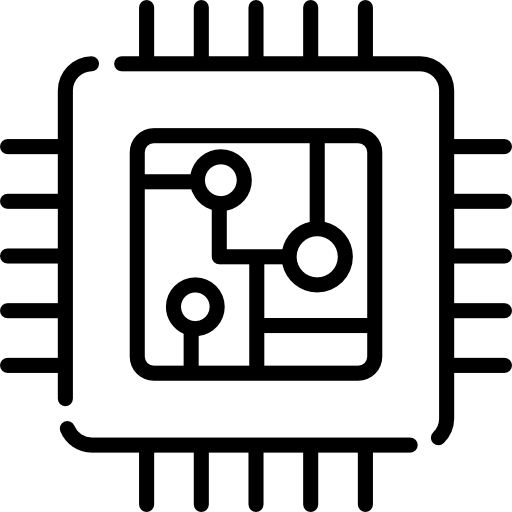
Name: Conor Loughran

**Systems Architecture**

*COM181*



Name: Conor Loughran

B-Number: B00795499

Contents

[Part 1 3](#_Toc46391589)

[Addition of two 4-bit numbers 3](#_Toc46391590)

[Algebraic Formula 4](#_Toc46391591)

[Truth Table 4](#_Toc46391592)

[Subtraction of two 4-bit numbers 5](#_Toc46391593)

[Algebraic Formula 6](#_Toc46391594)

[Truth Table 6](#_Toc46391595)

[Negating a single 4-bit number 7](#_Toc46391596)

[Algebraic Formula 8](#_Toc46391597)

[Truth Table 8](#_Toc46391598)

[Shift Right (divide by two) of a single 4-bit number 9](#_Toc46391599)

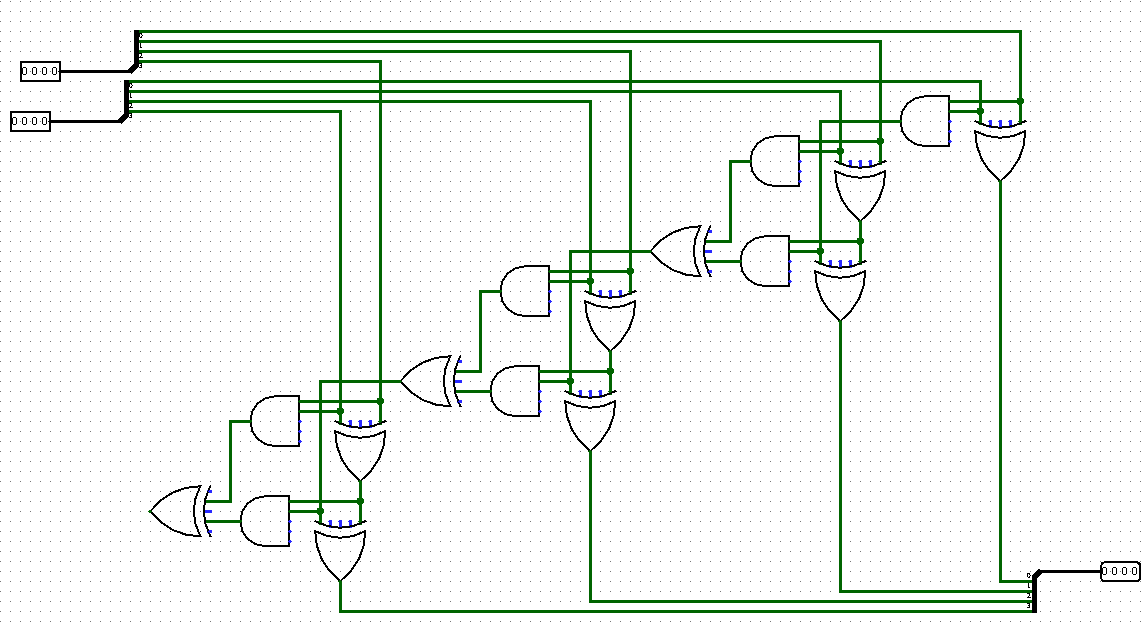
[Algebraic Formula 9](#_Toc46391600)

