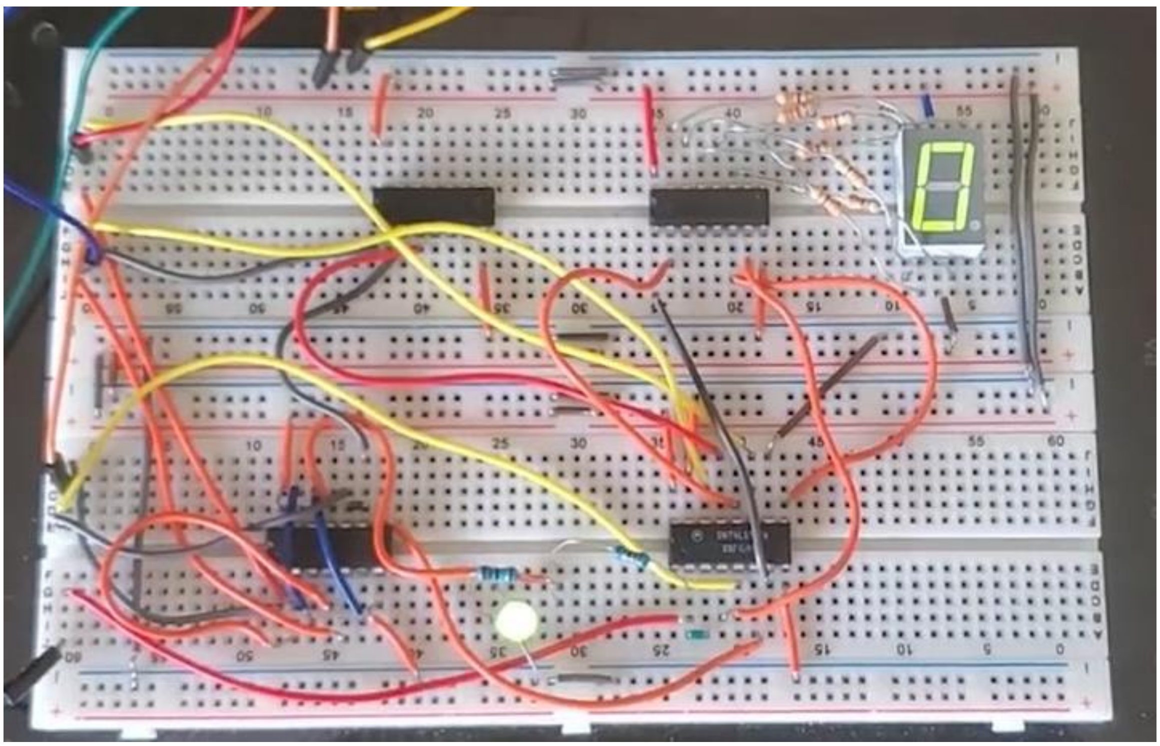
# Combinational Logic Applications

**ECE2300L Module 3 Report**




**Kyler Martinez**


**October 21st, 2020**

# Introduction

The learning objectives of the module include designing a code converter using combinations of components such as encoders, decoders, multiplexors, and arithmetic circuits. Students will also learn how to compare unsigned binary numbers and signed number in sign-magnitude/2's complement representations. Finally, to get the skills to be able to analyze a half/full adder and assemble a multi-bit add/subtract circuit with overflow detection logic.

# Activity 3.1 — Binary To BCD Converter

Truth Table For Binary To BCD Converter

| Decimal | Binary $B_3B_2B_1B_0$ | BCD $D_4D_3D_2D_1D_0$ | Adjusted $A_3A_2A_1A_0$ |
|---|---|---|---|
| 0 | 0000 | 0 0000 | 0000 |
| 1 | 0001 | 0 0001 | 0001 |
| 2 | 0010 | 0 0010 | 0010 |
| 3 | 0011 | 0 0011 | 0011 |
| 4 | 0100 | 0 0100 | 0100 |
| 5 | 0101 | 0 0101 | 0101 |
| 6 | 0110 | 0 0110 | 0110 |
| 7 | 0111 | 0 0111 | 0111 |

| Decimal | Binary $B_3B_2B_1B_0$ | BCD $D_4D_3D_2D_1D_0$ | Adjusted $A_3A_2A_1A_0$ |
|---|---|---|---|
| 8 | 1000 | 0 1000 | 1000 |
| 9 | 1001 | 0 1001 | 1001 |
| 10 | 1010 | 1 0000 | 0000 |
| 11 | 1011 | 1 0001 | 0001 |
| 12 | 1100 | 1 0010 | 0010 |
| 13 | 1101 | 1 0011 | 0011 |
| 14 | 1110 | 1 0100 | 0100 |
| 15 | 1111 | 1 0101 | 0101 |

**Table 1: Truth Table For Binary To BCD Converter**

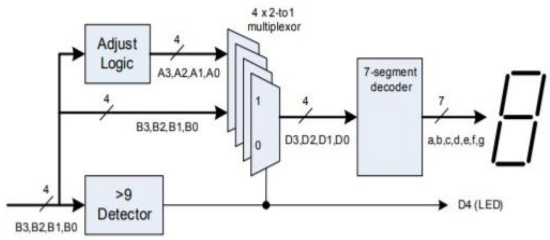



**Figure 1: Block Diagram Of The Logic Circuit**

A. $A_0 = B_0$, $A_1 = B_1'$, $A_2 = (B_2)(B_1)$, $A_3 = 0$ when $D_4 = 1$, else $A_0 = B_0$, $A_1 = B_1$, $A_2 = B_2$, $A_3 = B_3$

B.

To reduce the amount of words on the diagram the following variables are used.

X = SN74LS00  Y = SN74LS157N  Z = SN74LS08  a = SN74LS47

B = LTS-64606

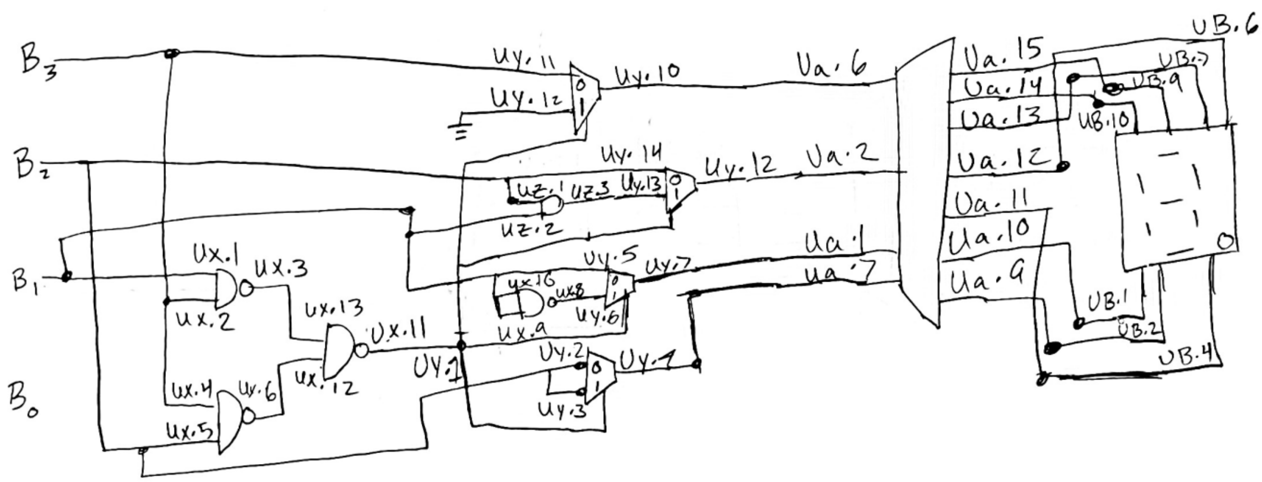



**Figure 2: Wiring Diagram**

C. Link To Video Demo: https://youtu.be/HZQJOI6bDo8

## Activity 3.2 — Half Adder/Full Adder

Truth Table For Half Adder

| A | B | $C_{out}$ | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

$C_{out} = AB$
$S = A \oplus B$

Truth Table For Full Adder

| $C_{in}$ | A | B | $C_{out}$ | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

$C_{out} = AB + C_{in}(A+B) = AB + C_{in}(A) + C_{in}(B)$
$S = A \oplus B \oplus C_{in}$

**Table 3: Truth Tables For Adders**

A.

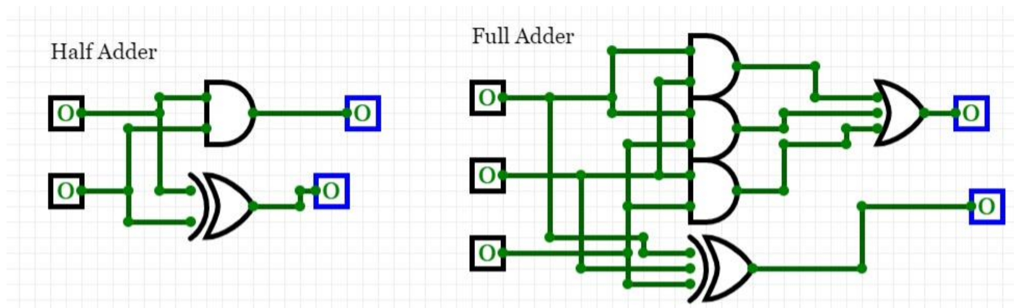



**Figure 4: CircuitVerse Diagrams for Adders**

B.

To design a circuit to count the number of 1's in a 5-bit input it would not be the best idea to use traditional tools like K-maps and truth tables to devlope the logic. This is due to the fact that it would take a long time to list out every possible input combination and the corresponding outputs. It would then take much longer to create 5 variable K-maps for each output and would make it difficult to expand the circuit to include higher bits.

C.

We could use half and full adders to design the circuit since the adders can be used to add the number of 1's in the input. The five input can be broken into two segements, a three bit and a two bit and use a half adder to find the number of 1's in the two bit and the full adder to find the number of 1's in the three bit segement. One way to complete the circuit is to use a half adder with the sum outputs of the half and full adder. The output of the sum adder of this half adder will then be the least significant bit for our total of 1's since we added bits that were in the least siginifcant positions. Then the carry from all the adders will go through a full adder and result in the most significant bit and the middle signigicant bit. Another solution is to use two full adders and begin with the sum outputs of the two and an input set to ground for the first one and then the sum of this first sull adder will be the least signigicant bit and then if the carry is brought into another full adder the carries from the original half and full added can be used and result in the most and second most siginicant bit. Both of these implementations achieve the same end result but are implemented slgihtly differnetly.

D..

