a) The user has requested a design using Logisim of an Arithmetic Logic Unit (ALU) with a 3 bit instruction set to provide 8 operations; below is the design process of the ALU, which consists of using simpler subcircuits as building blocks that individually perform some of the operations and eventually having a complete ALU that can perform all of the user requested operations.

Arithmetic Implementation:

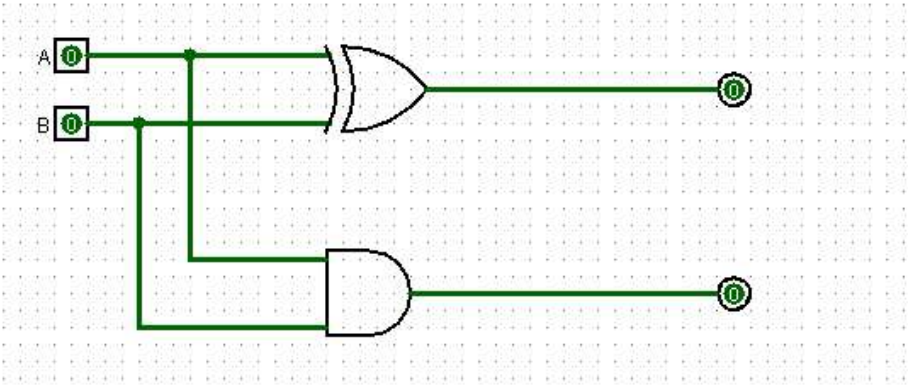


Figure 1: 1-bit Half Adder

The most basic component of the Arithmetic component of the ALU is the Half Adder. It is made by passing two 1-bit values A and B through an XOR gate and an AND gate in parallel as seen in figure 1.

A	B	Sum	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 1: Truth table for the Half Adder

The half adder is abstracted into a model of input pins (A and B) and output pins (Sum and Cout) and is used to build the 1 bit Full Adder bellow:

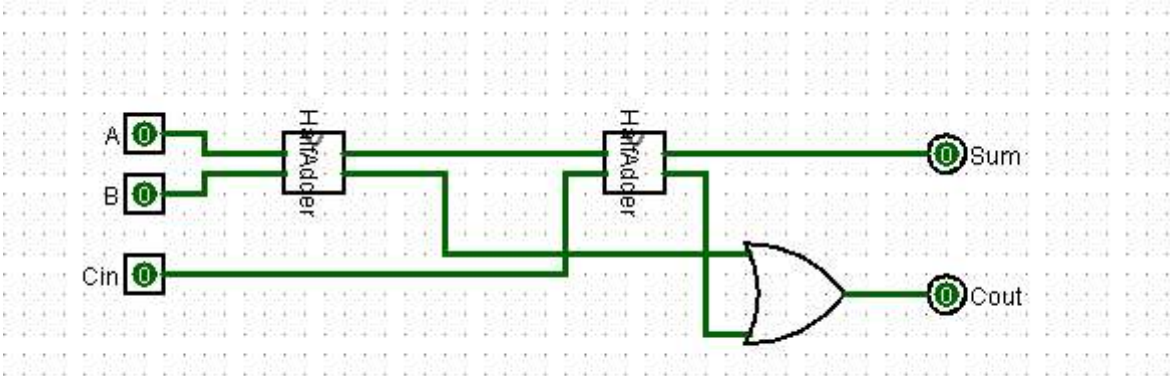


Figure 2: 1-bit Full Adder

The 1 bit full adder, as shown above, shows the functionality of the half adder but not the internal components, allowing us to focus on the functionality of the full adder that now has a carry in pin for an input.

The 1-bit full adder behaves similarly to the half adder except it includes a carry in (Cin) pin.

