CS 3410 P1 : ALU

Akshay Yadava

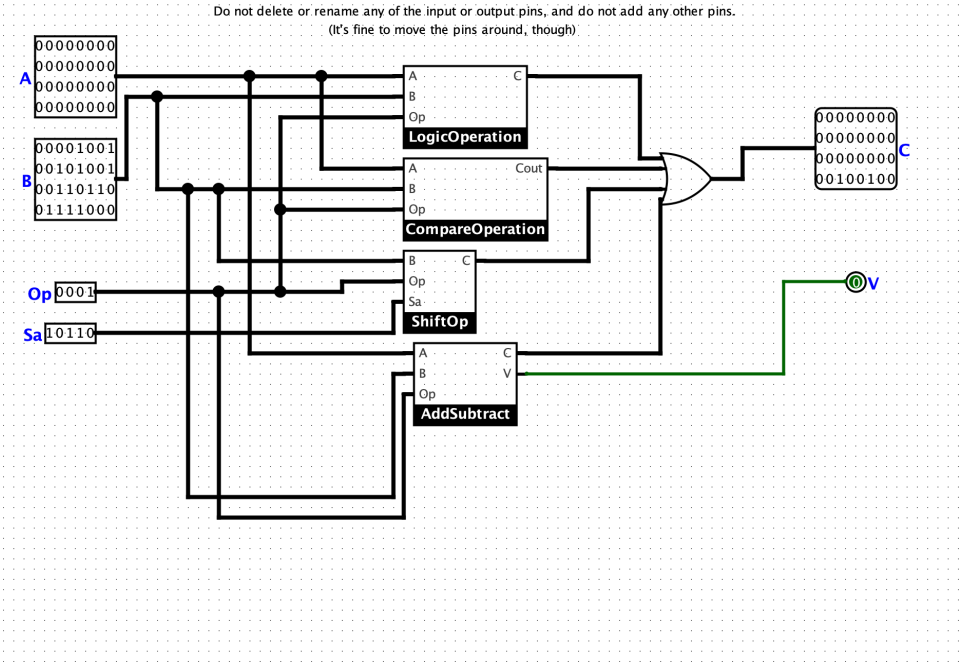
aay29

February 16, 2022

1 Overview

The purpose of this project was to implement an Arithmetic Logic Unit in the software Logisim. An ALU is a combinational circuit that may perform arithmetic and bitwise operations on binary numbers. In this project, comparison operations (less than, equal to, greater than, not equal), logic operations (and, or, xor, nor), simple arithmetic (addition, subtraction), and shift operations (left shift, right shift arithmetic, right shift logical) were all implemented. Depicted below is a picture of the entire ALU circuit.