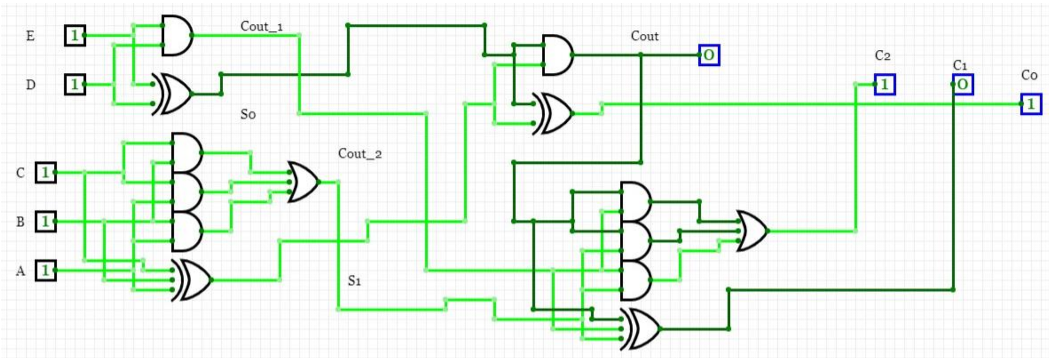
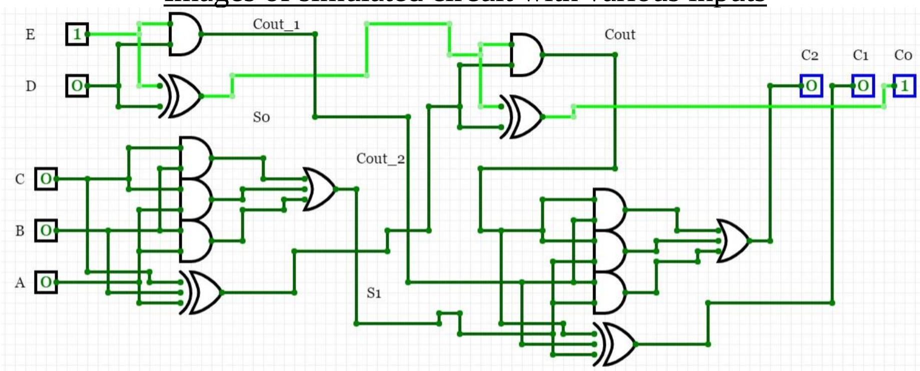
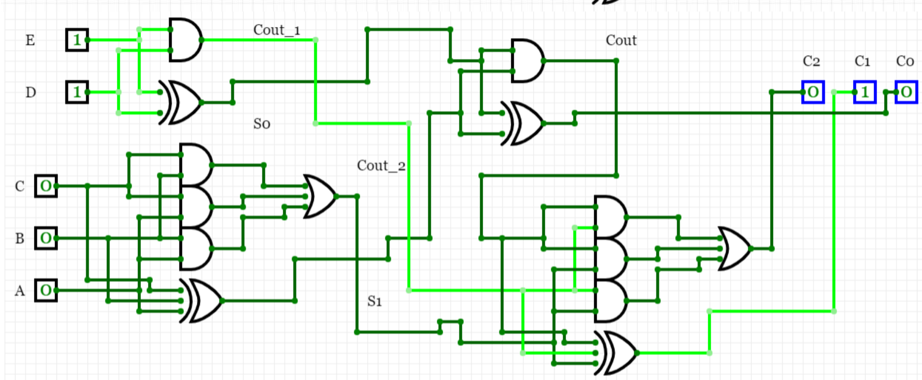


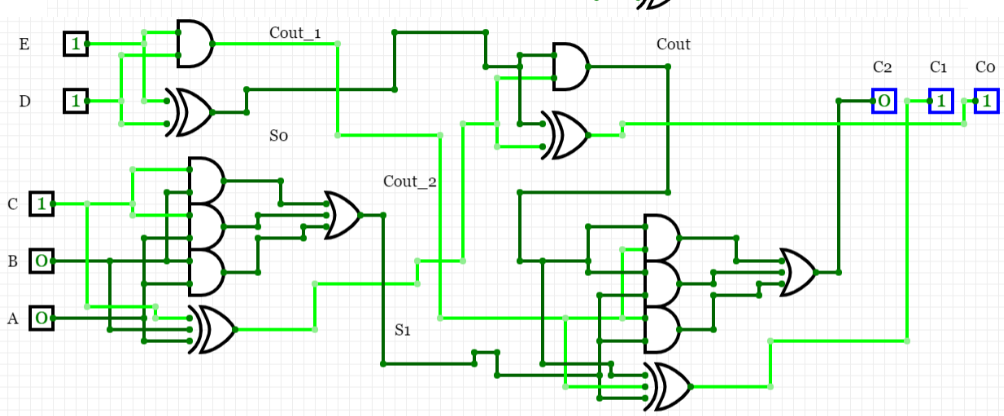

**Figure 5: CircuitVerse diagram that counts the 1's in a 5-bit input**

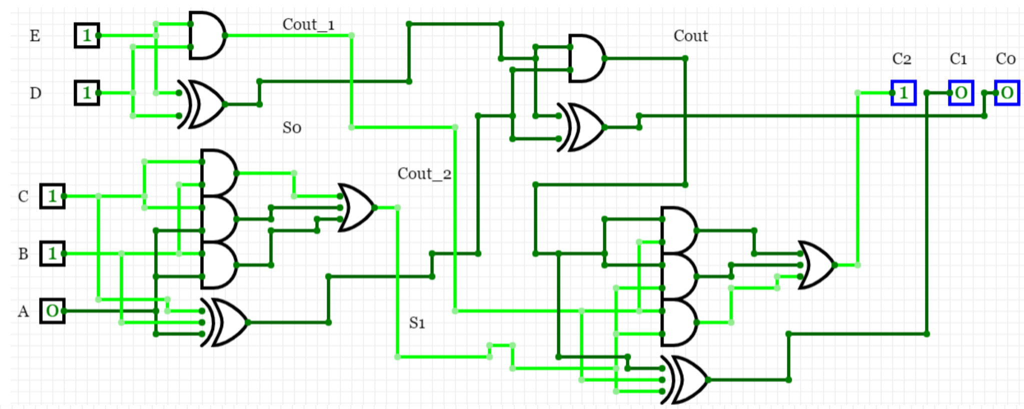
E.

<u>Images of simulated circuit with various inputs</u>

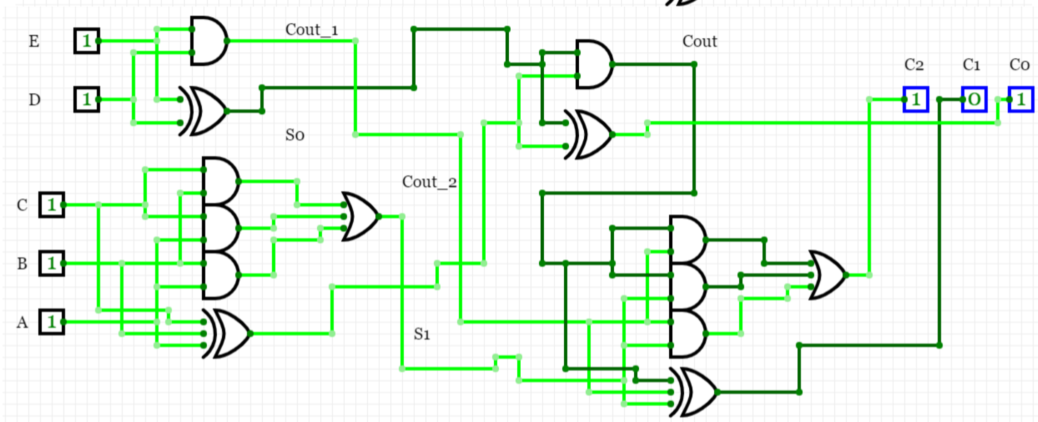


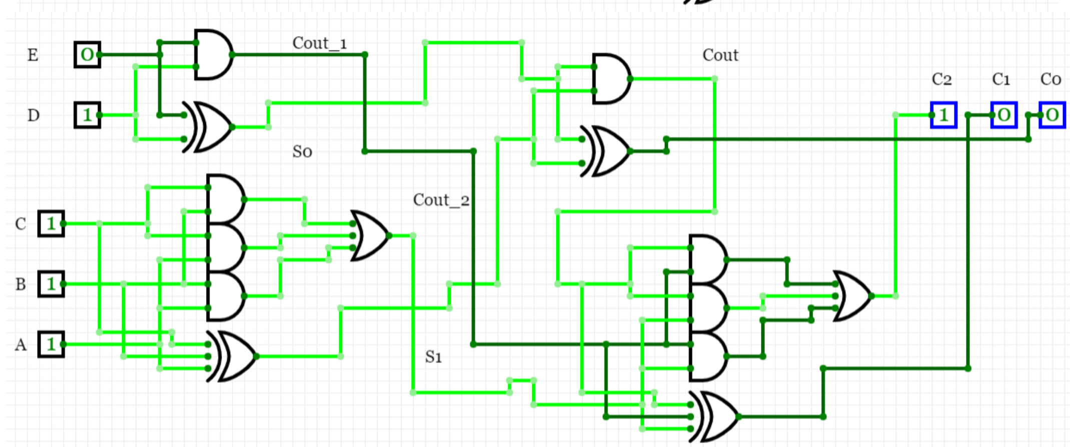





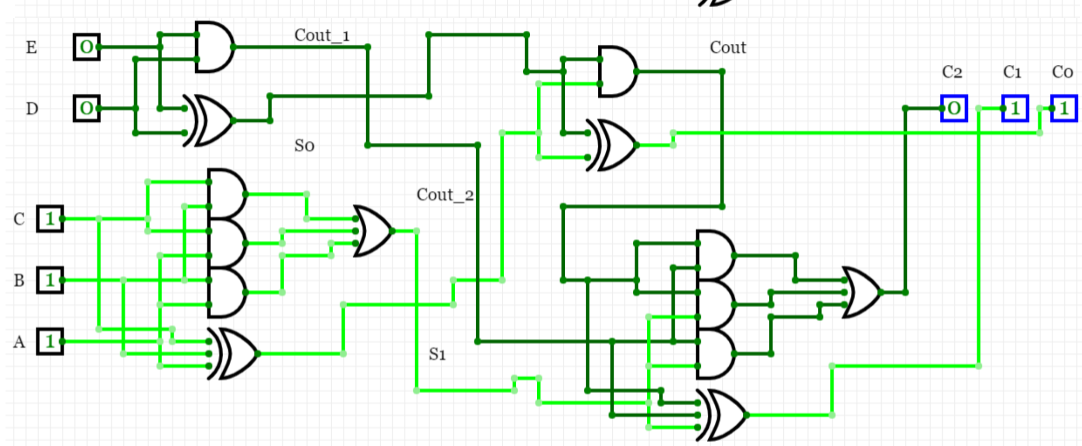
E
1
D
1
C
1
B
1
A
0
Cout_1
S0
Cout_2
S1
Cout
C2
C1
C0
1
0
0