$S = A \oplus B \oplus Cin$ and $Cout = AB + Cin(A \oplus B)$

Cin	A	B	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 2: Truth Table of the 1 bit full adder

The full adder can be used to perform both addition and subtraction and this component will be generalised into a component with the same pins

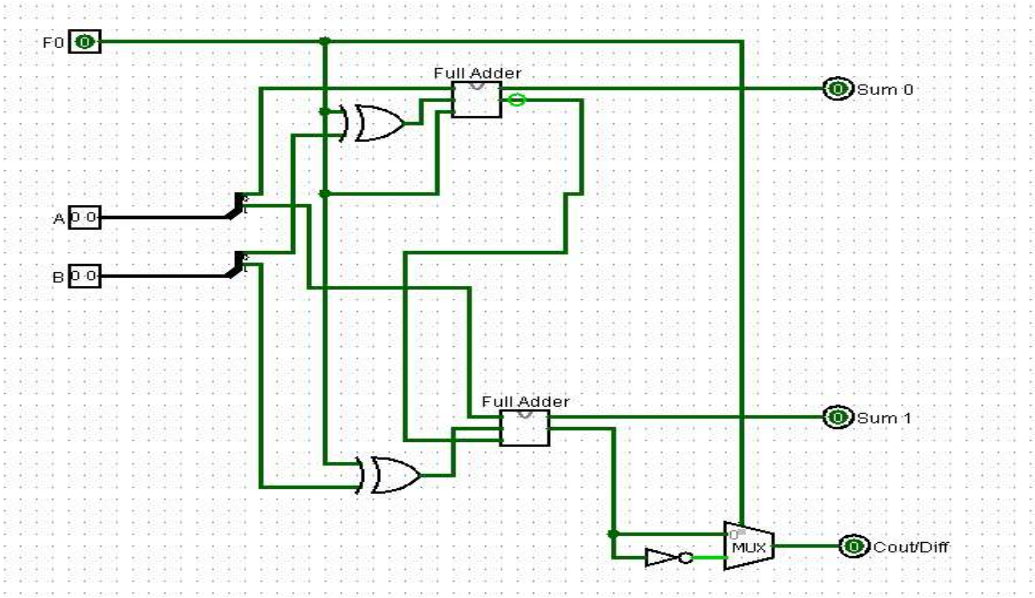


Figure 3: The 2-bit ripple Adder/Subtractor

The Full Adders are used to build the 2 bit adder/subtractor circuit by feeding the carry out of one to the carry in of another. A selection line is used to switch between adding and subtracting modes by performing B XOR Sel to decide the second input as:

$$B = B \oplus 0$$

$$NOT\ (B) = B \oplus 1$$

And therefore the full adders are working out:

Sum = A + NOT(B) + 1 = A - B when F0 = 1 and

Sum = A + B when F0 = 0

Since the carry out from the subtrator is inverted, it must be inverted into the Difference but only if F0 = 1 (only in subtraction mode). Which is why the MUX is used (to invert only if F0 = 1).

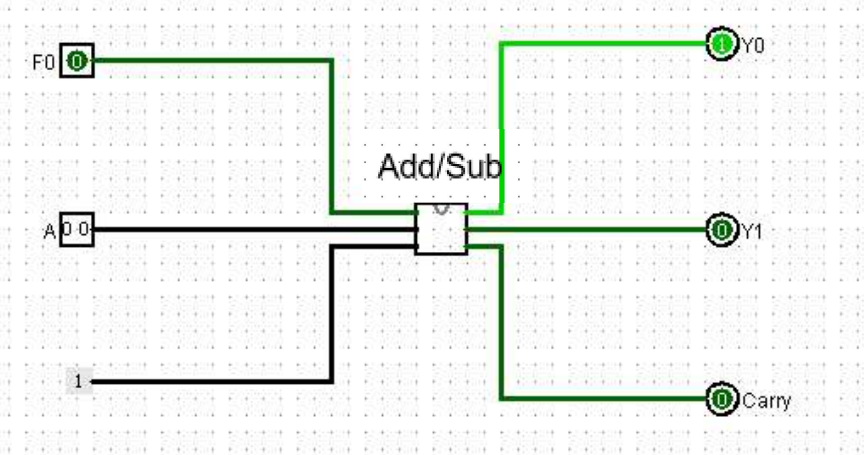


Figure 4: The 2 bit increment/decrement

In figure 4 above, the same general adder and subtractor component is being used to perform the Increment and Decrement functions where B = 1 (Constant input) and F0 is used to select between Sum = A + 1 (When F0 = 0) and Sum = A - 1 (When F0 = 1).

