Name: Khau Lien Kiet
Student ID: V202000119
Lab Instructor: Truong Tuan Vu
College: CECS

COMP2020 Design Documentation

Adopted from VinUniversity
October 19, 2021

1. Overview

In this project 1, we are aiming to design a RISC-V ALU, which is an Arithmetic Logic Unit. In this ALU requirement, we must perform four main core elements that are ShiftCircuit (Shift32), Add32, LogicGate32, Compare32. Details in each component would be defined in the later section.

2. Component Design Documentation

As mentioned above, there are four key components establishing the full complete ALU for this project. Although there is the ALU Implementation in Logism, we could not apply by default as it is not the objective of the project. Hence, it is an opportunity to build it from scratch.

A. Objectives

- Input: A, B, Op (Operation), Sa
- Output: C, V (considering an overflow)
- Full Op Requirements:

Op	name	C	V
0100	and	C = A & B	V = 0
0101	or	C = A | B	V = 0
000x	shift left logical	C = B << Sa	V = 0
1010	xor	C = A ^ B	V = 0
1011	nor	C = ~(A | B)	V = 0
1100	shift right logical	C = B >>> Sa	V = 0
1101	shift right arithmetic	C = B >> Sa	V = 0
1000	ne	C = (A != B) ? 000...0001 : 000...0000	V = 0
1001	eq	C = (A == B) ? 000...0001 : 000...0000	V = 0
1110	le	C = (A ≤ 0) ? 000...0001 : 000...0000	V = 0
1111	gt	C = (A > 0) ? 000...0001 : 000...0000	V = 0
011x	subtract	C = A - B	V = overflow
001x	add	C = A + B	V = overflow

- Overall: According to the ALU32 instruction, to satisfy all of the operations inside the ALU, we have to group the same logical function into one circuit for easier performance. Thus, I divide it into four group based on my analysis:
    - ShiftCircuit: (000x) Shift Left Logical, (1100) Shift Right Logical, (1101) Shift Right Arithmetic
    - Add32: (011x) Subtract, (001x) Add
    - LogicGate32: (0100) And, (0101) Or, (1010) XOR, (1011) NOR
    - Compare32: (1000) Ne, (1001) Eq, (1110) Le, (1111) Gt

***Disclaimer:

- Not duplicate the components
- Not use unnecessarily large MUX(es)
- According to the details of Specifications on Canvas page
-

B. Implementation Details

Based on my group strategy, each subsection below would be clarified how can I get to the main concept and idea for building all 4 key component circuits.

2.1. ShiftCircuit (Shift32)

- Outcome: Shift Circuit must combine the Shift Left Logical and Shift Right Logical as well as the Shift Right Arithmetic. Hence, I have to design the Shift Circuit satisfied both three kinds of shifting operations.
- Reuse the Shift Left Logical from the Wet Lab:

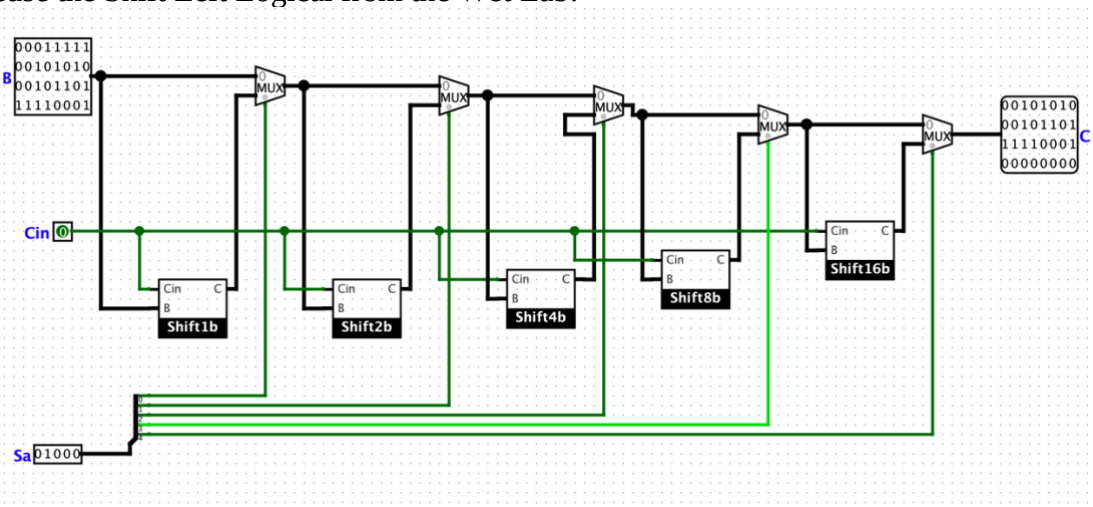


Figure 1: Shift Left Logical Subcircuit

- To determine how Shift Right can be executed in the ShiftCircuit, I come up with the Mirror subcircuit in order to flip backwards the order of the value

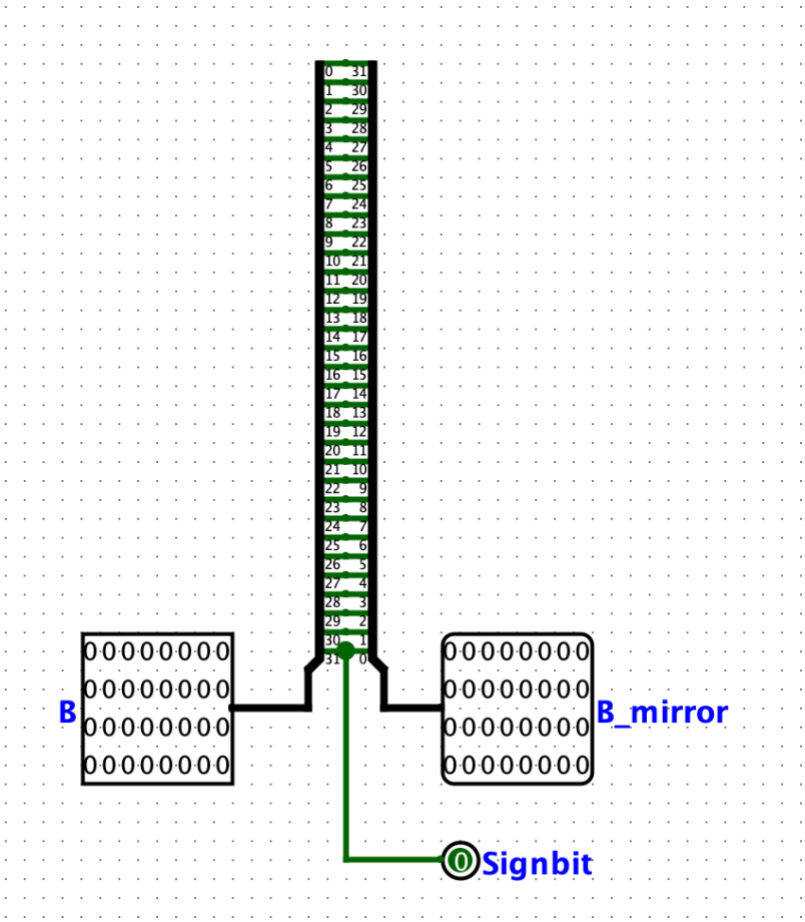


Figure 2: Mirror Subcircuit

- To generate the Shift Right without inventing other subcircuit and then adding to the ShiftCircuit, we apply 2 Mirror Subcircuits into the Shift Left Logical. The reason why I choose to use 2 Mirror as proven by the base case. For example, I want to Shift Right of the number 0011010 and the result would be 0001101. To perform in one ShiftCircuit, we mirror the number 0011010 the first time => 0101100. Then, shift left => 1011000. Finally, shift right one more time is that 0001101 (same as my expectation).
- When visualizing how my shifting component works, I decide to base on the operation in order to establish the logic of the ShiftCircuit:
    - If the Op is not the type of xx0x and the Op is not 00xx or 11xx, the left MUX would be disenabled. Hence, there is no input would be received in the MUX. Moreover, I also disable the Sa input as there is no point to let Sa gone to the LeftShift32 component to execute.
    - If the Op is 1xxx (Shift Right), it is used to distinguish the 0xxx (Shift Left)
    - If the Op is 000x, there is a signal from the 1st number to the left MUX and then that MUX is available. On the other hand, the 00xx is the Shift Left signal, thus, I send 0 to the MUX and then it chooses the B (not a B_mirror input). Furthermore, its Shift Left signal enable the right MUX since it sends 0 to it. Hence, the right MUX only receives the value of LeftShift32.