E
1
D
1
C
1
B
1
A
1
Cout_1
S0
Cout_2
S1
Cout
C2
C1
C0
1
0
1

E
0
D
1
C
1
B
1
A
1
Cout_1
S0
Cout_2
S1
Cout
C2
C1
C0
1
0
0

E
0
D
0
C
1
B
1
A
1
Cout_1
S0
Cout_2
S1
Cout
C2
C1
C0
0
1
1

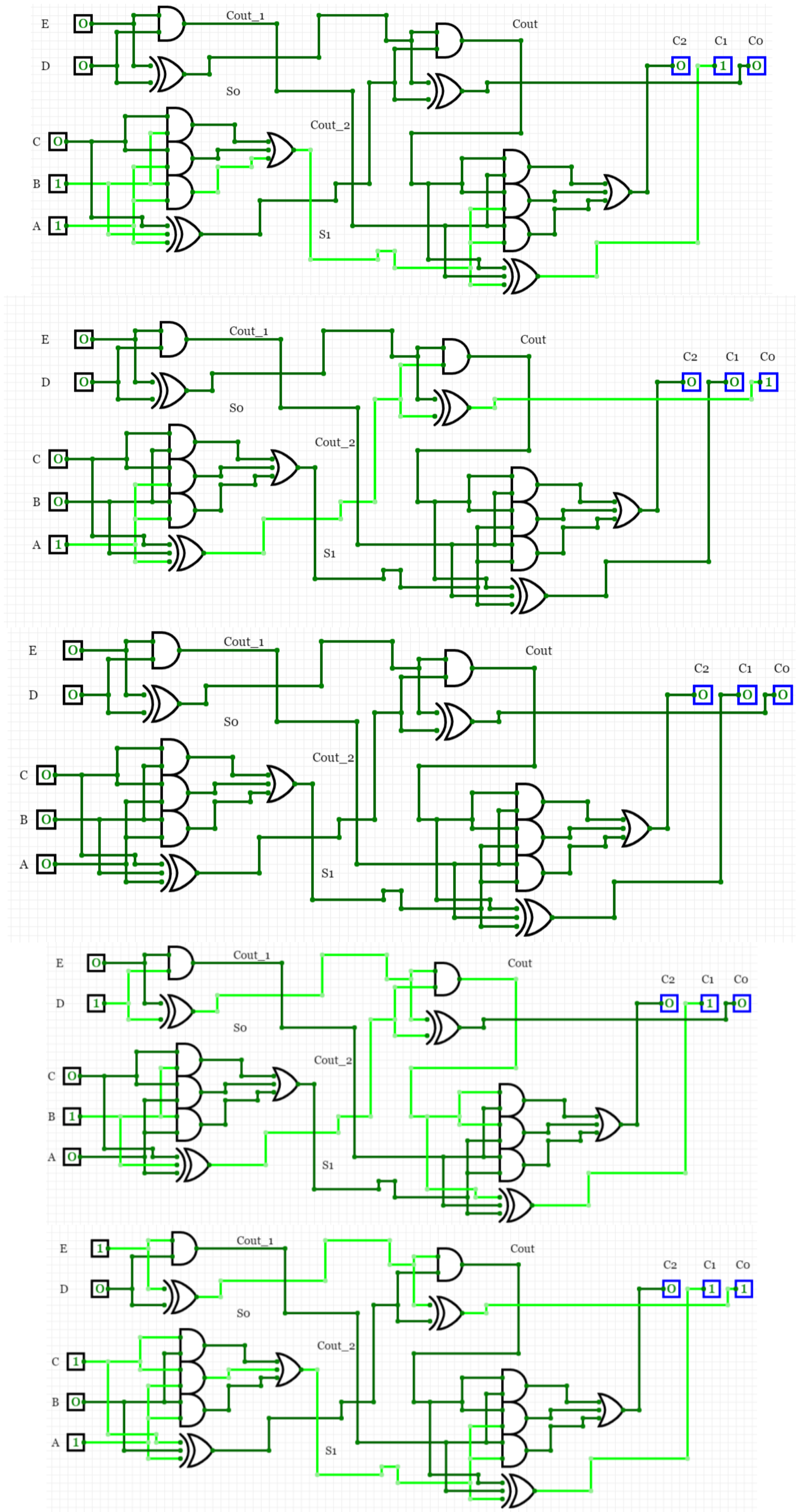



**Figure 6: Capture of circuit with different input combinations**

# Activity 3.3 — Iterative Comparator

A.

| in_gt | in_eq | in_lt | p | q | out_gt | out_eq | out_lt |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | x | x | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | x | x | 1 | 0 | 0 |

**Table 4: Truth table for single bit comparator**

B.

Logic expressions for the table above:
out_gt = in_gt + (in_eq)(p)(q’)
out_eq = (in_eq)(p’)(q’) + (in_eq)(p)(q)
out_lt = in_lt + (in_eq)(p’)(q)

C.

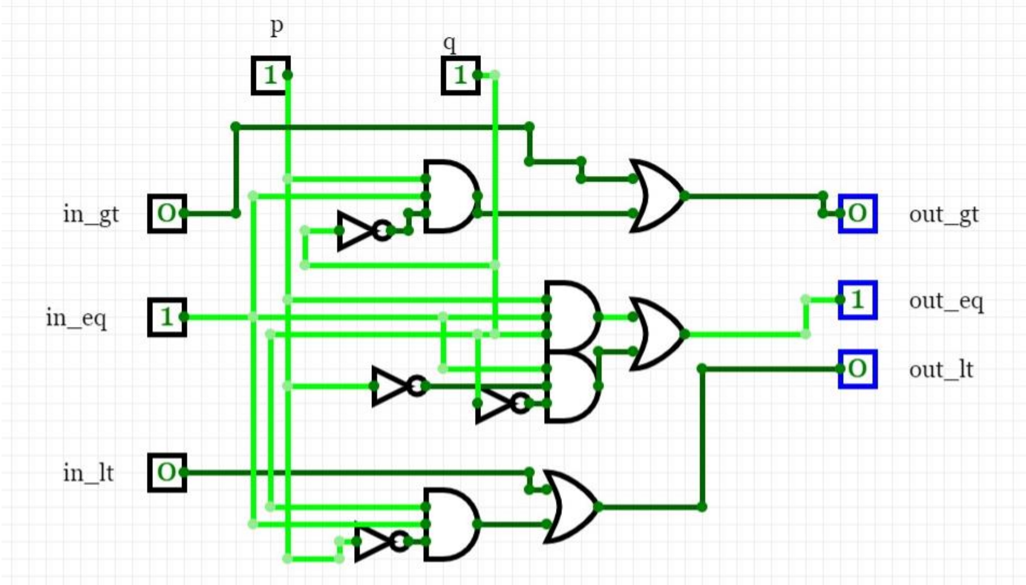



**Figure 7: Single bit comparator using CircuitVerse**

D.

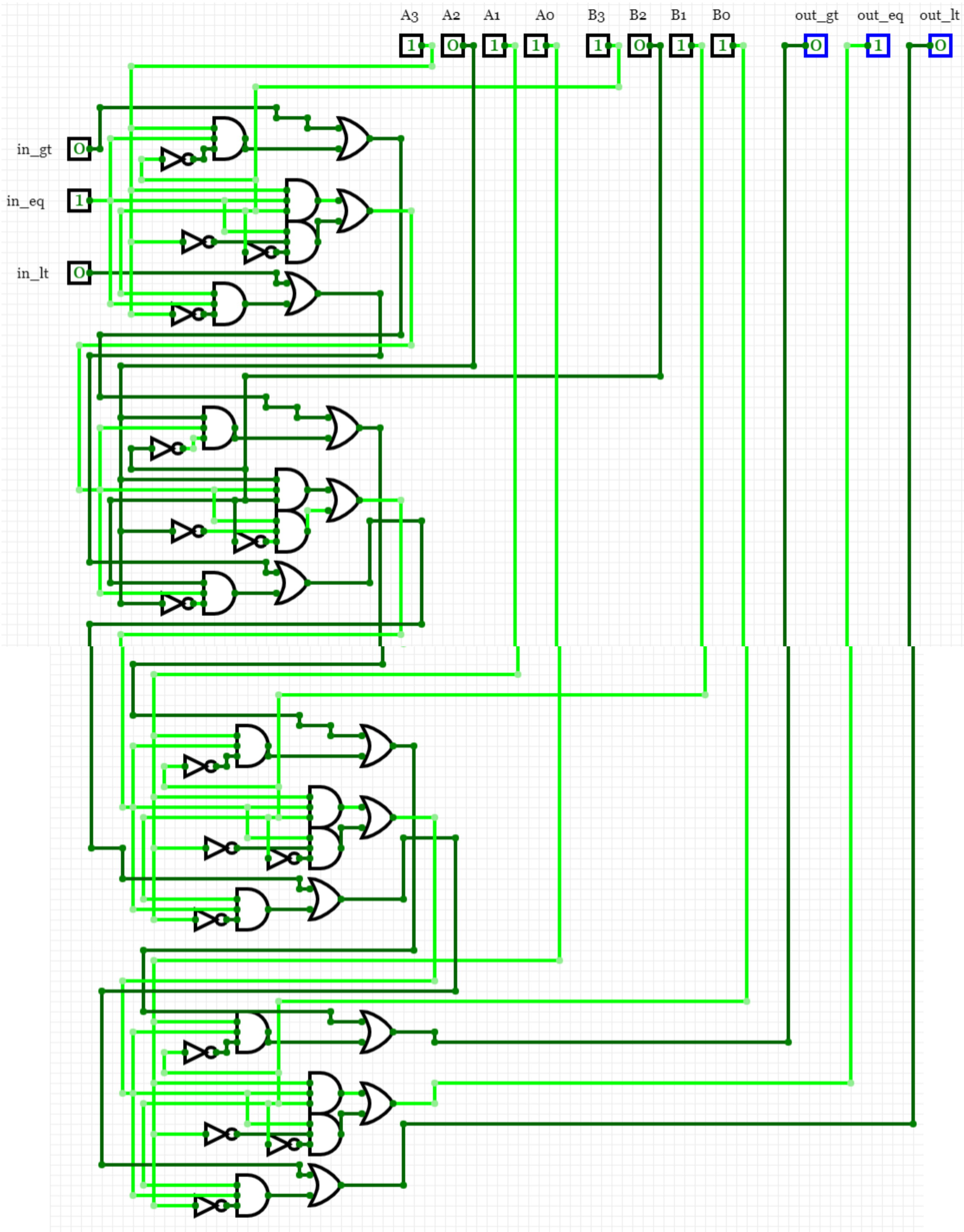
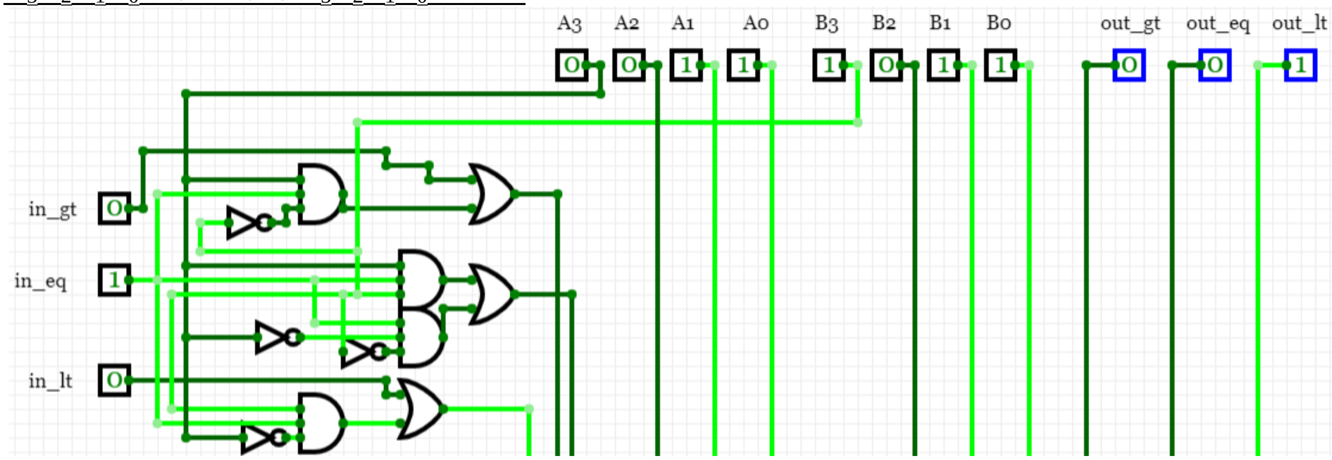