ALU Circuit

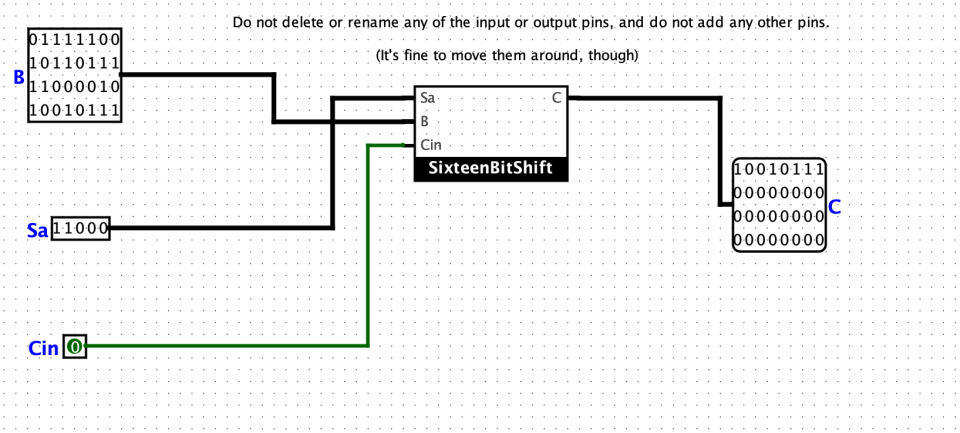


2 Component Design Documentation

1. LeftShift32

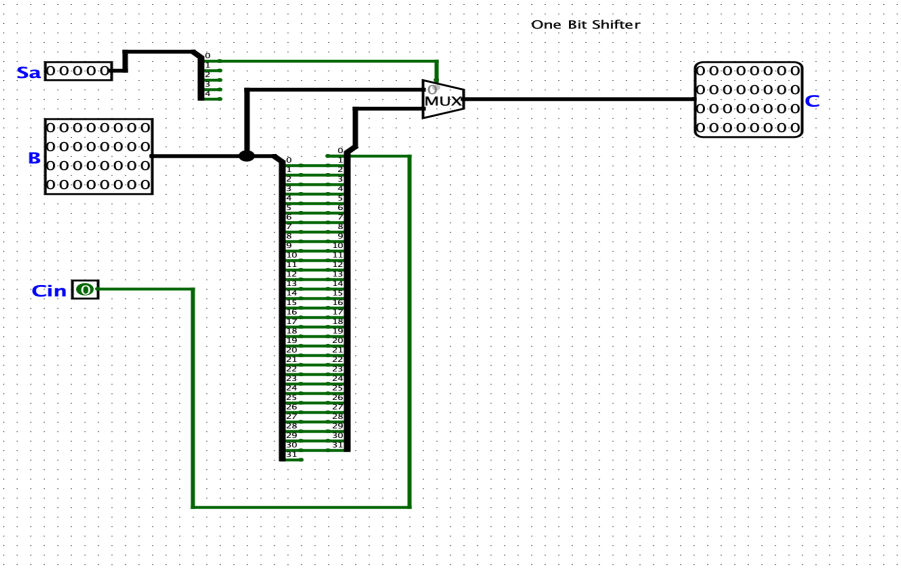
The first major component implemented in this project was the LeftShift32 circuit. Listed below is a picture of the entire circuit.

LeftShift32



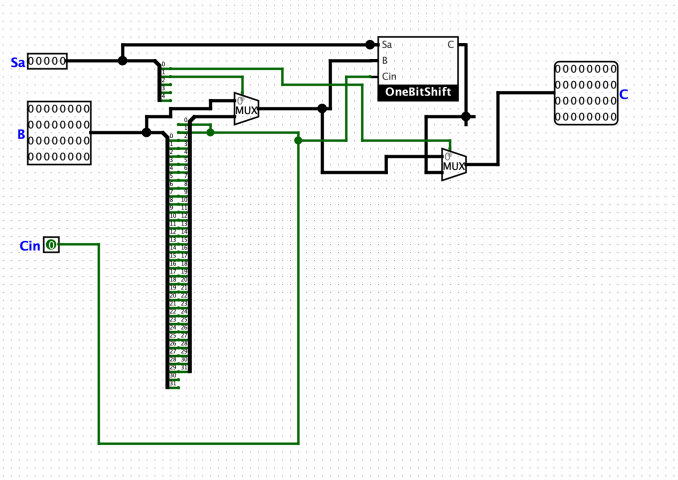
The intuition for creating the left shifter was to initially create a one bit left shifter, then use that circuit to create a 2 bit shifter, then a 4 bit shifter, 8 bit shifter, and then a 16 bit shifter. This circuit will help handle all the shift operations from the opcode. The first circuit created to make the 32 bit shifter is the 1 bit shifter depicted below.

One Bit Shifter

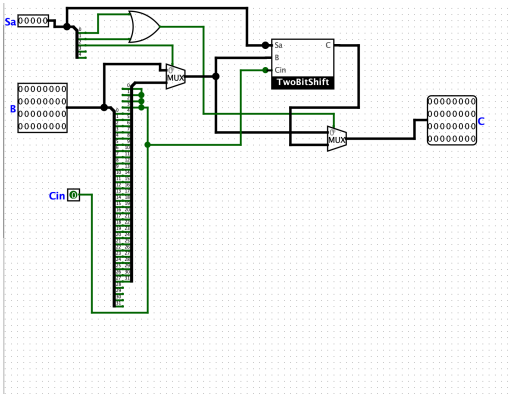


The one bit shifter functions by shifting the bits left one and filling in the least significant bit with the carry in. There is also a mux that uses the first bit of the shift input (Sa) to decide to choose the shifted version of B or not. This is then fed into the output. The intuition for the left shifter is to once again shift the bits by two, use a mux to decide to use the shifted value or not, and then feed that output of the shift into the one bit shifter and utilize another mux to decide to shift the already shifted two bit value by one based on the 0th bit of Sa. The intuition to use the prior bit shifter to implement the next shifter is used throughout. This way, with a 16 bit shifter, 8 bit shifter, 4 bit shifter, 2 bit shifter, and 1 bit shifter, any input can be shifted by the desired bits (0-31). The rest of the shifters are depicted below following this design process.