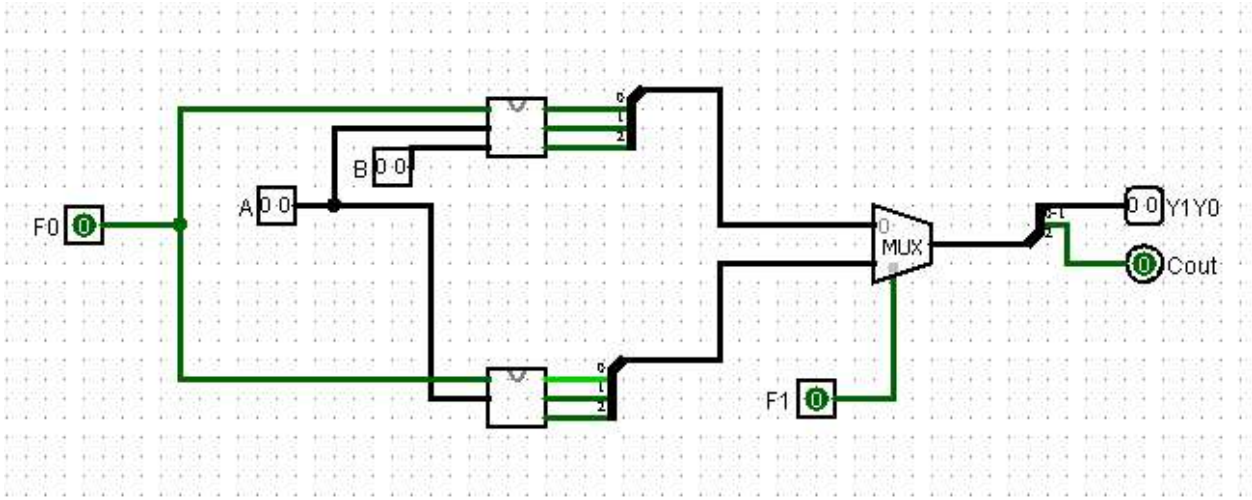


Figure 5: The Arithmetic module of the ALU

Using the abstracted models of the Adder/Subtractor and the Incrementer/Decrementer we arrive at the arithmetic module of the ALU. When F1 = 0 it is in Addition/Subtraction mode and when F1 = 1, it is in Increment/Decrement mode.

Select (Opcode)			Operation	
F2	F1	F0		
Not Implemented Yet	0	0	A+B	Addition
	0	1	A-B	Subtraction
	1	0	A+1	Increment
	1	1	A-1	Decrement

Table 3: Operations of the Arithmetic Module

Multiply/Divider Logic design:

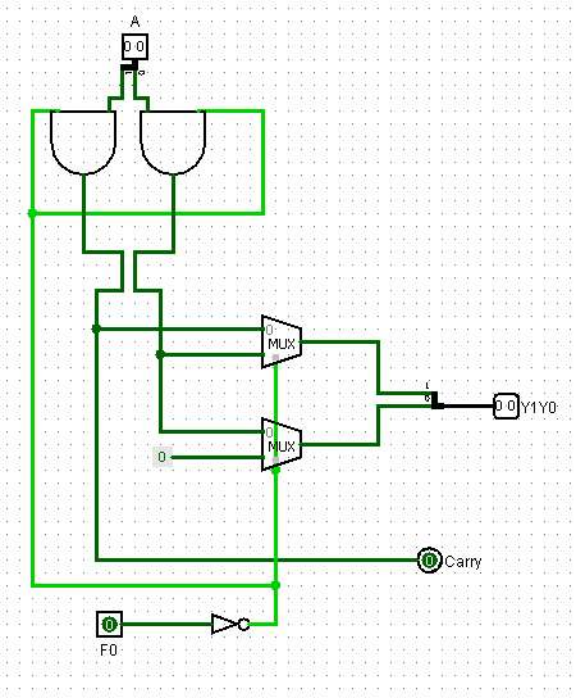


Figure 6: Ax2 or logical 1-bit left shifter

A1	A0	Y0	Y1	C
0	0	0	0	0
0	1	0	1	0
1	0	0	0	1
1	1	0	1	1

Table 4: Truth table of the Ax2 circuit

Figure 6 is the schematic for the logical left shifter that will be used to multiply the A input by 2. As: $2^n \rightarrow 2^{n+1} = 2^n \times 2$ eg. $1_2 << 0b1 = 10_2$,in Table 4; C is the carry and has the same values of A1, C is not a required output, but for clarity is present.