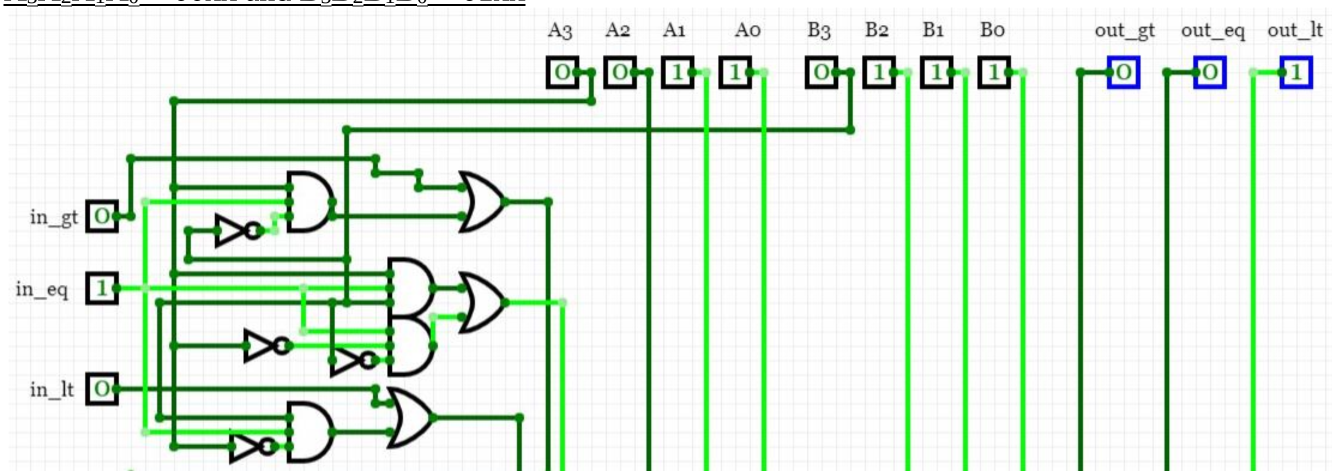

**Figure 8: Cascading 4-bit magnitude comparator**

E.
Screen shots with various input combinations.
Note: Due to the size of the circuit I only took pictures where any inputs and outputs are visible to decrease the size of the report.

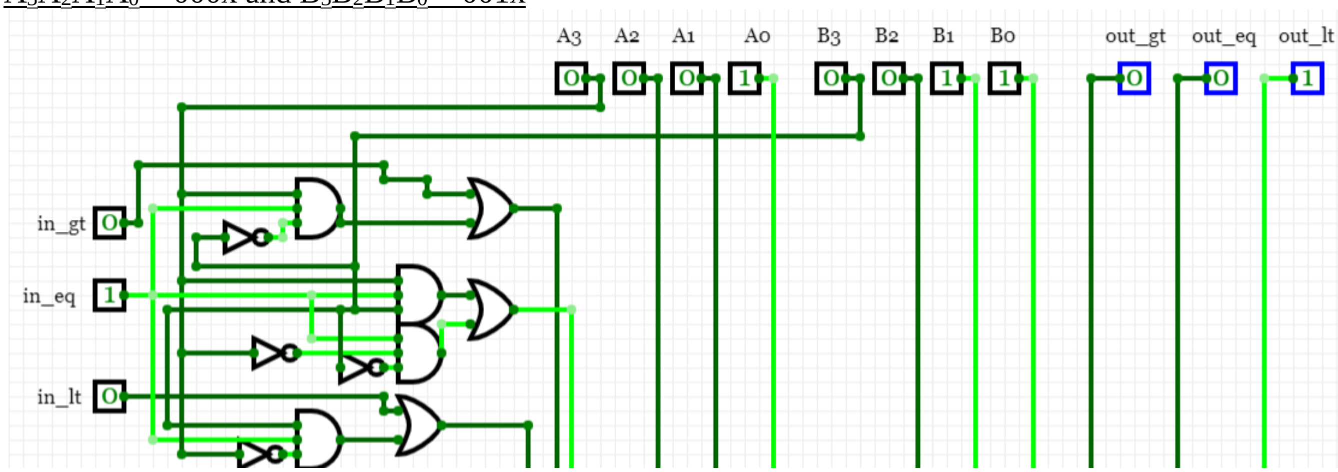
$A_3A_2A_1A_0$ = 0xxx and $B_3B_2B_1B_0$ = 1xxx



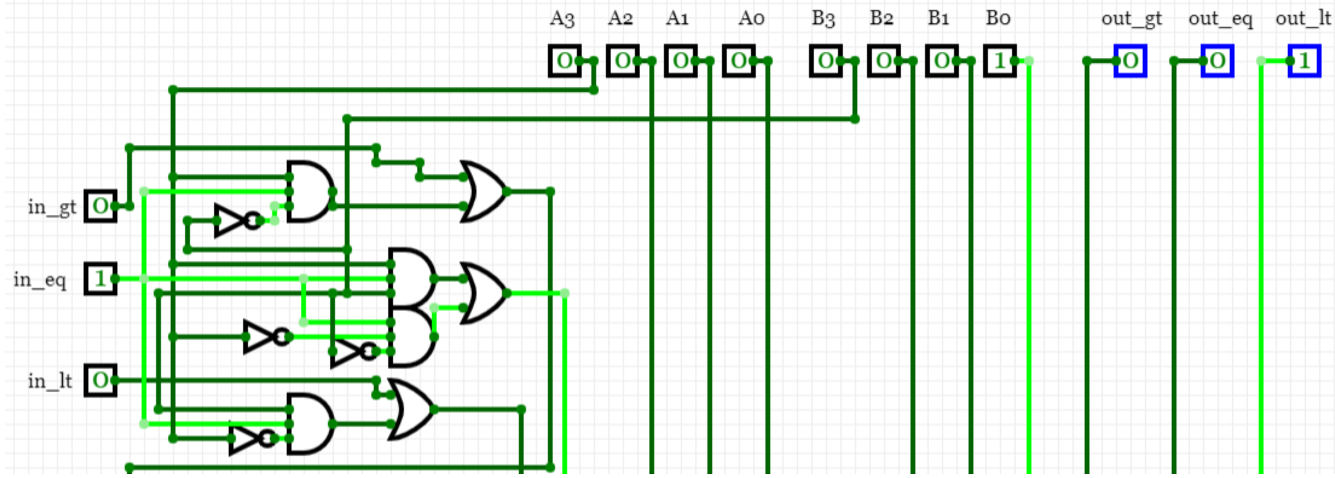

$A_3A_2A_1A_0$ = 00xx and $B_3B_2B_1B_0$ = 01xx



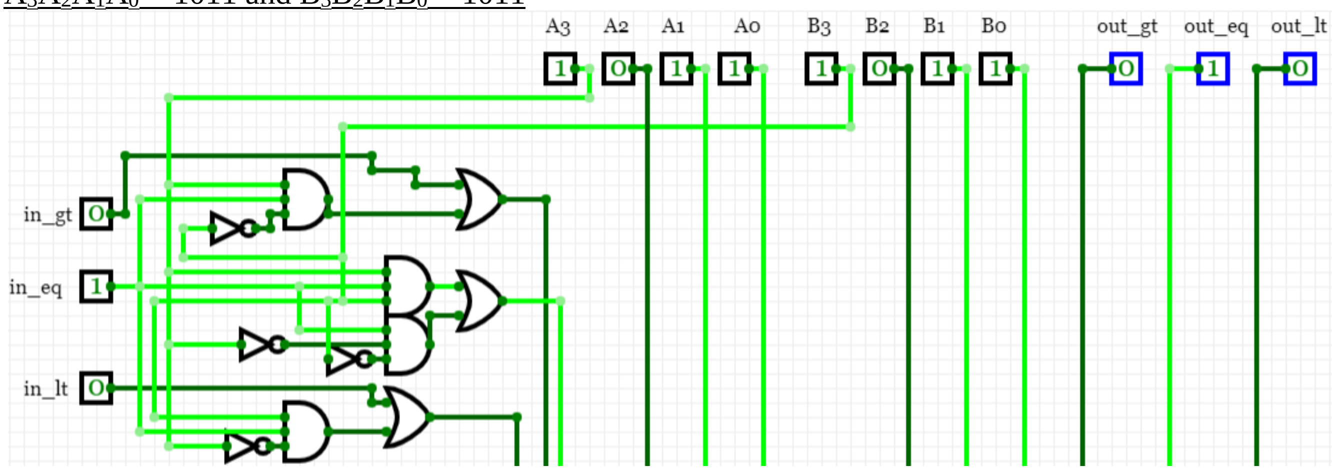

$A_3A_2A_1A_0$ = 000x and $B_3B_2B_1B_0$ = 001x



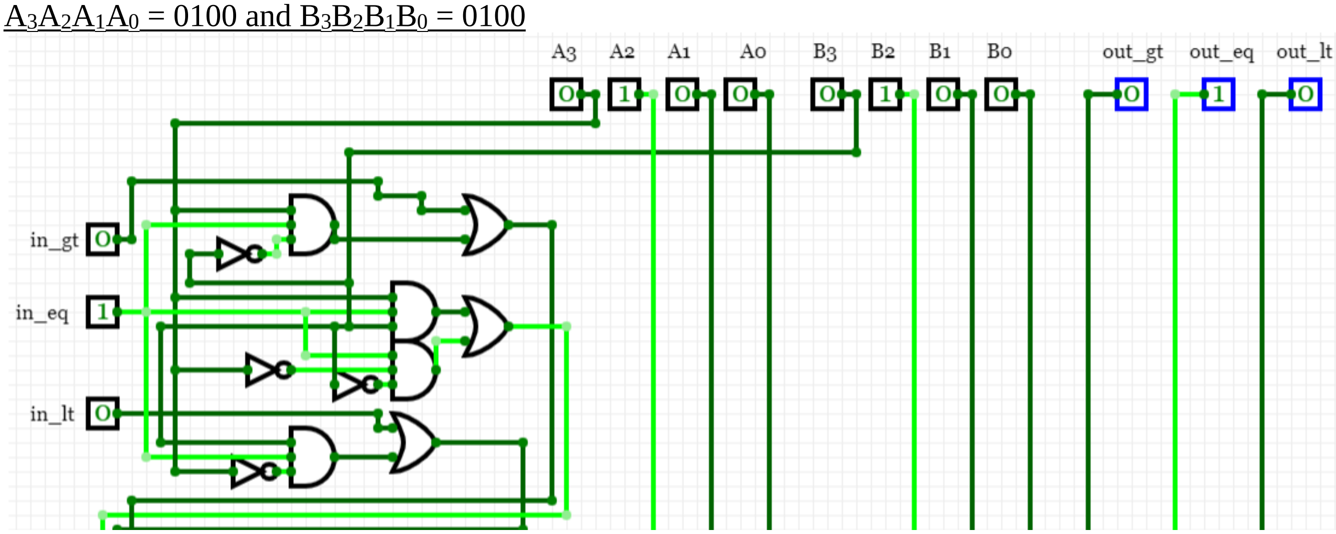
$A_3A_2A_1A_0$ = 0000 and $B_3B_2B_1B_0$ = 0001



