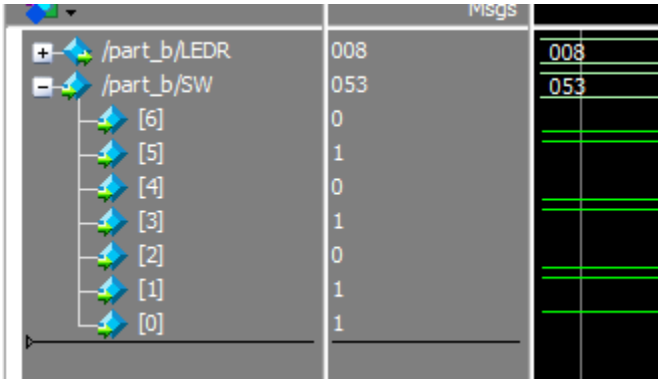
Results and Answers to Questions

Part 1:
ModelSim output:

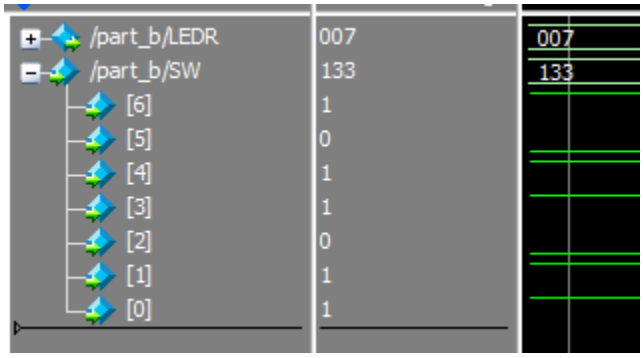


It's a bit hard to read this output, but we can see that the HEX1 output is correct (0x40 corresponds to 0b1000_0000, which is all segments on except for the middle one, and 0x79 corresponds to 0b1111_1001, which is all segments off except for the right two), and we can kind of extrapolate that HEX0 is correct because it also corresponds to the 0 and 1 outputs from HEX1 for the first part.

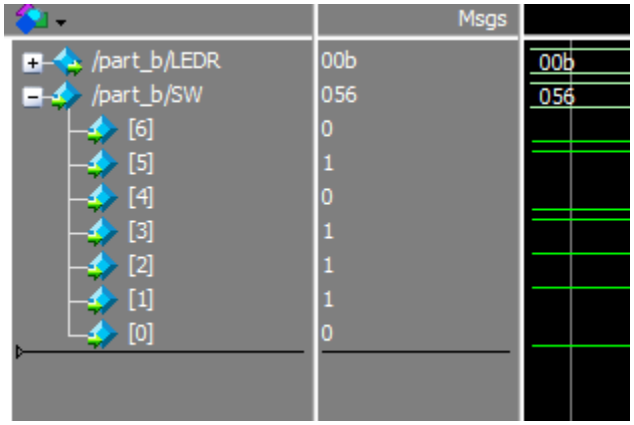
Part 2:
ModelSim outputs:



3 (A) + 5 (B) + 0 (C) = 8



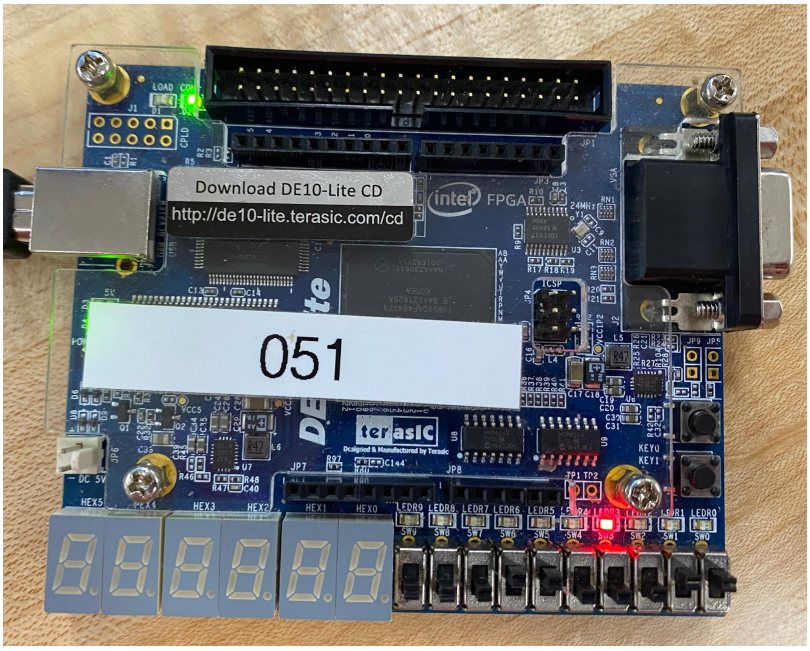
3 (A) + 3 (B) + 1 (C) = 7



6 (A) + 5 (B) + 0 (C) = 11 = 0xB

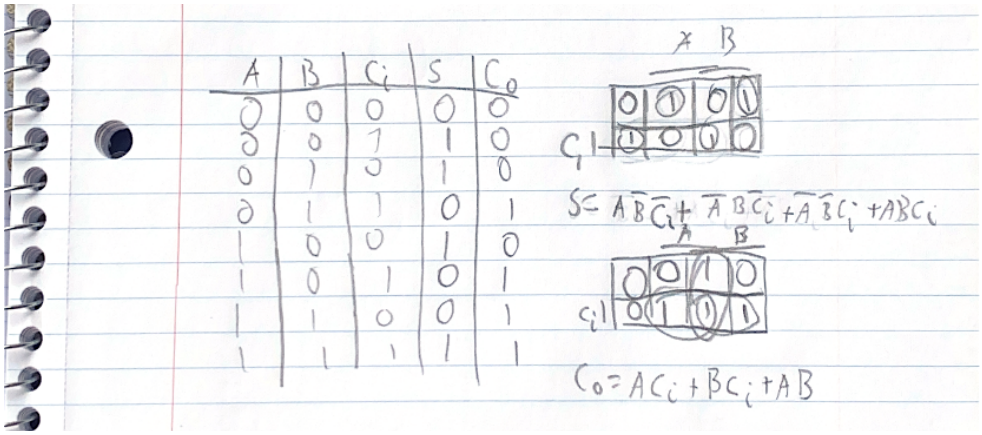
These are all verifiably correct values!

Demonstration on FPGA:



Demonstration for part 2 (Cin = 1, B = 4, A = 3, S = 8, Cout = 0)

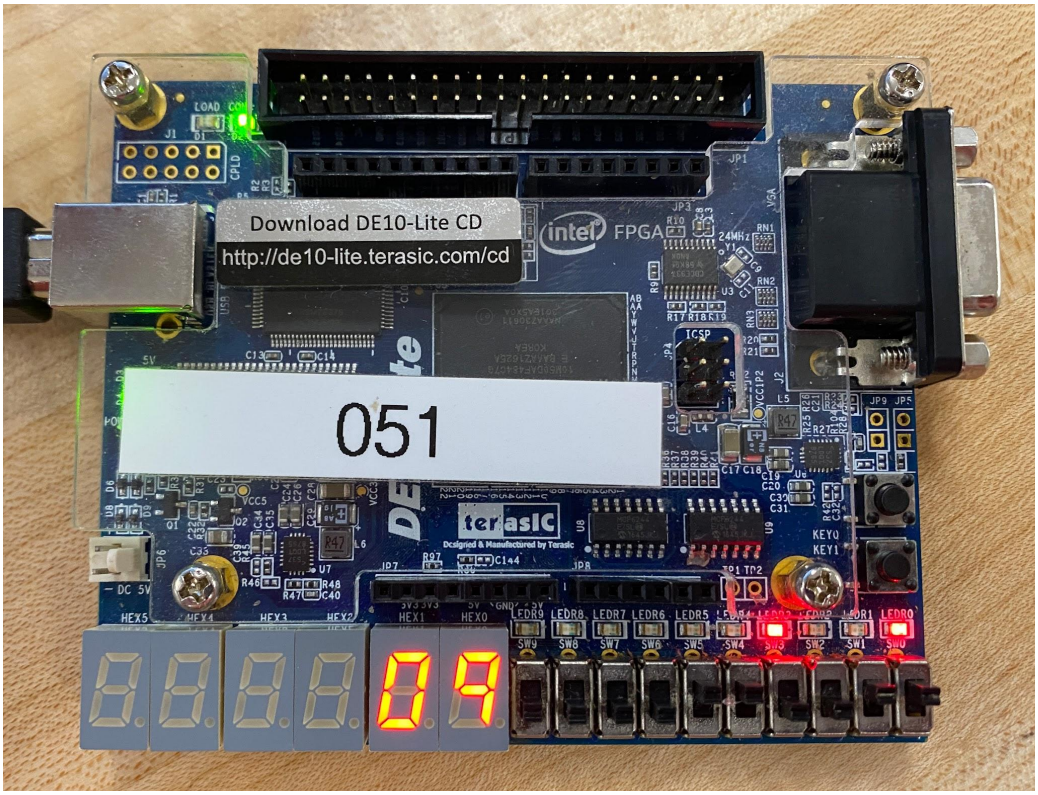
Answer to question:



There is a pretty big benefit to the XOR and mux setup given in Figure 2a. You can see that the amount of gates we need to make S and C_o with only inverters, AND, and OR gates is pretty large, while the XOR and mux setup only requires 2 XORs, and 4 more gates required to construct a 2 by 1 mux to produce both outputs!

Octal is a base 8 representation of numbers. The reason why it's convenient to represent the SW inputs to the 3-bit adder as octal is because base 8 is 2^3, which means that it combines groups of 3 bits, which make it easy to see what inputs we are feeding into the adder. It's not convenient for the 4-bit output number because it would split the output into two digits, while with hex it would group it into 1.

Part 3:
Demonstration on FPGA:

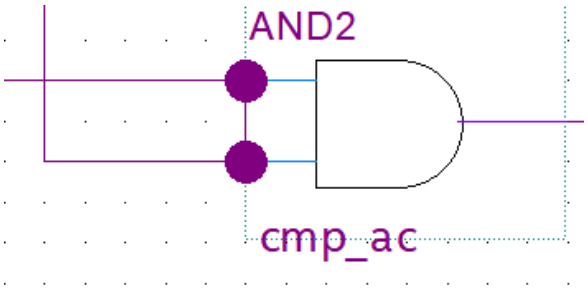


Demonstration for part 3 (C = 0, B = 6, A = 3, 3 + 6 = 9!)

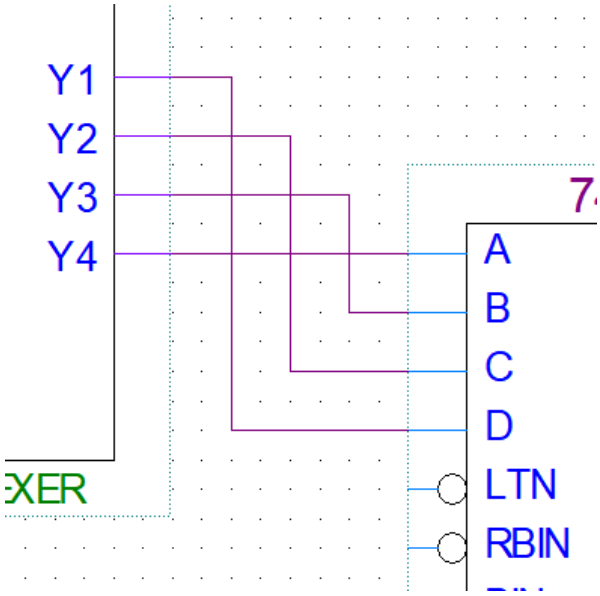
Conclusions

I think the main lesson I learned from this lab is how a multiplexer works. In our circuit, it acts somewhat like a switch, forwarding different wires based on the select input. I also learned how to make custom components that can be reused in different designs, which is useful in the same way that packaging up software is for software: it makes it much easier to reason about logic designs, along with reducing duplication.

A couple things went wrong with this lab. Initially, the circuit in part 1 did not compile for me initially because I accidentally connected two outputs to one input of an AND gate, which obviously didn't work. However, it took me a bit to figure out what had went wrong and I ended up renaming a ton of things that didn't need to be.



Also, the BCD to 7-segment circuit had a flipped bit order from the outputs on my mux, so it took me a little bit to figure out why my ones place display was showing A instead of 2, etc. I simply rewired the circuit after figuring out in ModelSim that my inputs were flipped and everything else was mostly smooth sailing.



I don't think there was much I could've improved upon in my test procedures. I wrote a Tcl script to test both parts 1 and 2, and also tested them physically on the FPGA, which were both relatively streamlined approaches and I was pretty happy with both. However, I really wish we were able to write test benches in Verilog, because that feels like it would be much easier to work with in ModelSim.

My results were as expected; there wasn't much that could've went wrong here. Overall, though, this was a good experience and I learned more about testing and combinational logic components.