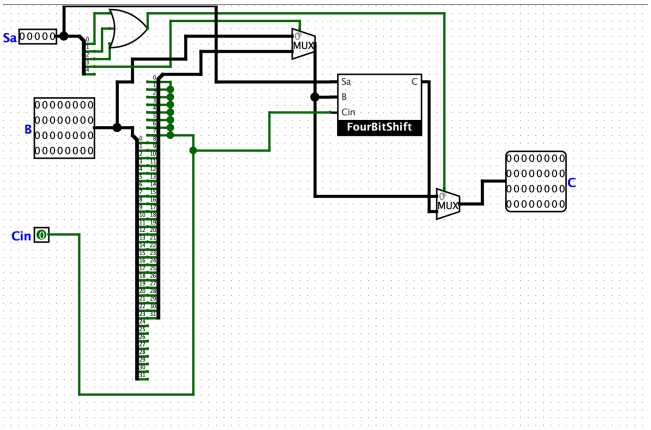
2 Bit Shifter



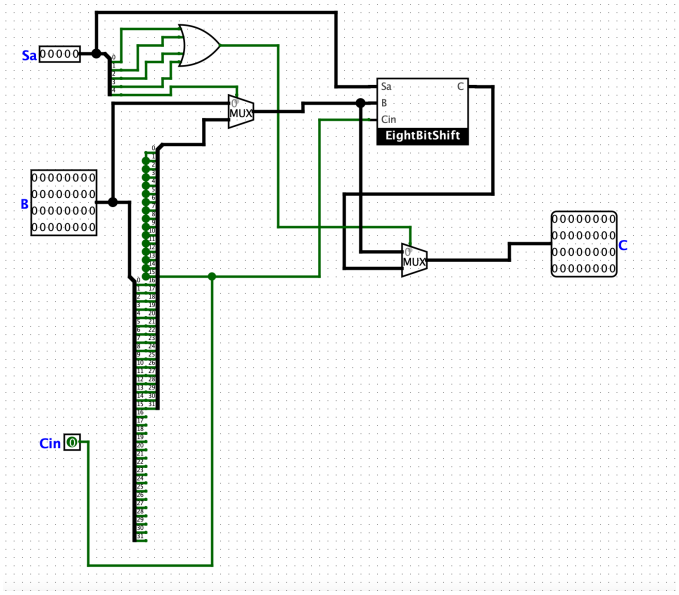
4 Bit Shifter



8 Bit Shifter

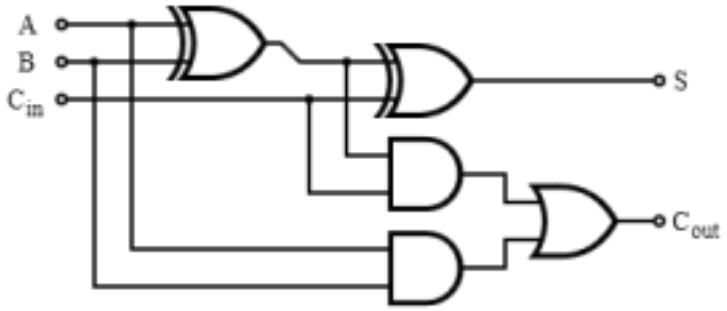


16 Bit Shifter



2. Add32

The next circuit implemented in this project was the Add32 circuit. To implement this, the idea was to implement a one bit adder, then use this to build a two bit adder, then a four bit adder, and so on until 32 bit, a similar flow to implementing the left shift 32 circuit.

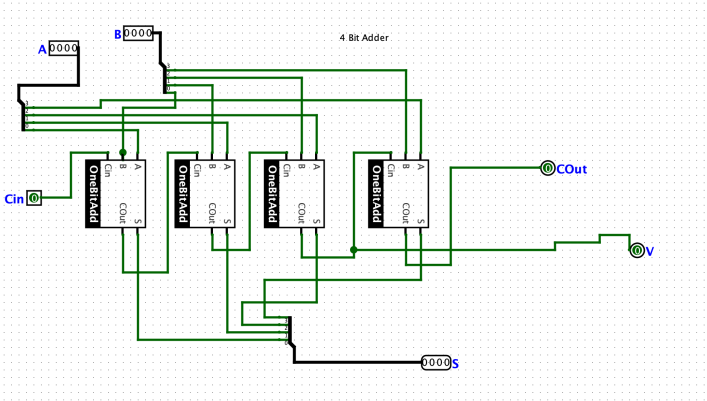


In the above diagram, *A* and *B* are the 1-bit inputs, while C_{in} is the carry-in bit. *S* is the output bit, while C_{out} is the carry-out bit. This is the optimal circuit for the following truth table:

A	*B*	C_{in}	*S*	C_{out}
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

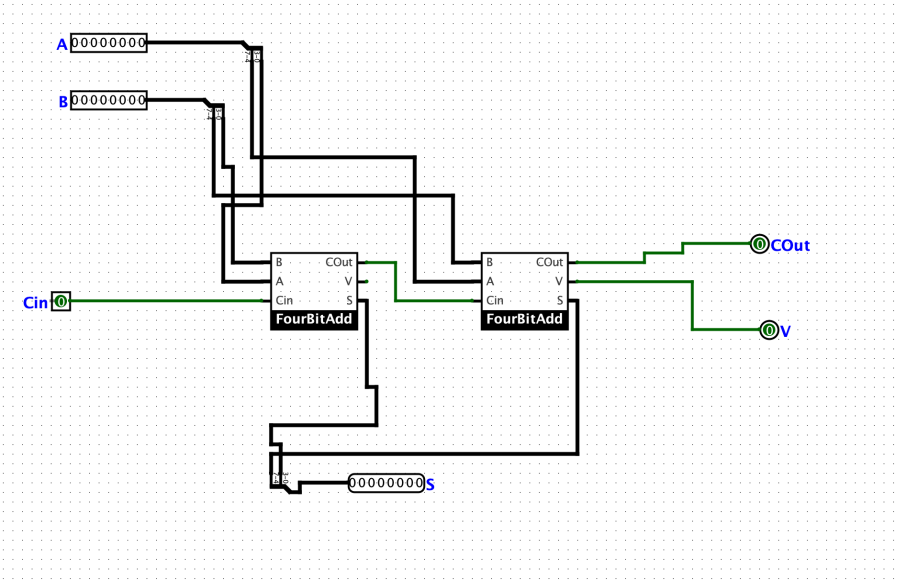
With the 1 bit adder implemented, it was used to create a four bit adder composed as follows:

4 Bit Add

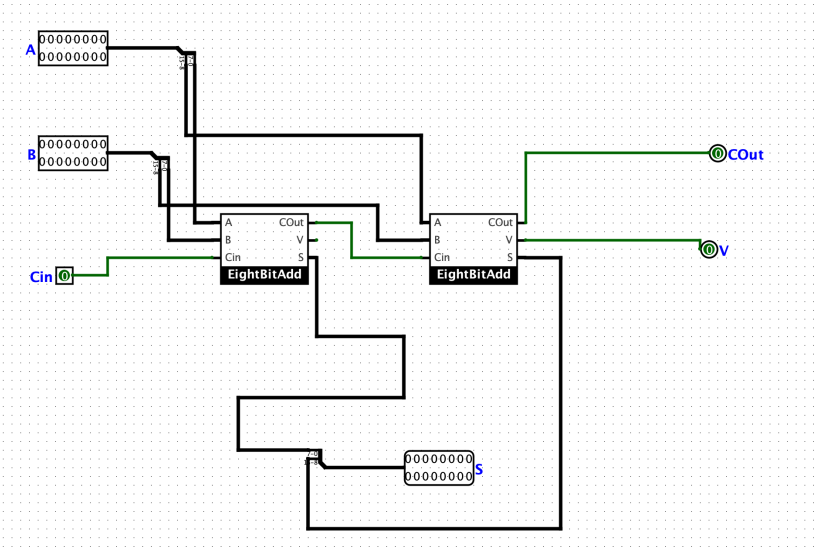


A splitter is used to decompose the 4 bits of A and B. Then, each of the bits is passed into the one bit adder with the correct indexes. The carry out of each operation is passed to the carry in of the next operation. The Overflow is also documented. After all the bits have been added bitwise with the correct carry-ins, the sum is composed by using a splitter to have the correct bitwise value of the addition. The same flow and justifications are used in the following circuits to build up to the full 32 bit adder.