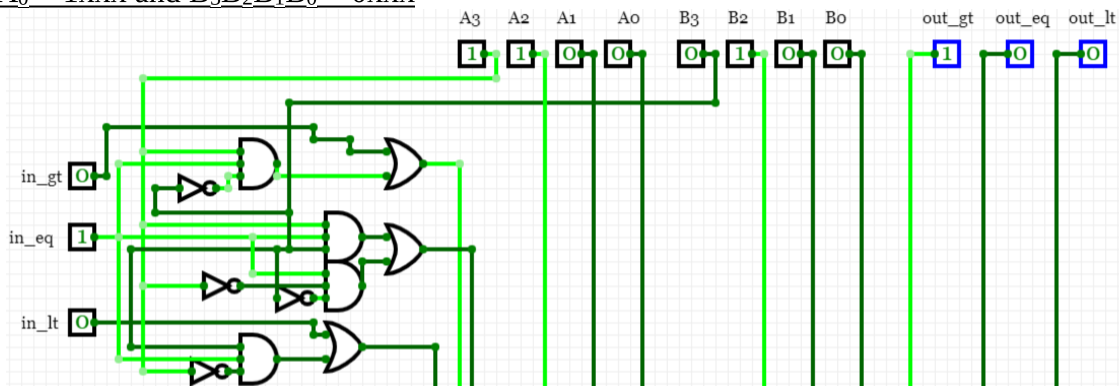

$A_3A_2A_1A_0$ = 1011 and $B_3B_2B_1B_0$ = 1011



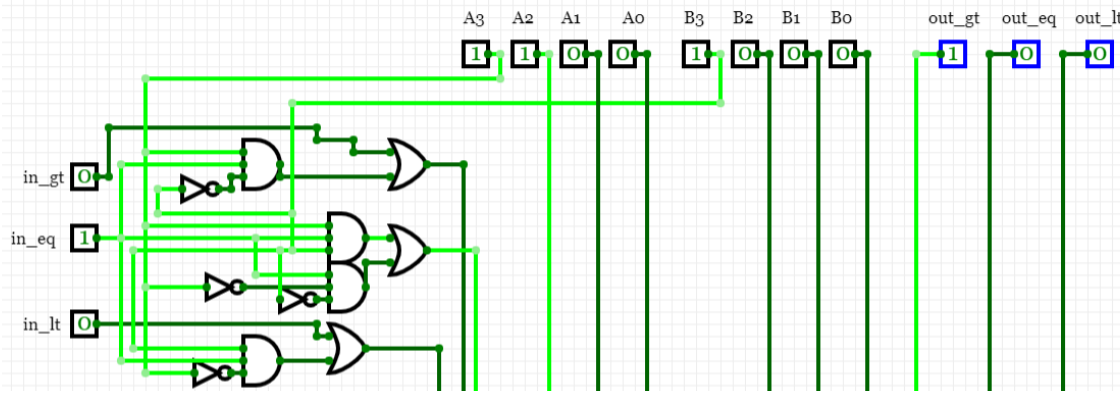

$A_3A_2A_1A_0$ = 0100 and $B_3B_2B_1B_0$ = 0100



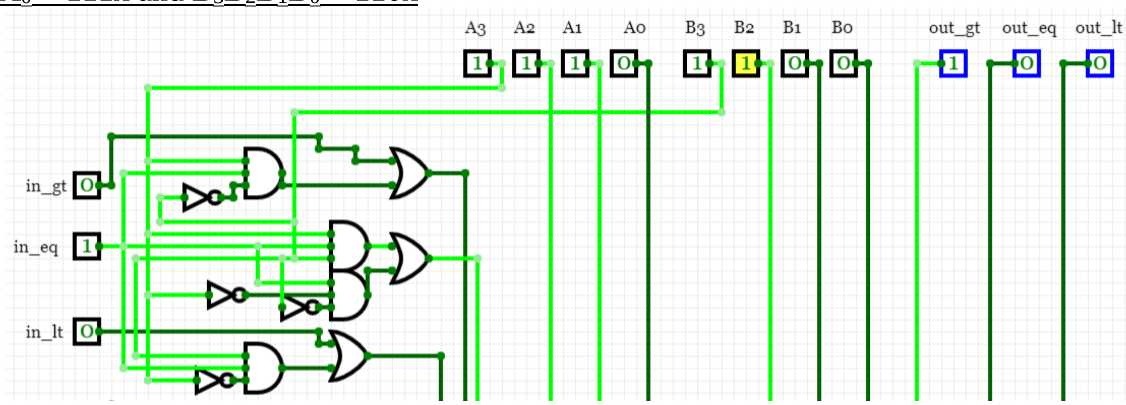
$A_3A_2A_1A_0$ = 1xxx and $B_3B_2B_1B_0$ = 0xxx



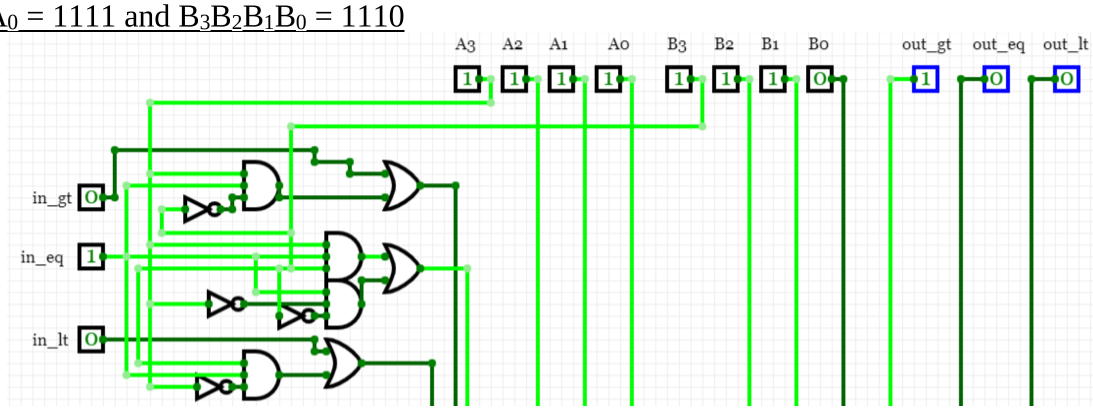

$A_3A_2A_1A_0$ = 11xx and $B_3B_2B_1B_0$ = 10xx




$A_3A_2A_1A_0$ = 111x and $B_3B_2B_1B_0$ = 110x




$A_3A_2A_1A_0$ = 1111 and $B_3B_2B_1B_0$ = 1110



# Activity 3.4 — 4-bit Add/Subtract in 2’s Complement Representation

A.

For an 8-bit binary number in unsigned representation, the decimal range is [0,255]
For an 8-bit binary number in sign magnitude representation, the decimal range is [-127, +127]
For an 8-bit binary number in 2’s complement representation, the decimal range is [-128, +127]

B.

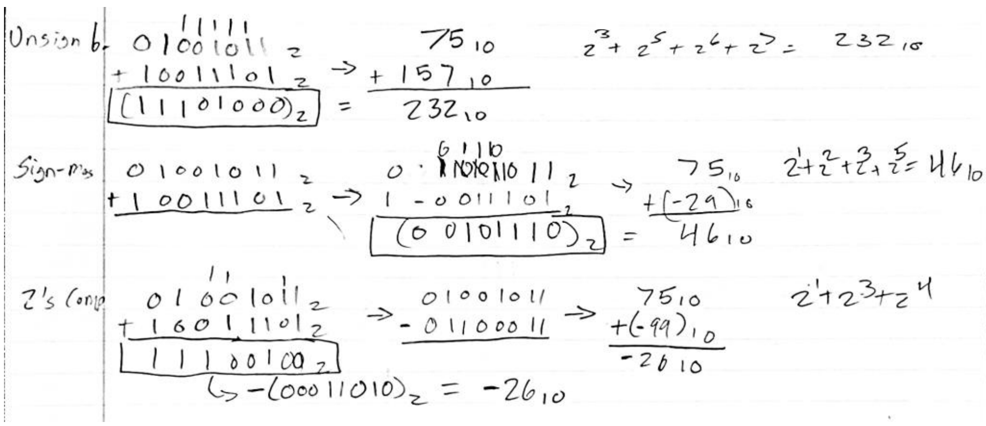


**Figure 9: Handwritten work**

If the numbers are in unsigned representation $01001011_2 + 10011101_2 = 11101000_2$
If the numbers are in sign magnitude representation $01001011_2 + 10011101_2 = 00101110_2$
If the numbers are in 2’s complement representation $01001011_2 + 10011101_2 = 11100100_2$

C.

2’s complementation is preferred since it has a slightly larger range which would be a larger benefit at lower bits rather than larger bits since there is a higher probability your sum is on the boundary of your bits if you are at smaller bits. It also helps eliminate the confusion of having two numbers representing zero. Finally, the largest benefit of 2’s complement is allowing for the user to use the same hardware for both positive and negative numbers. This is due to the fact that the problems does not have to be changed to a subtraction problem, however the result from the addition may need to be converted to sign magnitude to be in a form that is recognizable since the result of 2’s complement edition is in 2’s complement.

D.

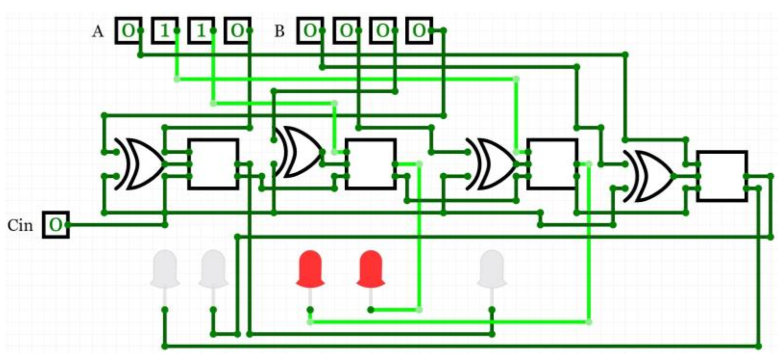



**Figure 10: CircuitVerse Diagram of Adder Circuit**

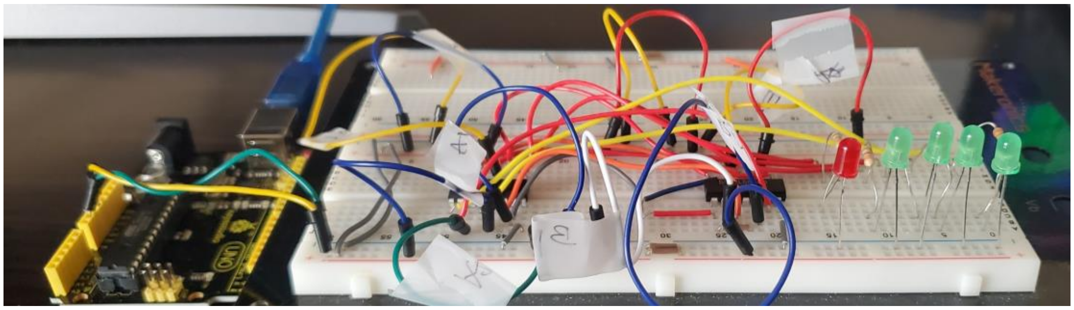


**Figure 11: Circuit With All Outputs At Logic Level 0**

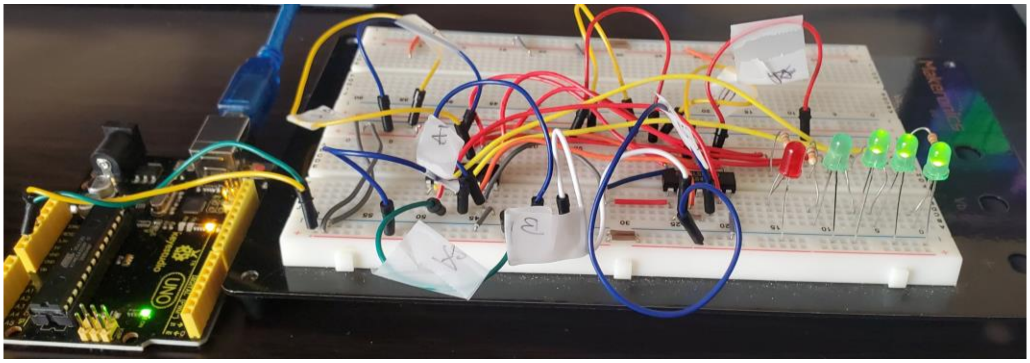


**Figure 12: Circuit Performing 0101 (A) + 0010 (B), Cin = 0**

The output was found to be 0 0111 which translates to 3 in decimal which is the expected answer.