[Truth Table 9](#_Toc46391601)

[Part 2 10](#_Toc46391602)

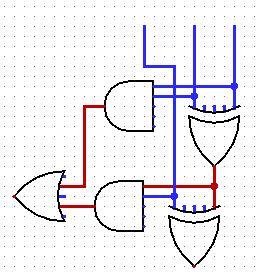
# Part 1

## Addition of two 4-bit numbers



The circuit is made up of the component to the left which is called a full adder. This component will take one bit from each integer and compare them against each other. It starts off with A and B only passing the XOR gate if they are opposite which then gets compared against the bit that has been carried in from the previous adder. If A and B are the same, then an AND gate is used and fed into an OR gate to pass the carried bit onto the next adder in the sequence.

Above is the complete circuit of the addition function. The user selects a 4-bit number from the two selectors at the top which split the signal into four separate wires which allow for them to be compared and the output to be calculated. The output of each XOR gate is collected and fed back into a wire splitter to display the output number.



Cin

B

A

XOR

Cout

Sum

### Algebraic Formula

XOR1 =.B+.A

Cout = (A.B) + (Cin.XOR)

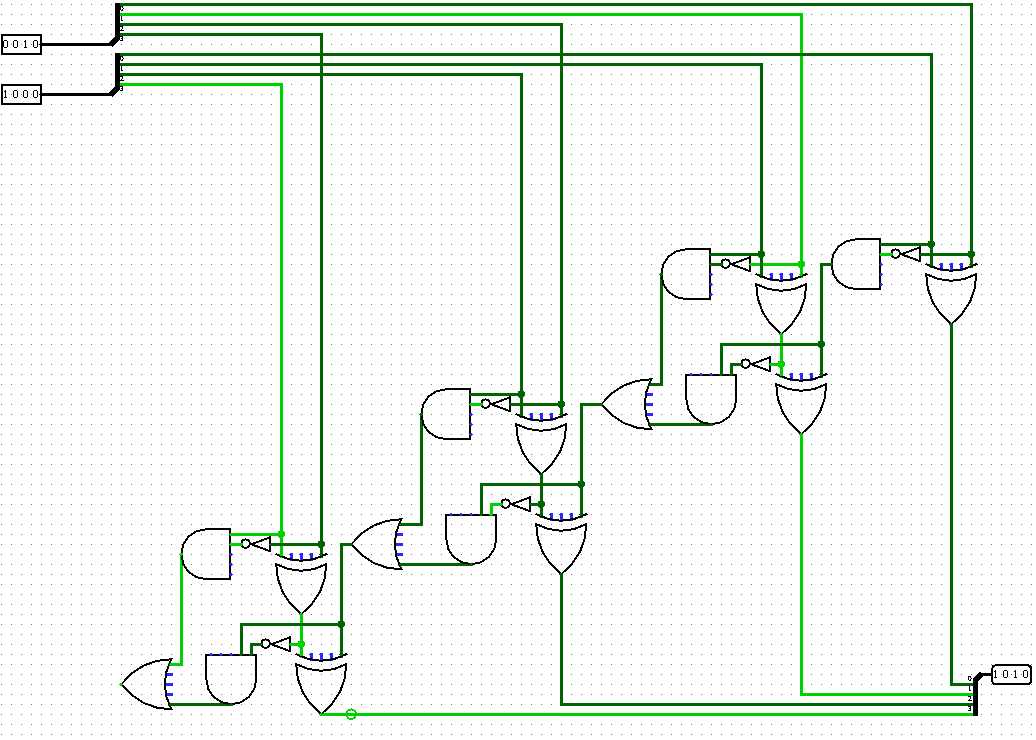
Sum = ()+(.XOR)

Sum = ( . Cin ) + ( . (.B+.A) )

### Truth Table

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | Cin |  |  |  | .A | .B | XOR | XOR.Cin | .XOR | Sum | Cout |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