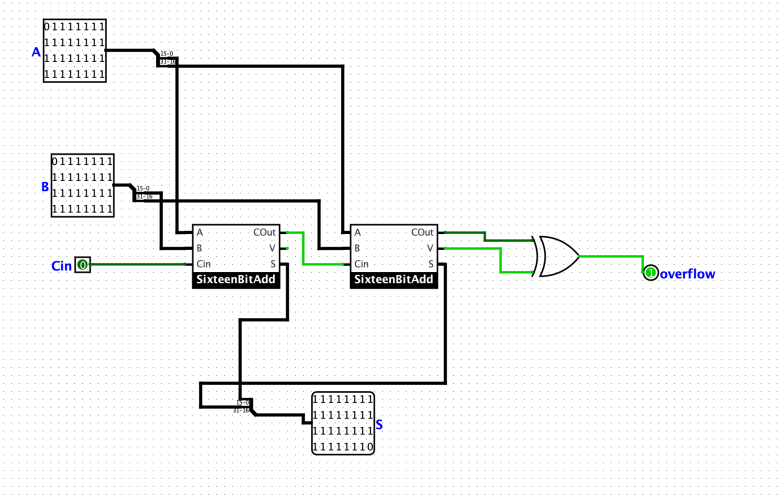
8 Bit Add



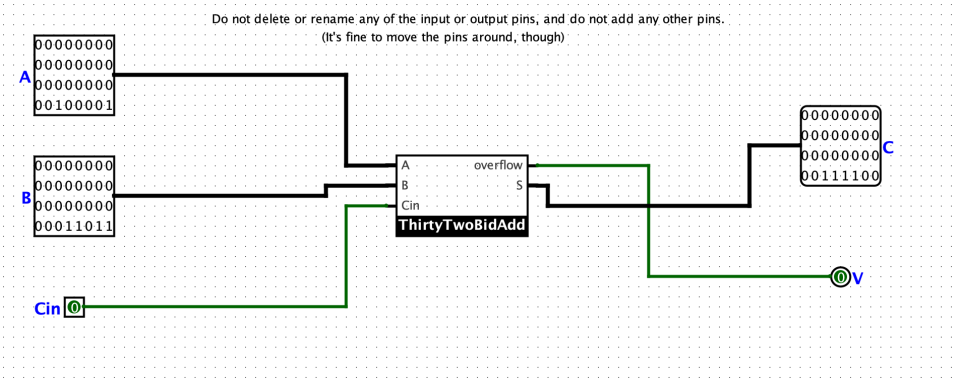
16 Bit Add



32 Bit Add



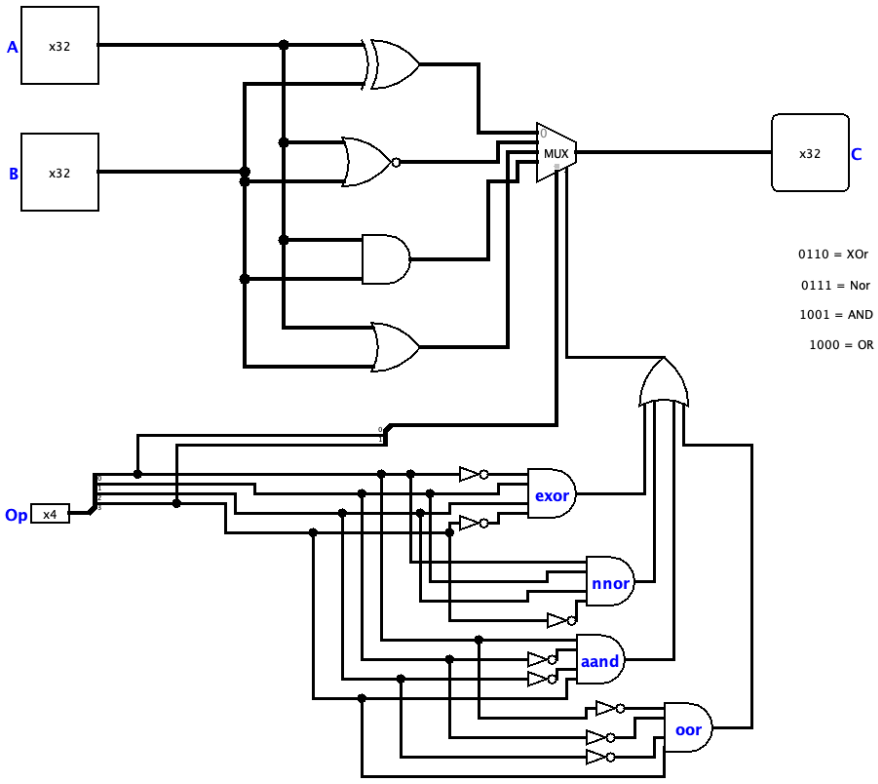
Final Add32



3. Logic Operation

The next circuit implemented was the Logic Operation circuit. This circuit handles the xor, nor, and, and or operations as instructed by the opcode. Depicted below is the logic operation circuit.