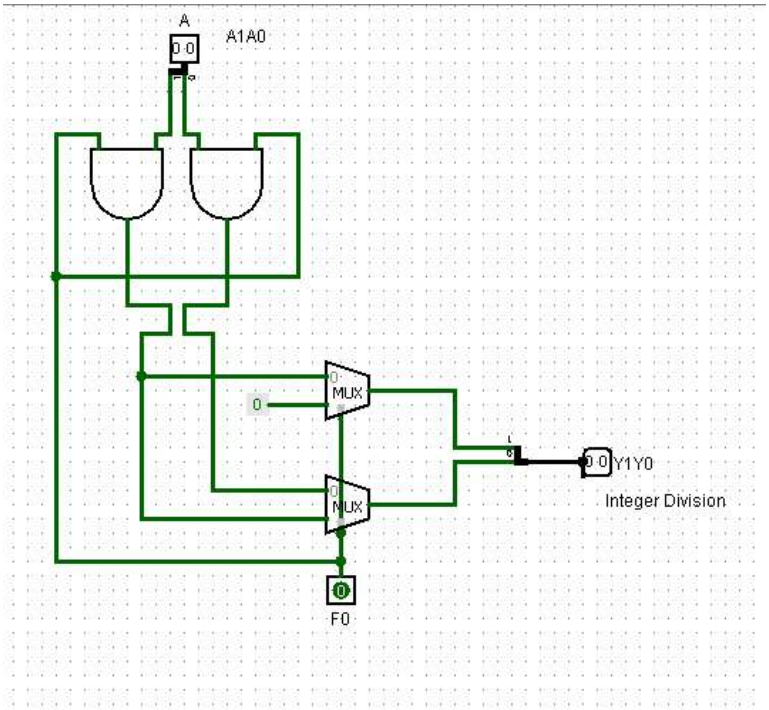


Figure 7: A/2 or logical 1-bit right shifter

The same argument as seen in the multiplier circuit in Figure 6, will now be applied to the divider circuit as seen in Figure 7. As: $A^n \rightarrow A^{n-1} = A \div 2$,eg $10_2 >> 0b1 = 1_2$. Note: this circuit performs integer division, i.e. ½ = 0 and 3/2=1 as seen in Table 5 below.

A1	A0	Y0	Y1
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1

Table 5: Truth table of the A/2 circuit

In Figure 6 and Figure 7; F0 and the AND gates near the 2-bit input A, will ensure the correct operation (multiply or divide) occurs when F2 is HIGH and F1 is LOW, the operation depends on the F0 input.

Bitwise AND/OR Logic:

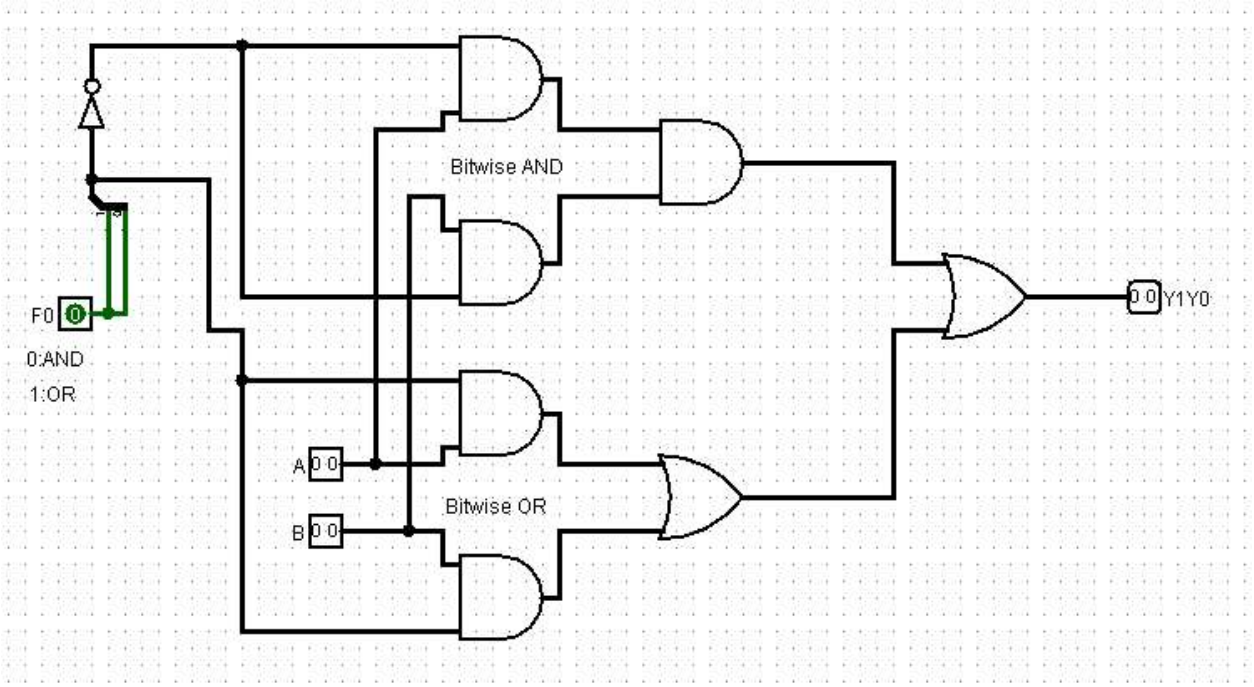


Figure 8: Bitwise AND/OR subcircuit

The 4 AND gates in a column and the F0 input determine which circuit is active (Bitwise AND or Bitwise OR).
Input A and input B are both 2 bits as is the output Y0Y1.
The bitwise AND circuit works by comparing A_x with B_x and only if both are HIGH will the output Y_x goes HIGH.
The bitwise OR circuit works by checking if either A_x 'or' B_x are HIGH, if TRUE the output Y_x will go HIGH.
All Gates are 2-bit inputs and outputs.

The truth table for this operation is intentionally left out as it is a simple operation but requires 32 columns (16 per operation).

Logical Circuit implementation:
This is the design to include Figure 6, Figure 7 and Figure 8 in a larger subcircuit.

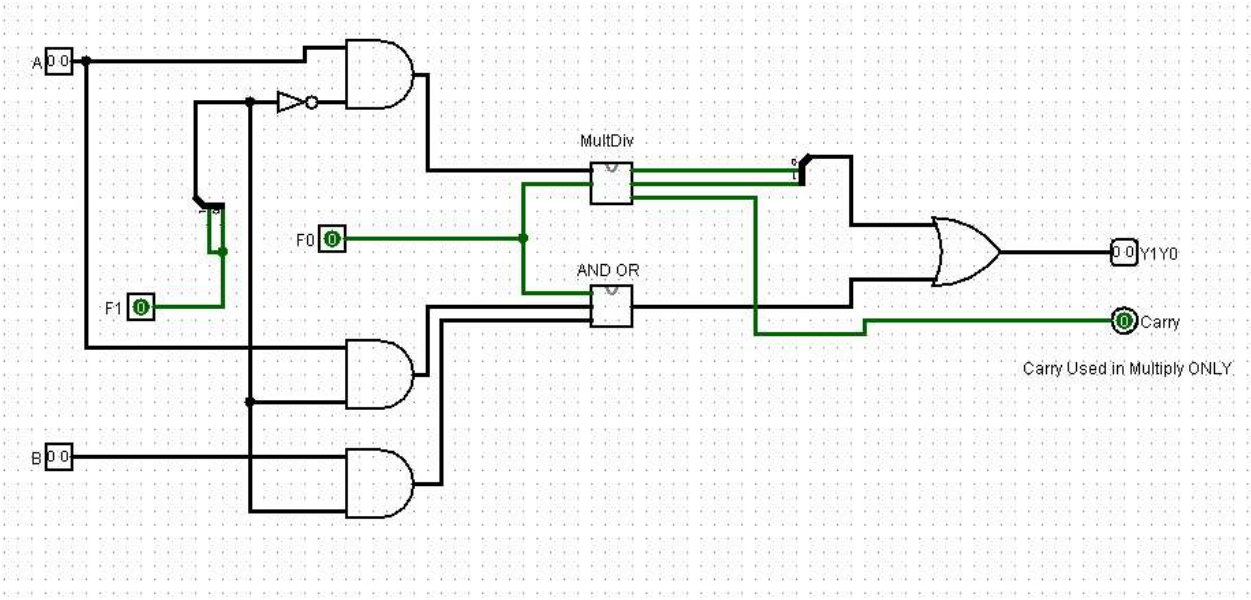


Figure 9: Logic circuit selector

The “MultDiv” and “AND OR” seen in Figure 9 correspond to the respective Logic subcircuits above.