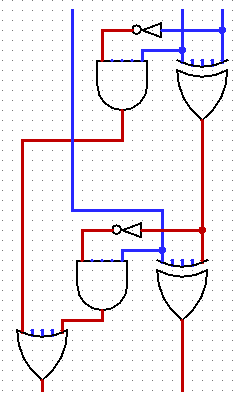
## Subtraction of two 4-bit numbers



Carry *IN*

B

A



XOR2

XOR1

This circuit was used for taking two separate 4-bit numbers and subtracting one from another. If the number being subtracted is larger than the other, then it will return a twos compliment of the negative value. The circuit to the left is the same circuit just re arranged for easier reading. The input A and B are compared against an XOR gate and an AND gate where input A is negative before comparison. The carry bit is then fed into the second XOR gate along with the output of the first XOR gate and that gives the difference. The carry bit and XOR 1 output are also compared against an AND gate to pipe into a final OR gate to check whether there is a bit to be carried over to the next bit comparison.

Carry *OUT*

Difference

### Algebraic Formula

XOR1 =(.B)+(.A)

= XOR1 *(Inverting XOR1 will just give XOR1 but swap the positions of the brackets therefore staying the same)*

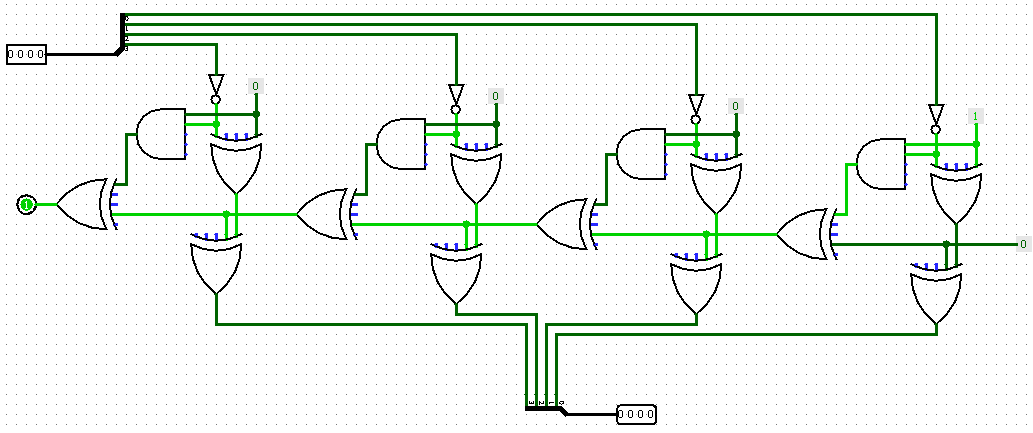
Difference = .C*in* + .XOR1

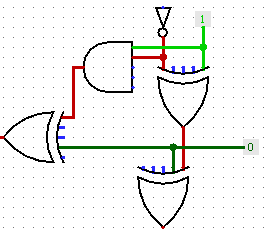
Carry OUT = .B + .C*in*

### Truth Table

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C*in* |  |  | .B | .A | XOR1 | .C*in* | .XOR1 | Difference | Carry OUT |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## Negating a single 4-bit number





A

Constant

Carry

Sum

The circuit above uses five different gates for each bit that is to be negated. To negate a 4-bit number we use Two’s compliment that follows a simple set of rules that can be replicated in a logic circuit. First off, we must flip the bit which is the purpose of the BUBBLE gate. After flipping each bit, we must add one onto the 4-bit number and to do this we use a full adder. The circuit follows the same layout as the full adder circuit above which outputs a single bit at the bottom if the input is zero or if the output is one then it becomes the carry bit and is sent to the next full adder and the output at the bottom remains zero.

### Algebraic Formula

XOR1 =( A.B)+(.)

Sum = .C*in* + .XOR1

Therefore, SUM = (((A.B)+(.)) . C*in) +* ( . ((A.B)+(.)))