Logic Operation Circuit



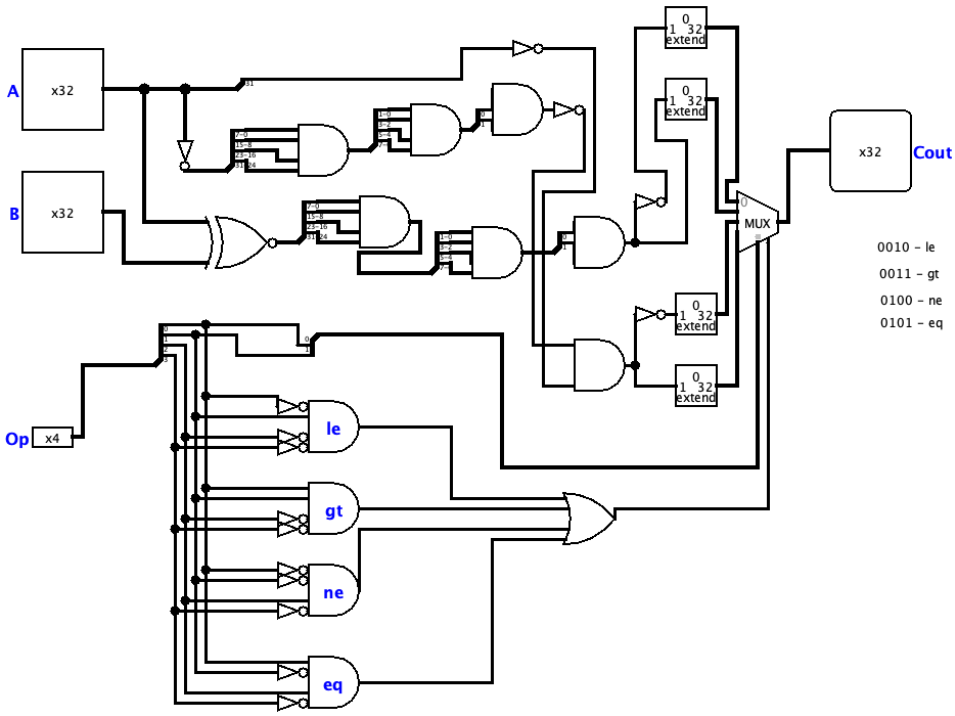
The intuition for this circuit was to initially compute all logic operations (xor, nor, and, and or) and then use a mux to decide on which operation to output as depicted by the opcode. Given the opcodes are 0110, 0111, 1001, and 1000, we can see that the 0th and 3rd bit are all different in each opcode. Therefore, we can use the 0th and 3rd bit of the opcode as a selector for the mux. The various bit combos correspond to the correct logic operation from the selector mux. Furthermore, an enabler was used on the mux to ensure that the operation is only completed if the opcode is valid. Therefore, there are 4 different opcodes that could be valid, and an or gate is used to ensure at least one of these opcodes is satisfied. If none of the opcodes are satisfied, this operation will output only 0s.

OpCode	Mux
0110	0
0111	1
1001	2
1000	3
Otherwise	Disable

4. Compare Operation

The next circuit implemented was the compare operation circuit as depicted below.

Compare Circuit



Given the input A, we first check to see if A is bigger than 0. Given the numbers are represented in two's complement, if A is bigger than 0, its 31st bit will be 0, while at least one of the other bits must be nonzero. If A satisfies these conditions, the A is positive, otherwise A is negative. We use the AND gates to continually assert all the remaining bits are the same or not. The result of this operation is a 32 bit 0 or 1 because of the sign extender which will be the output.

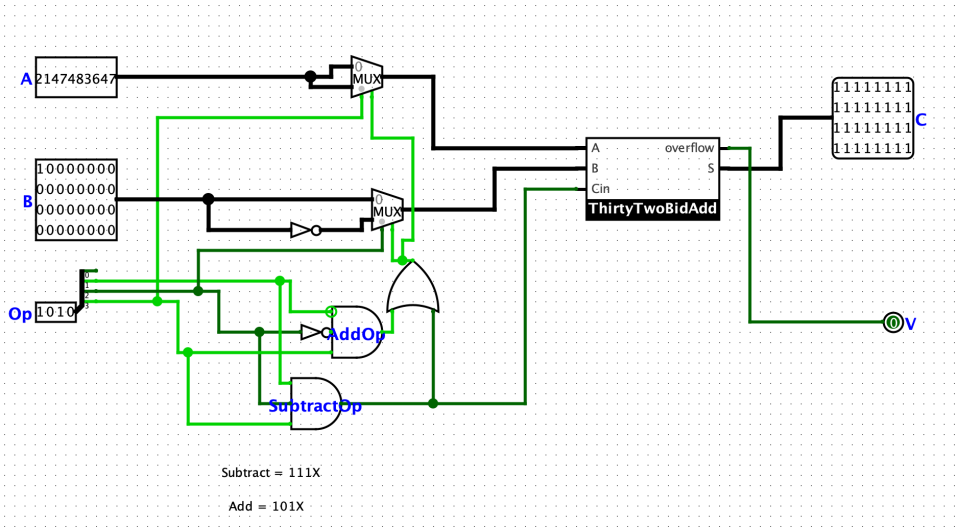
Next we must compare A and B. If A is equal to B, A^B produces a 32-bit 0. We continually use AND gates to check if A^B is 0. If yes, A is equal to B, yet if A^B is not 0, then A and B are not equal. This is because the XOR gate will only output a 1 if the bits are different.