To the left and right of these subcircuits are components used to select the desired logic operation determined by F1. This can also be implemented using MUX’s as seen Figure 5.

A,B are 2-bit inputs and Y0Y1 is a 2-bit output.

The NOT gate before the AND gate ensures that only one of the Logic operations can be performed based on the value of F1.

Select (Opcode)			Operation	
F2	F1	F0		
Not Implemented Yet	0	0	Ax2	A multiply by 2
	0	1	A/2	A divide by 2
	1	0	$A_x \& B_x$	Bitwise AND
	1	1	$A_x \mid B_x$	Bitwise OR

Table 6: Operation and corresponding Select operation

The complete ALU:

Using the same concept as seen in Figure 9 and particularly in Figure 5 the two subcircuits seen in the respective figures can now be combined into the complete ALU as seen in Figure 10 below:

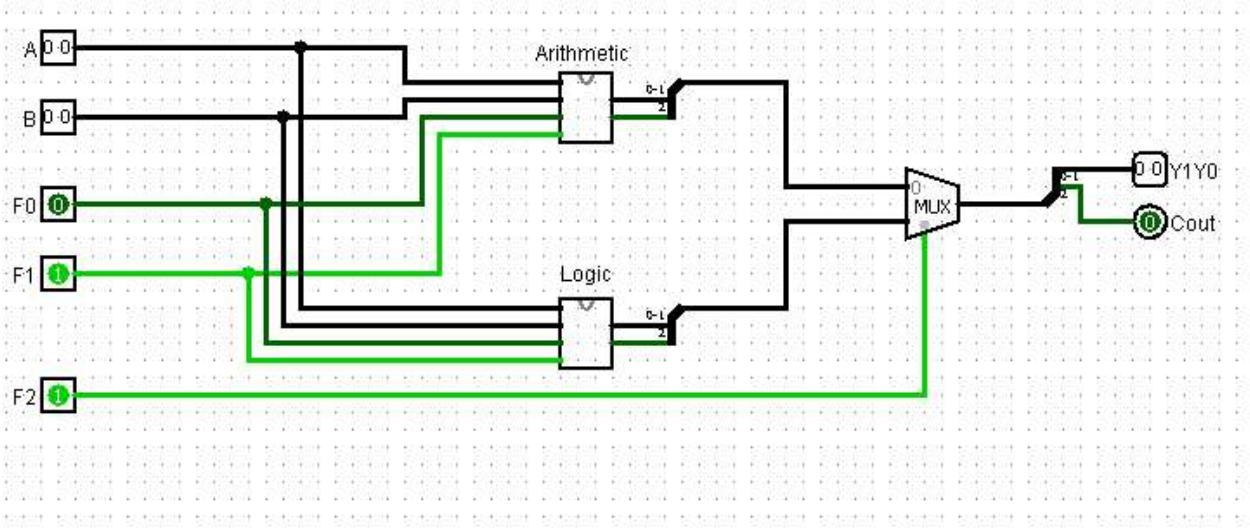


Figure 10: The complete 2-bit Arithmetic and Logic Unit

Again, Cout is not a requirement but is used for clarity during some operations.