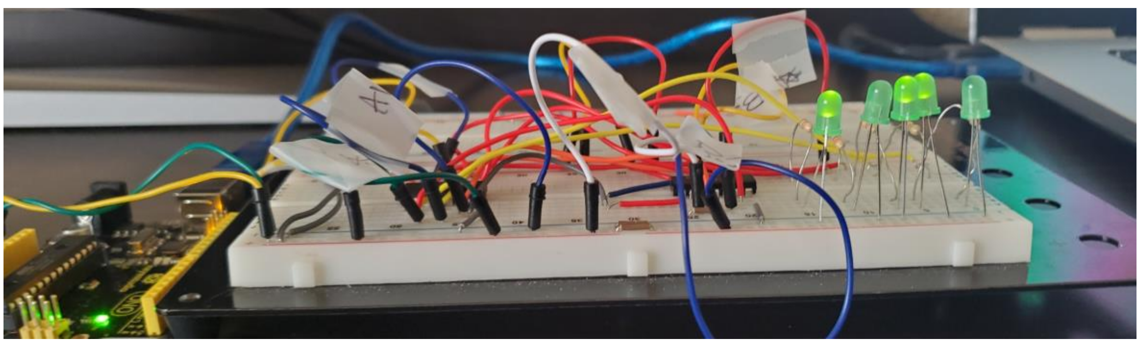


**Figure 13: Circuit Performing 0111 (A) + 1111 (B), Cin = 0**

The output was found to be 1 0110 which translates to 6 in decimal which is the expected answer.

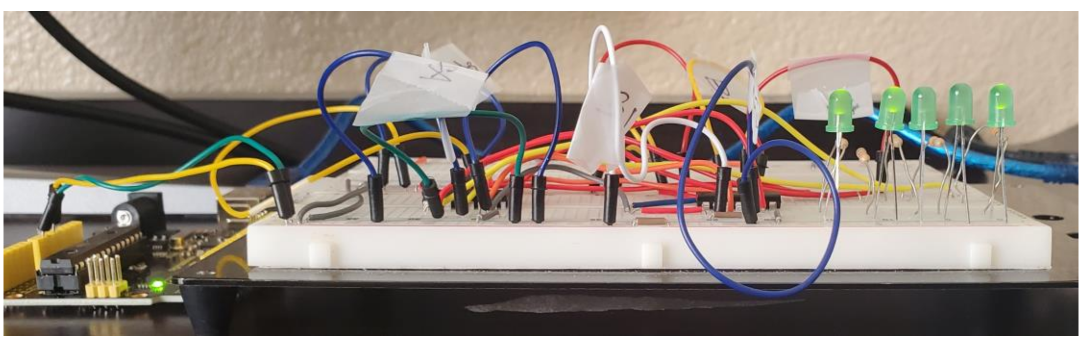


**Figure 14: Circuit Performing 1100 (A) + 1101 (B), Cin = 0**

The output was found to be 1 1001 which translates to -7 in decimal which is the expected answer.

To create an output that showcases overflow with A and B both less than 0. To do this, A and B were both chosen to be -5 in decimal, which is 1011.

**Figure 15: Circuit Performing 1011 (A) + 1011 (B), Cin = 0**

The output was found to be 1 0110 which translates to 6 in decimal, with the removal of the carry, which is not the expected answer showing an overflow error The expected answer is -10 in decimal and is out of the range for 4-bit binary numbers. Another indication of overflow is the fact that $A_3$ and $B_3$ are equal to 1 but $S_3 = 0$ which also shows overflow.

**Figure 16: Circuit Performing 0101 (A) - 0010 (B), Cin = 1**

The output was found to be 1 0011 which translates to 3 in decimal which is the expected answer.

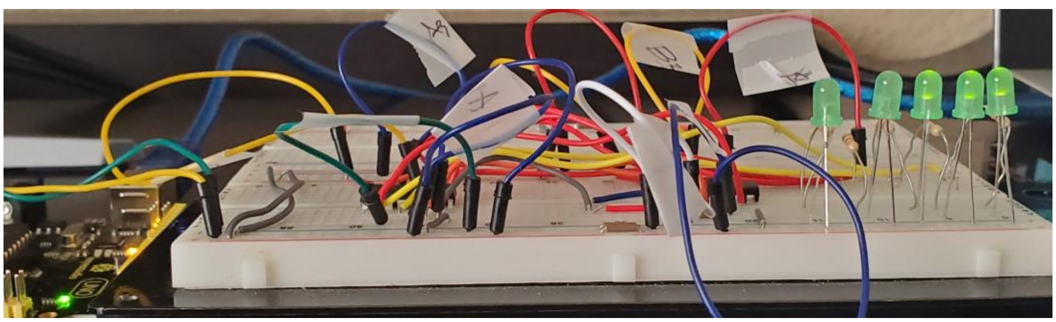


**Figure 17: Circuit Performing 0001 (A) - 1010 (B), Cin = 1**

The output was found to be 0 0111 which translates to 7 in decimal which is the expected answer.

**Figure 18: Circuit Performing 1001 (A) - 1101 (B), Cin = 1**

The output was found to be 0 1100 which translates to -4 in decimal which is the expected answer.

To create an output that showcases overflow with A is less than zero and B is greater than zero. To do this, A was chosen to be 1011 which is -5 in decimal and B to be 0101, which is 5 in decimal.

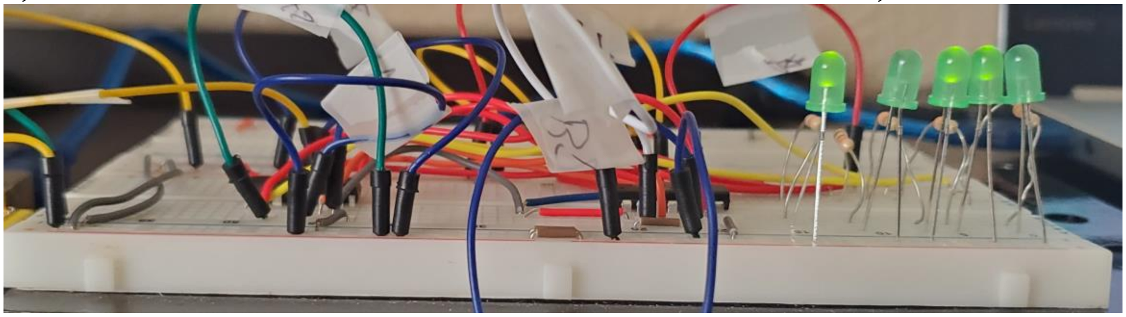


**Figure 19: Circuit Performing 1011 (A) - 0101 (B), Cin = 1**

The output was found to be 1 0110 which translates to 6 in decimal, with the removal of the carry, which is not the expected answer showing an overflow error The expected answer is -10 in decimal and is out of the range for 4-bit binary numbers. Another indication of overflow is the fact that $A_3$ and $B_3$ are different and $S_3 = 0$ which also shows overflow.

e.

C4 is not a good indicator of overflow since C4 is the carry and since we are adding binary numbers in two's complement there is the chance that we get a carryout that would be discarded in order to get the correct answer. This is the result of taking the two's complement and as such, any carry produced would need to be removed since there is an extra $2^{n+1}$ that is added when taking the 2's complement.

Two Counter examples:

Counter Example:

$7 + (-1) = 6$

```
  0111
+ 1111
 10110
```

$C_4 = 1$ but if we disregard the carry we get 0110 which is 6 in decimal, the correct answer.

Counter Example

$-6 + (-1) = -7$

```
  1010
+ 1111
 11001
```

$C_4 = 1$, but if we disregard the carry, we get 1001 which is -7, the correct answer.

e.

$C_4 \oplus C_3$ is a good indicator of overflow since overflow occurs when both $A_3$ and $B_3$ are equal and $S_3$ is opposite of the input bits. If $A_3$ and $B_3$ are equal then $S_3$ will only equal 1 if $C_3$ is 1 which will also result in a carry of 1 for $C_4$. So if $C_3$ is equal to zero and $A_3$ and $B_3$ are equal, then $S_3$ will be zero and $C_4$ will be 1. This is a case of overflow and could be identified with the logic expression $C_4 \oplus C_3$.

Two Examples:

Example.

$6 + 4 = 10$

```
  0110
+ 0100
 01010
```

Our result 1010, seems correct however this would be a negative number which is impossible when adding two positive numbers. $C_4 = 0$, but $C_3 = 1$ $C_4 \oplus C_3$ would be 1 showing overflow.

Example

$-5 + -5 = -10$

```
  1011
+ 1011
 10110
```

We have overflow since if we removed the carry, our answer would be 0110 which is 6 and not -10. $C_3 = 0$ and $C_4 = 1$. $C_3 \oplus C_4 = 1$ which would correctly identify overflow.

# Exercise 3.1 — Mux-Demux Transmission

I started by creating a truth table and formed a form of a truth table that helped me find solution. I made the selection inputs of the demux be the outputs of a logical function involving the first

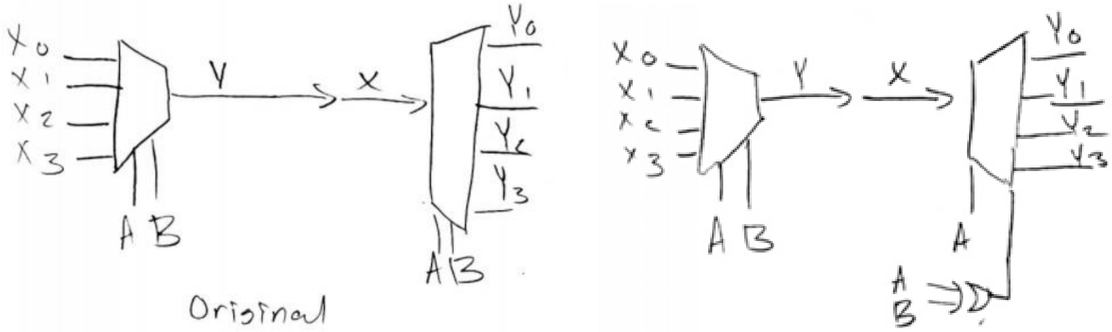



**Figure 20: Original Mux to Demux Diagram and Diagram With Added Control Logic**

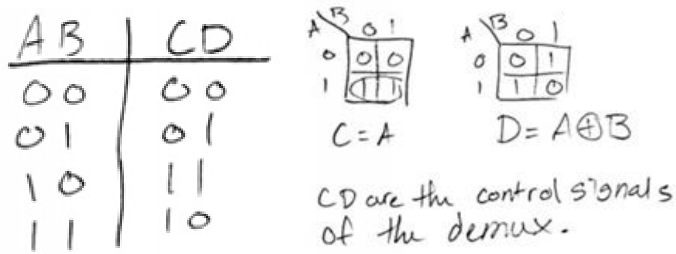



**Figure 21: Truth Table & K-Map to find the logic for the control signals**

By applying the XOR logic gate to the second selector input will result in $X_2$ activating $Y_3$ and $X_3$ activating $Y_2$ and $X_0$ and $X_1$ activating $Y_0$ and $Y_1$ respectivly.