After we have done the comparisons, we must choose which input to select. This is based on the opcode again. As we can see, the opcodes are 0010, 0011, 0100, and 0101. We can see the opcodes all differ in the 1st and 2nd bit, therefore we will use these bits as the selector bits to the mux. Furthemore, there is an enabler on the mux that will only allow the operation to be completed if one of the opcodes is asserted.

OpCode	Mux
0100	0
0101	1
0010	2
0011	3
Otherwise	Disable

5. Add/Subtract

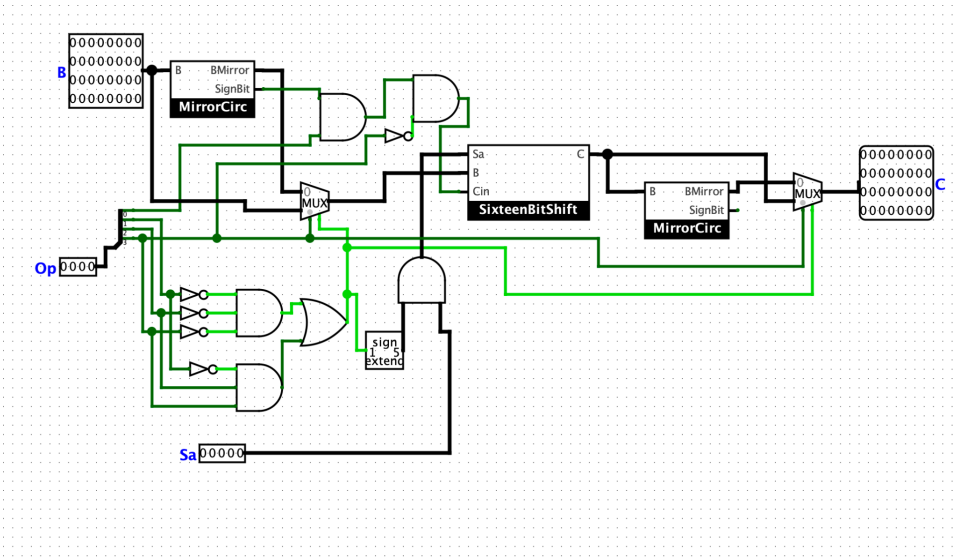
The next part of the ALU implemented was the Add/Subtract circuit as depicted below.



The intuition here is that subtraction is done by negating B, adding 1, and summing it to A. Initially, there is a mux that decides between the negated version of B or unnegated version of B depending on the 3rd bit of the opcode, which indicates either addition or subtraction. If it is subtraction, the negated version of B is chosen, and there is a Cin passed to the ThirtyTwoBitAdd circuit. If it is addition, the regular version of B is used and addition is done as usual. The mux on the top of the screen has an enabler that is only true if the opcode is either addition or subtraction. The selector bit does not matter since A is passed into both inputs of the mux. This enabler was used to ensure that A is passed only if the opcode is correct for the operation, and nothing else is outputted if the opcode is incorrect. This will be explained in the final portion of this document.

Op	Cin	Mux	A	B
111X	1	1	A	~B
101X	0	0	A	B
Otherwise	0	0/1	0	0

6. Shift Operation



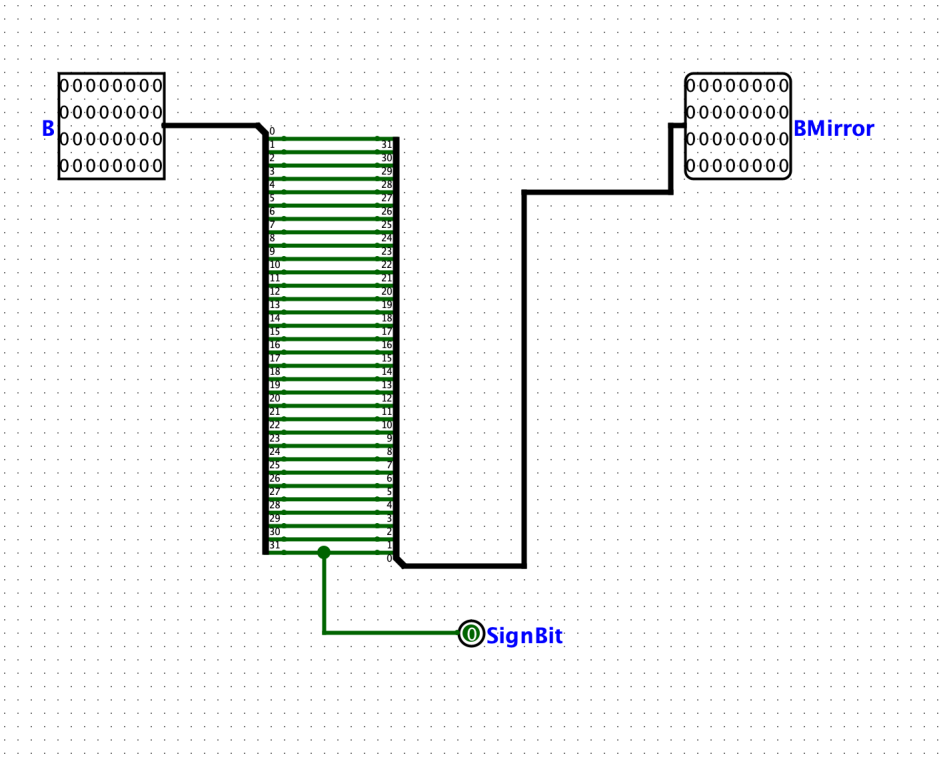
Initially, we use a splitter on the opcodes to ensure that we are completing a shift operation. If the opcode satisfies 000X or 110X, we know we are doing a shift. If this condition is asserted, we use an and gate and sign extend 1 to five bits to pass in the shift value into the shifter.