Carry OUT = (.Cin) + ()

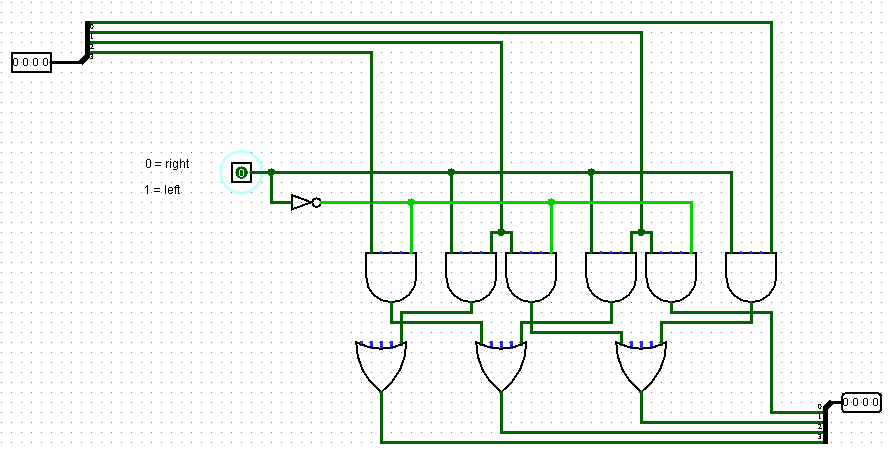
### Truth Table

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C*in* |  |  |  | . | A.B | ( A.B)+(.)  *XOR1* | . Cin | . XOR1 | Sum |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | *0* | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | *1* | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | *0* | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | *0* | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | *0* | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | *0* | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | *0* | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | *1* | 0 | 1 |

Carry Out Bit Truth Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | (.Cin) | () | C*out* |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |

## Shift Right (divide by two) of a single 4-bit number



The circuit above will take a single 4-bit number and shift it either to the right or the left. This is done by using a separate controlling signal to determine whether the shift is happening right or left. Each bit of gets piped into an AND gate which compares against the shift controlling signal. If passed through it hits an OR gate which checks to convert two potential signals into one.

### Algebraic Formula

Input1 = A Input2 = B Shift Direction Control Signal = C

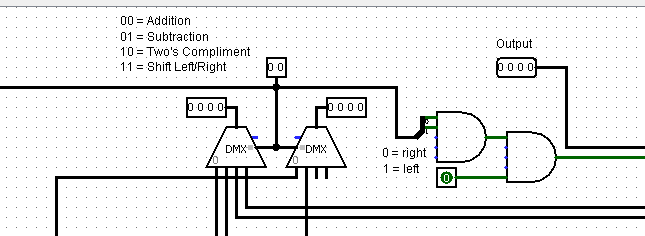
Right Shift Output = (A.C) + (B.C)

Left Shift Output = (A.) + (B.)

### Truth Table

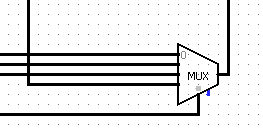
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C |  | (A.C) | (B.C) | (A.) | (B.) | O*right* | O*left* |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

# Part 2



To complete the ALU I took each individual function that I had created and connected them all together using two single inputs and hooked up to one single output. To keep things simple for the end user I had four signals to be changed that would affect the output in the top right. There are two 4-bit integers that can be changed to perform calculations. To change the function that the numbers are being passed through to I have a 2-bit integer that is hooked up to two identical demultiplexers. These multiplexers will send the 4-bit signal out to the corresponding circuit functions, using wire splitters to perform the calculations on each bit individually.

When the ALU is set to negate a number or shift the number it only uses the first 4-bit integer. When shifting left or right a controller signal can be used just below the output that can switch between left and right.



Finally, the output of the ALU is collected at one multiplexer at the bottom of the circuit. This takes the outputs off all circuits but will only send the desired one as it uses the same selector bit as the one used for the demultiplexer.

In the folder there will be two separate .circ files that will allow you to view and interact with the complete ALU and each individual components.