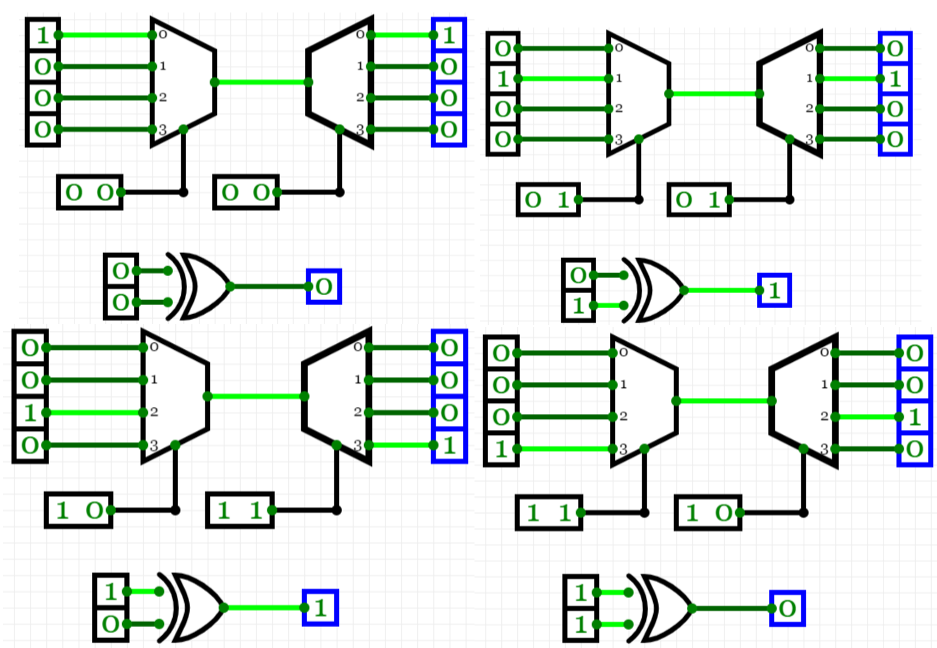


**Figure 21: CircuitVerse Implementation To Confirm The Design**

# Exercise 3.2 — Follow-Up On Activity 3.2

To scale up the circuit in activity 3.2 to be able to count the number of 1's in a 7-bit input, I took the approach of adding onto the original circuit as opposed to incorporating the new bits in the early stages of the circuit. I chose this since approach since I would be more inclined to continue adding onto my original circuit in lab, rather than spending the time to rebuild it. However this methodology may result in using more than materials than necessary. My approach began with adding the additional bits to the least significant bit of the original circuit with a full adder. The sum input becomes the new $C_0$ and then the carry from the full adder is added to the old $C_1$ and this process continues with each of order of magnitude until each bit is added. This process results in an extra bit $C_3$ however it is not necessary for this number of inputs since the total number of 1's of a 7-bit input can be represented by three bits.

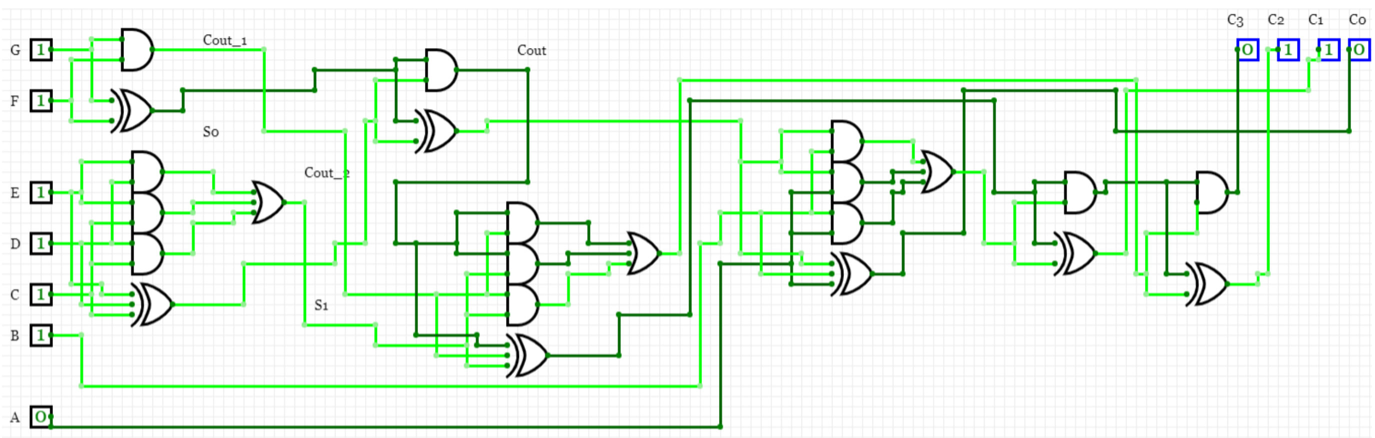



**Figure 21: CircuitVerse Implementation Of 7-Bit 1's Adder**

# Exercise 3.3 - Follow-Up On Activity 3.3

To implement a 4-bit comparator circuit that would detect if a number were greater than 4 and less than or equal to 10, I began with two comparator circuits. I then used an AND gate that would take two inputs, one that said whether the number, n, was greater than 4, and one that said if n was less than or equal to 10. For the input that used that determined if the number was greater than 4, I connected the greater than output to the AND gate. For the other input I used the out_eq and out_lt outputs and put them through an OR gate and then put the output into the final AND gate. The output of this AND gate would be at logic level one if both the number were greater than 4 or less than or equal to 10. To simplify the circuit, I began removing the outputs that I did not use which allowed for me to remove portions of the circuit that did not affect the output I needed. The downside that I can foresee is that this circuit will only be useful for this application and would not be easily changed for another application, however this implementation uses far less gates.