If we are completing a left shift (110X), we use the regular version of B to complete the shift. Yet, if we are completing a right shift (000X), we must mirror B, shift, and then mirror it once again to get the correct right shift.

If the first bit of the opcode is one and there is a sign but, this sign bit is passed into the Cin for the shift. The initial mux decides to use the mirrored version of B or not depending on the last bit of the opcode. The second mux does the same, yet has an enabler to ensure the opcode is indicating a shift.

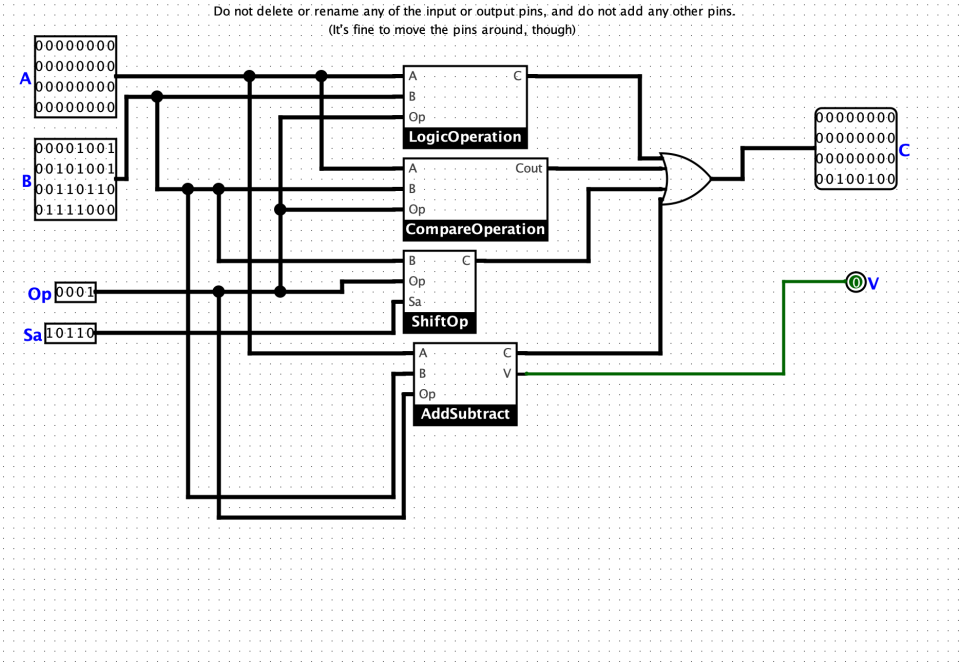
Op	Left-Mux	Right-Mux	Cin	B	Sa
0000	0	0	0	B	Sa
0001	0	0	1	B	Sa
110X	1	1	0/1	B	Sa
Otherwise	Disable	Disable	0/1	B	Sa

Mirror Circuit



This is the mirror circuit as used in the prior circuit. All this does is mirror the bits of B to ensure right shifting can be done using the LeftShift32 circuit. This circuit also keeps track of the sign bit of B to keep track of when to pass Cin when doing an arithmetic right shift.

7. Final ALU



Here is the final picture of the ALU. Since each operation is asserted with an enabler in each circuit, the given subcircuit will only output the corresponding operation if the op code is asserted, and otherwise 0. Therefore, we can pass A, B, Op, and Sa into each of the subcomponents and use a final or gate connecting all the subcomponents to get the corresponding operation.

8. Justification

While I believed I may have used too many gates to assert that a certain opcode is valid when completing an operation and there is probably a more efficient way to complete this, I believe this ALU is more reliable now and will ensure that only the correct calculations are outputted when they must be.