Select (Opcode)			Operation	
F2	F1	F0		
0	0	0	A + B	Addition
0	0	1	A - B	Subtraction
0	1	0	A + 1	Increment A
0	1	1	A - 1	Decrement A
1	0	0	Ax2	A multiply by 2
1	0	1	A/2	A divide by 2
1	1	0	$A_x \& B_x$	Bitwise AND
1	1	1	$A_x \mid B_x$	Bitwise OR

Table 7: The operations of the ALU on 2 bit inputs A and B

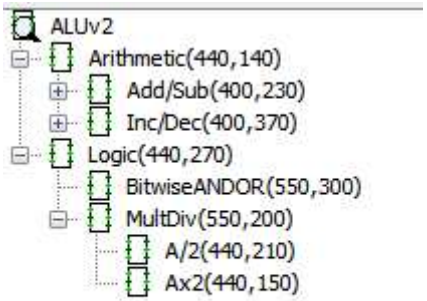
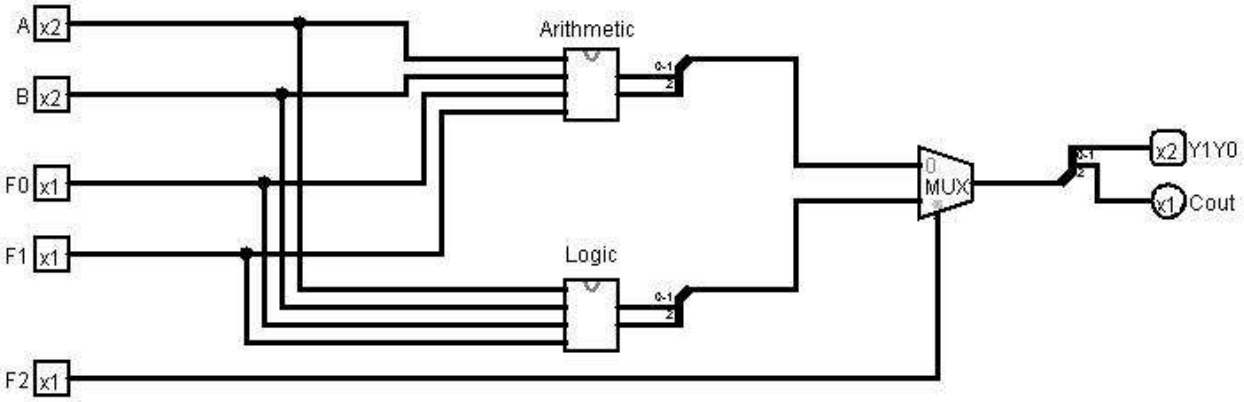


Figure 11: ALU Simulation Tree

b)



Final ALU design.

The complete Truth Table for the ALU designed above:

Note: Y1 ↔ Y0 (swap the Y0 and Y1 label around)

A0	A1	B0	B1	F0	F1	F2	Y0	Y1	Cout
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	1	0	1	0	0
0	0	0	0	0	1	1	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	0	1	0	1	0	0	0
0	0	0	0	1	1	0	1	1	1
0	0	0	0	1	1	1	0	0	0
0	0	0	1	0	0	0	0	1	0
0	0	0	1	0	0	1	0	0	0
0	0	0	1	0	1	0	1	0	0
0	0	0	1	0	1	1	0	0	0
0	0	0	1	1	0	0	0	1	1
0	0	0	1	1	0	1	0	0	0
0	0	0	1	1	1	0	1	1	1
0	0	0	1	1	1	1	0	1	0
0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	1	0	0	0
0	0	1	0	0	1	0	1	0	0
0	0	1	0	0	1	1	0	0	0
0	0	1	0	1	0	0	1	1	1
0	0	1	0	1	0	1	0	0	0
0	0	1	0	1	1	0	1	1	1
0	0	1	0	1	1	1	1	0	0
0	0	1	1	0	0	0	1	1	0
0	0	1	1	0	0	1	0	0	0
0	0	1	1	0	1	0	1	0	0
0	0	1	1	0	1	1	0	0	0
0	0	1	1	1	0	0	1	0	1
0	0	1	1	1	0	1	0	0	0
0	0	1	1	1	1	0	1	1	1
0	0	1	1	1	1	1	1	1	0
0	1	0	0	0	0	0	0	1	0