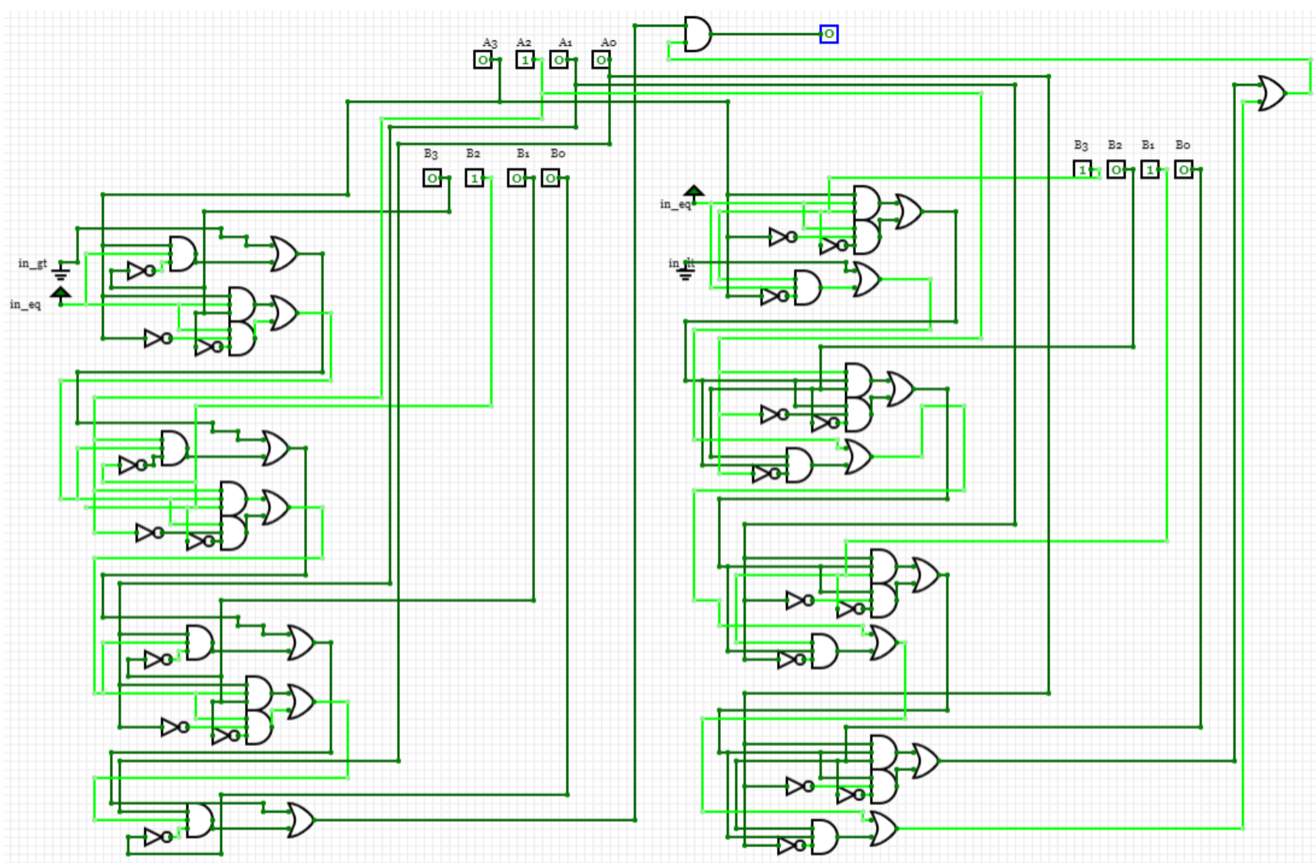
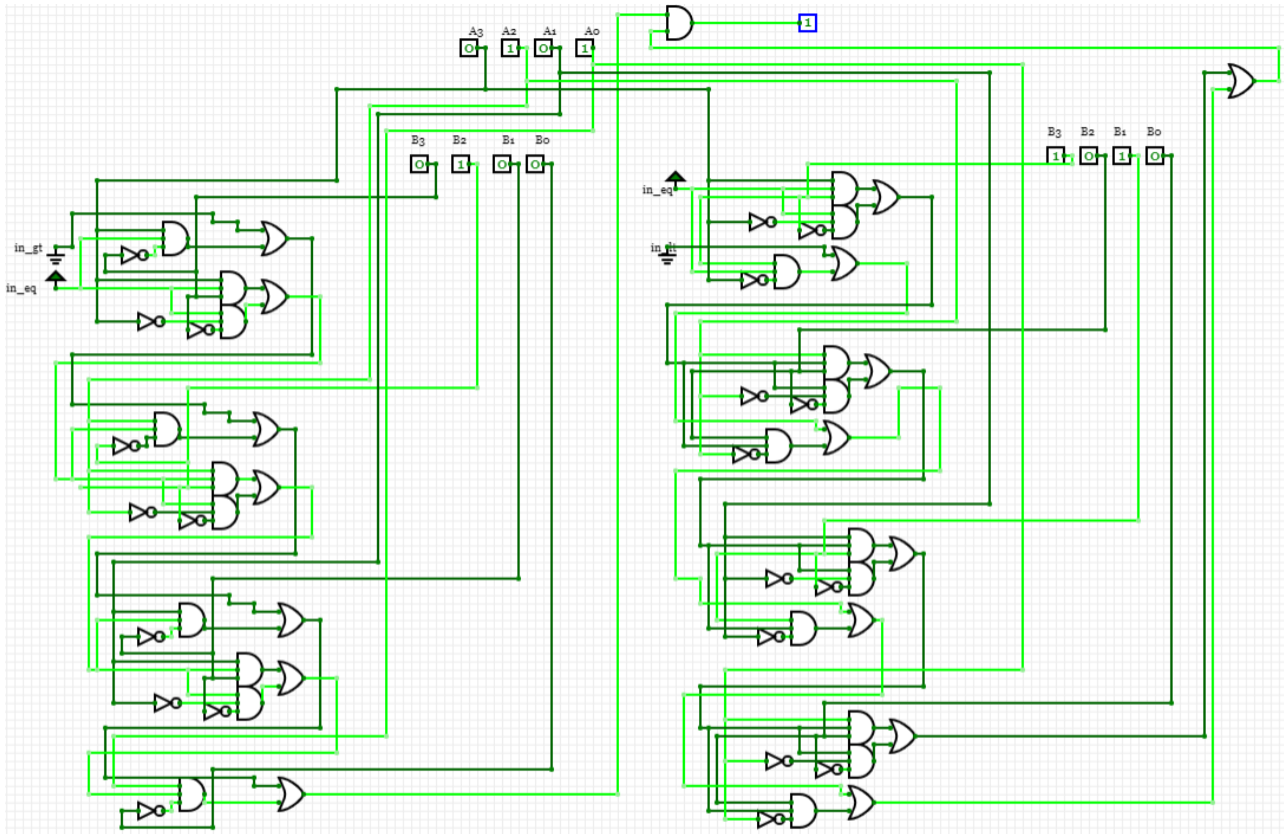
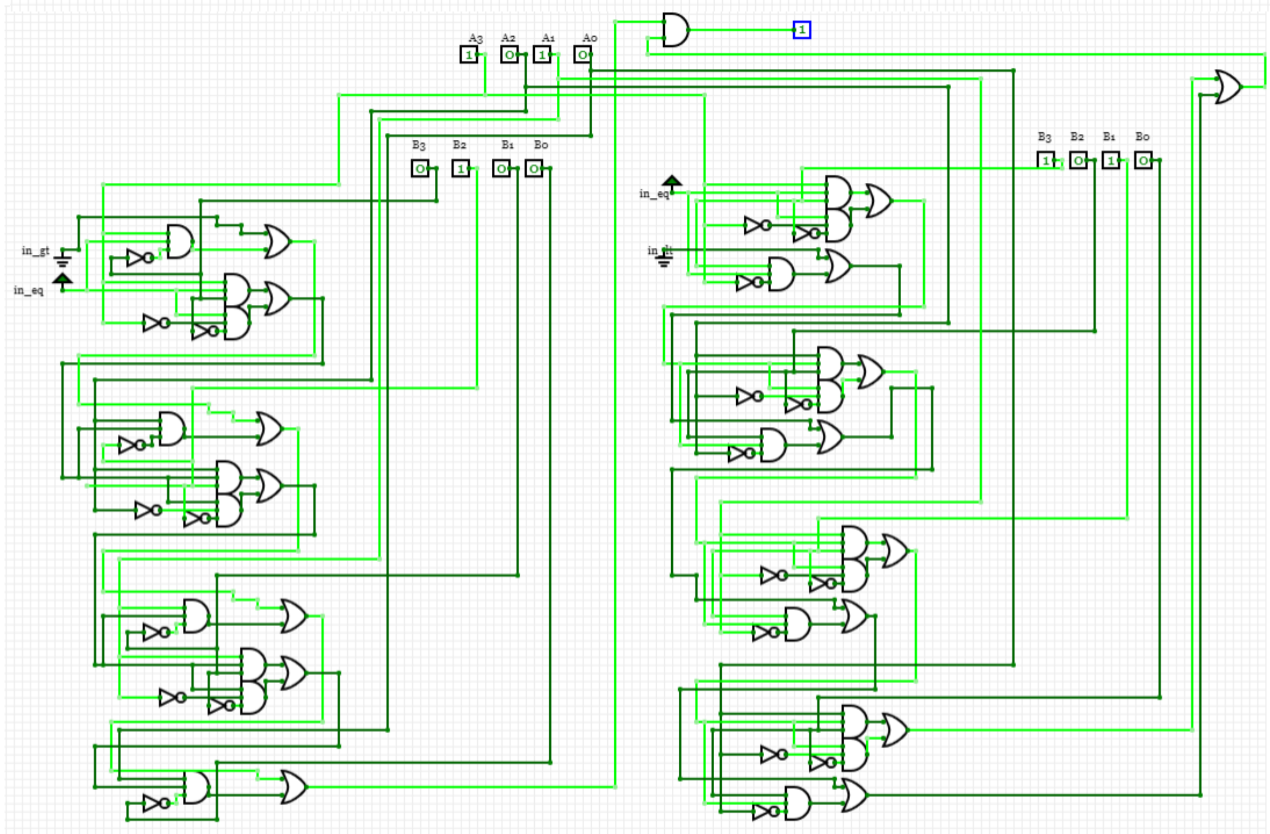
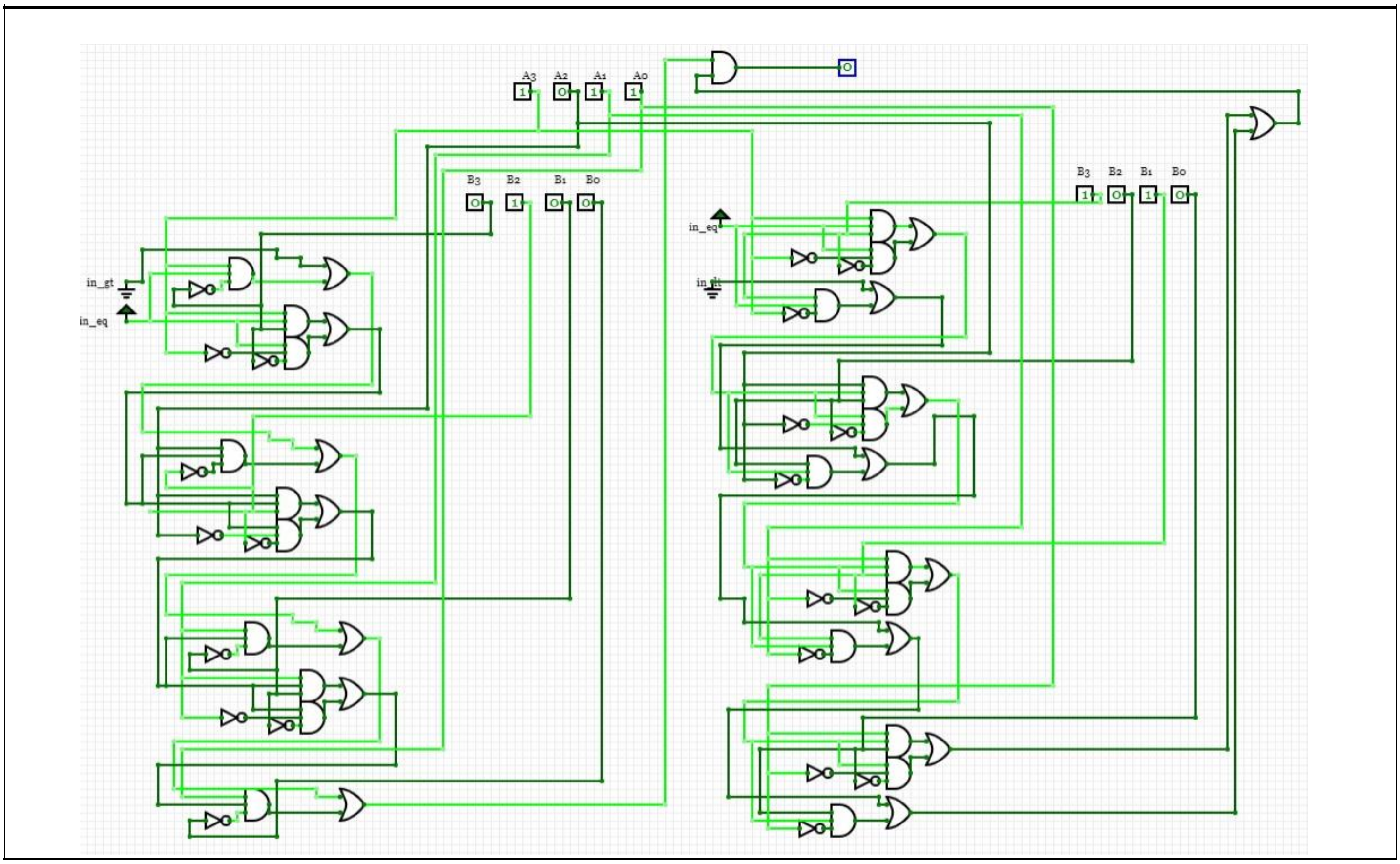



**Figure 22: CircuitVerse Implementation Of Comparator With Various Inputs**


A3
A2
A1
A0
B3
B2
B1
B0
in_gt
in_eq
in_lt


A3
A2
A1
A0
B3
B2
B1
B0
in_gt
in_eq
in_lt



A3
A2
A1
A0
B3
B2
B1
B0
in_gt
in_eq
in_eq
in_lt
B3
B2
B1
B0

# Exercise 3.4 - Follow-Up On Activity 3.4

A.

The two general cases when we have overflow are when $A_3$ and $B_3$ are both zero and $S_3$ is one which implies that $C_3$ is one but $C_4$ is zero since the addition of 0 and 1 does not produce a carry. The other case where overflow occurs is when $A_3$ and $B_3$ are both one and $S_3$ is zero which implies that $C_3$ is zero and $C_4$ is one since the addition of 1 and 1 produces a carry.

B.

To implement overflow detection with the 74LS283 we would use the logic function $O(A_3,B_3,S_3) = A_3B_3S_3' + (A_3)'(B_3)'S_3$ which incorporates both general cases where overflow occurs.

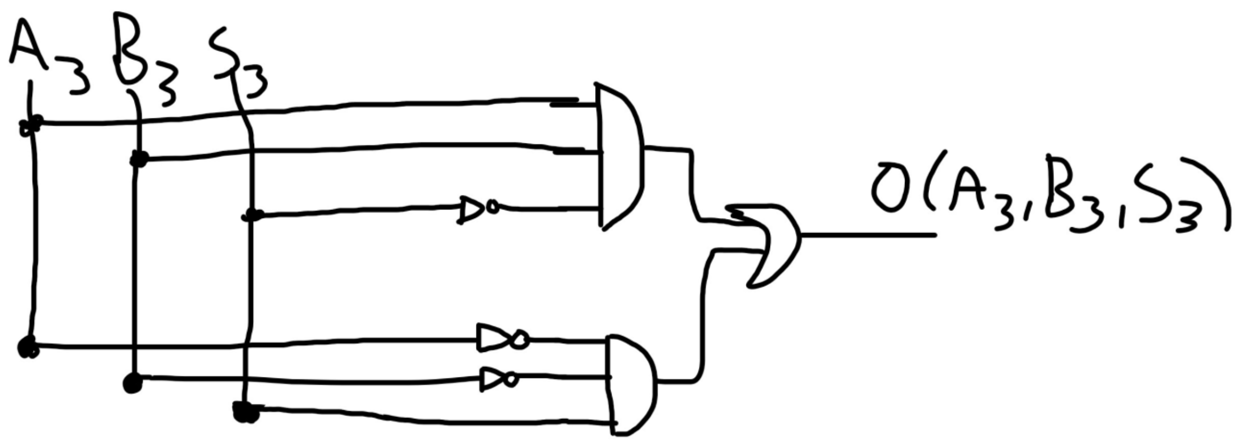



**Figure 23: Logic Diagram To Determine Overflow.**