0	1	0	0	0	0	1	0	0	1
0	1	0	0	0	1	0	1	1	0
0	1	0	0	0	1	1	0	0	0
0	1	0	0	1	0	0	0	1	0
0	1	0	0	1	0	1	1	0	0
0	1	0	0	1	1	0	1	0	0
0	1	0	0	1	1	1	0	1	0
0	1	0	1	0	0	0	0	0	1
0	1	0	1	0	0	1	0	0	1
0	1	0	1	0	1	0	1	1	0
0	1	0	1	0	1	1	0	1	0
0	1	0	1	1	0	0	0	0	0
0	1	0	1	1	0	1	1	0	0
0	1	0	1	1	1	0	1	0	0
0	1	0	1	1	1	1	0	1	0
0	1	1	0	0	0	0	1	1	0
0	1	1	0	0	0	1	0	0	1
0	1	1	0	0	1	0	1	1	0
0	1	1	0	0	1	1	0	0	0
0	1	1	0	1	0	0	1	0	0
0	1	1	0	1	0	1	1	0	0
0	1	1	0	1	1	0	1	0	0
0	1	1	0	1	1	1	1	1	0
0	1	1	1	0	0	0	1	0	1
0	1	1	1	0	0	1	0	0	1
0	1	1	1	0	1	0	1	1	0
0	1	1	1	0	1	1	0	1	0
0	1	1	1	1	0	0	1	1	1
0	1	1	1	1	0	1	1	0	0
0	1	1	1	1	1	0	1	0	0
0	1	1	1	1	1	1	1	1	0
1	0	0	0	0	0	0	1	0	0
1	0	0	0	0	0	1	0	1	0
1	0	0	0	0	1	0	0	1	0

1	0	0	0	0	1	1	0	0	0
1	0	0	0	1	0	0	1	0	0
1	0	0	0	1	0	1	0	0	0
1	0	0	0	1	1	0	0	0	0
1	0	0	0	1	1	1	1	0	0
1	0	0	1	0	0	0	1	1	0
1	0	0	1	0	0	1	0	1	0
1	0	0	1	0	1	0	0	1	0
1	0	0	1	0	1	1	0	0	0
1	0	0	1	1	0	0	1	1	1
1	0	0	1	1	0	1	0	0	0
1	0	0	1	1	1	0	0	0	0
1	0	0	1	1	1	1	1	1	0
1	0	1	0	0	0	0	0	1	0
1	0	1	0	0	0	1	0	1	0
1	0	1	0	0	1	0	0	1	0
1	0	1	0	0	1	1	1	0	0
1	0	1	0	1	0	0	0	0	0
1	0	1	0	1	0	1	0	0	0
1	0	1	0	1	1	0	0	0	0
1	0	1	0	1	1	1	1	0	0
1	0	1	1	0	0	0	0	0	1
1	0	1	1	0	0	1	0	1	0
1	0	1	1	0	1	0	0	1	0
1	0	1	1	0	1	1	1	0	0
1	0	1	1	1	0	0	0	1	1
1	0	1	1	1	0	1	0	0	0
1	0	1	1	1	1	0	0	0	0
1	0	1	1	1	1	1	1	1	0
1	1	0	0	0	0	0	1	1	0
1	1	0	0	0	0	1	0	1	1
1	1	0	0	0	1	0	0	0	1
1	1	0	0	0	1	1	0	0	0
1	1	0	0	1	0	0	1	1	0

1	1	0	0	1	0	1	1	0	0
1	1	0	0	1	1	0	0	1	0
1	1	0	0	1	1	1	1	1	0
1	1	0	1	0	0	0	1	0	1
1	1	0	1	0	0	1	0	1	1
1	1	0	1	0	1	0	0	0	1
1	1	0	1	0	1	1	0	1	0
1	1	0	1	1	0	0	1	0	0
1	1	0	1	1	0	1	1	0	0
1	1	0	1	1	1	0	0	1	0
1	1	0	1	1	1	1	1	1	0
1	1	1	0	0	0	0	0	0	1
1	1	1	0	0	0	1	0	1	1
1	1	1	0	0	1	0	0	0	1
1	1	1	0	0	1	1	1	0	0
1	1	1	0	1	0	0	0	1	0
1	1	1	0	1	0	1	1	0	0
1	1	1	0	1	1	0	0	1	0
1	1	1	0	1	1	1	1	1	0
1	1	1	1	0	0	0	0	1	1
1	1	1	1	0	0	1	0	1	1
1	1	1	1	0	1	0	0	0	1
1	1	1	1	0	1	1	1	1	0
1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	0	1	1	0	0
1	1	1	1	1	1	0	0	1	0
1	1	1	1	1	1	1	1	1	0