- If the Op is 1100 and 1101, its path is different from the Shift Left Logic as both the left and right MUX, they receive 1 into and then choose the B_mirror to generate the value. However, the difference is its sign-bit as Shift Right Arithmetic (1101) is concerned on the sign-bit rather than Shift Right Logic (1100). Hence, if there is an arithmetic concern of the signed number (sign-bit = 1), the 0th number in the Op would be "1" and then it goes to the upper smaller AND Gate so that it generates 1 if it is a negative number. To assert it is the Shift Right operation, it goes through the Biggest AND and gives out the value into the Cin of the LeftShift32.

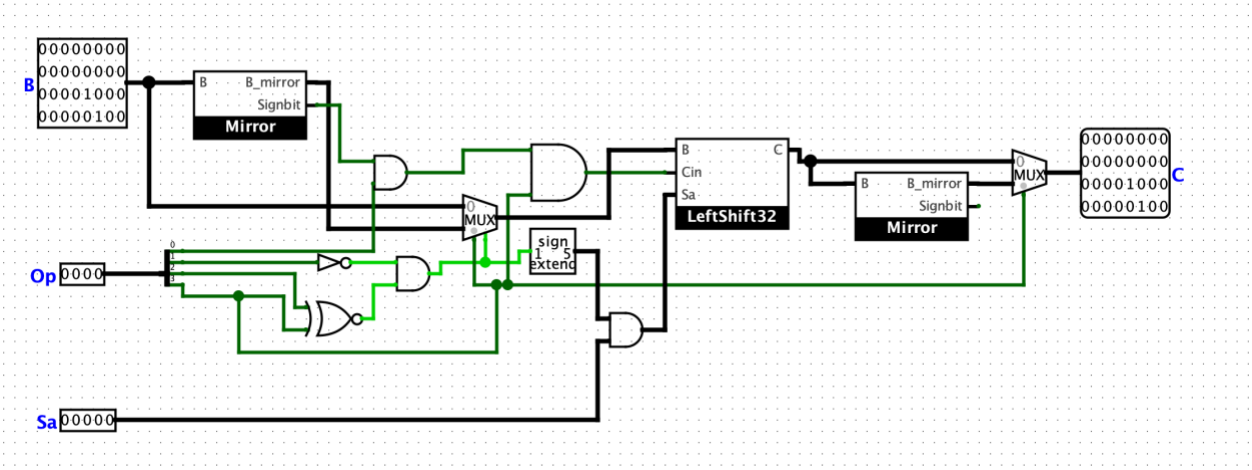


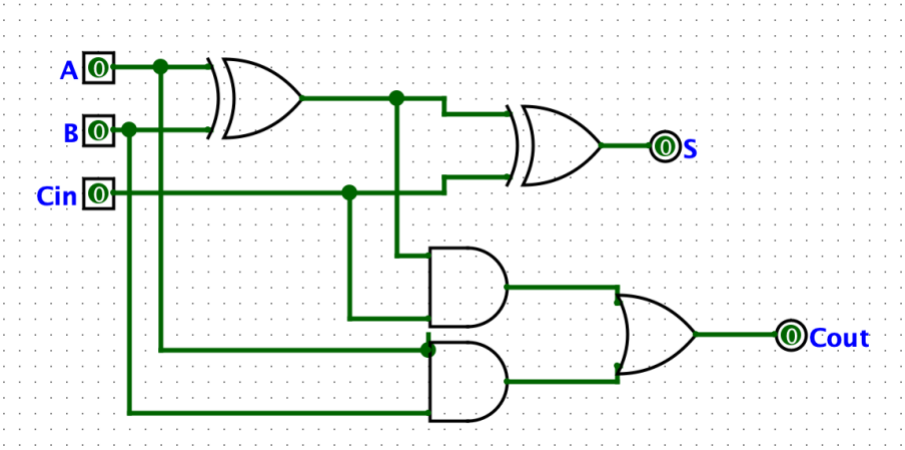
Figure 3: The Shift Circuit 32-bit designed for ALU component

- Truth table for the ShiftCircuit:

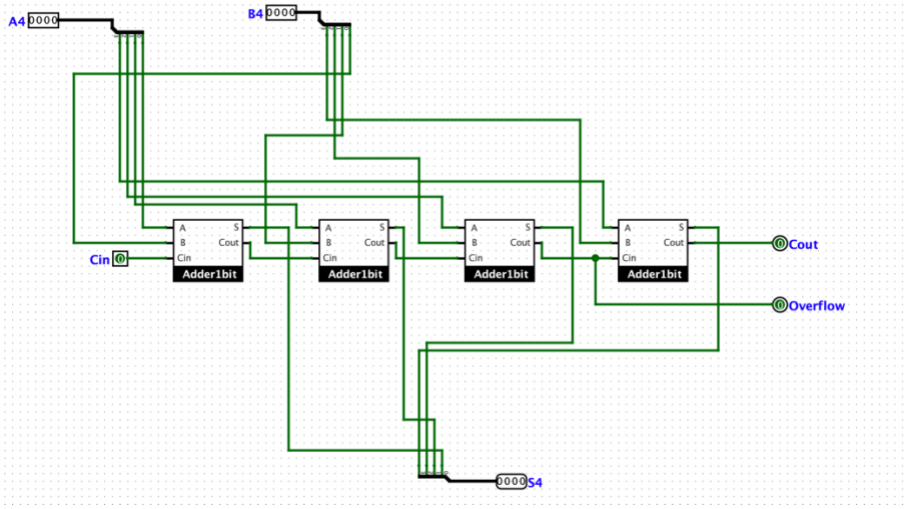
Op	Left MUX	Right MUX	B	Cin	Sa
000x	0	0	B	0	Yes
1100	1	1	B_mirror	0	Yes
1101	1	1	B_mirror	0 or 1	Yes
Others	Disabled	0 or 1	0	0	No

2.2.Add32

- Objective: Adding with 32-bit numbers and they are both two's complement numbers.
- Reuse the Add4bit from the Wet Lab by combining 4 full-bit adder to create the 4-bit adder.
  - Full-bit adder

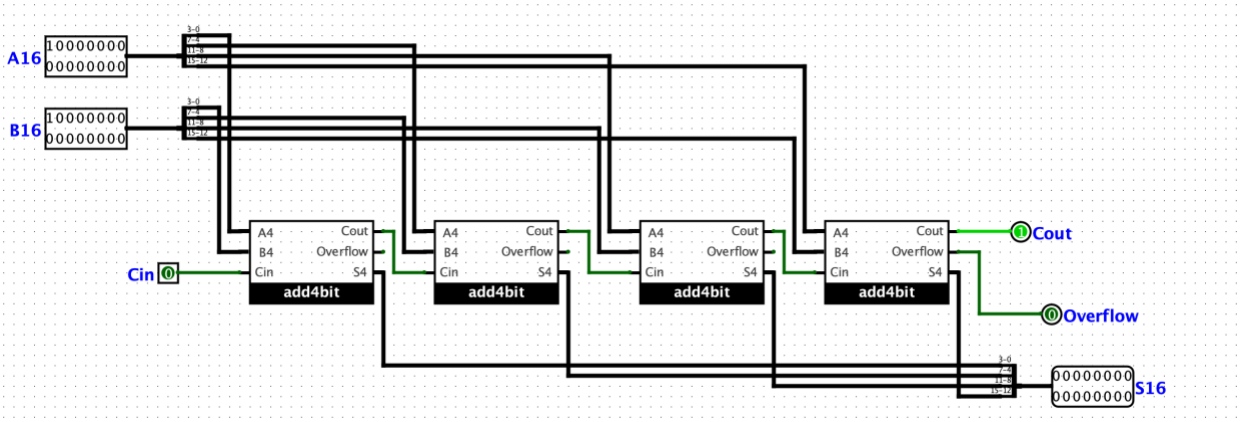


- 4-bit adder



  - Add 4 full-bit adder into to get 4-bit adder
  - Overflow detects by mapping the previous Cin of the msb (most significant bit)

- Create a 16-bit adder



  - Same procedure as 4-bit adder
  - Overflow also detects by notifying the overflow of the msb (the 4th add4bit block)

- Create a 32-bit adder (Add32)

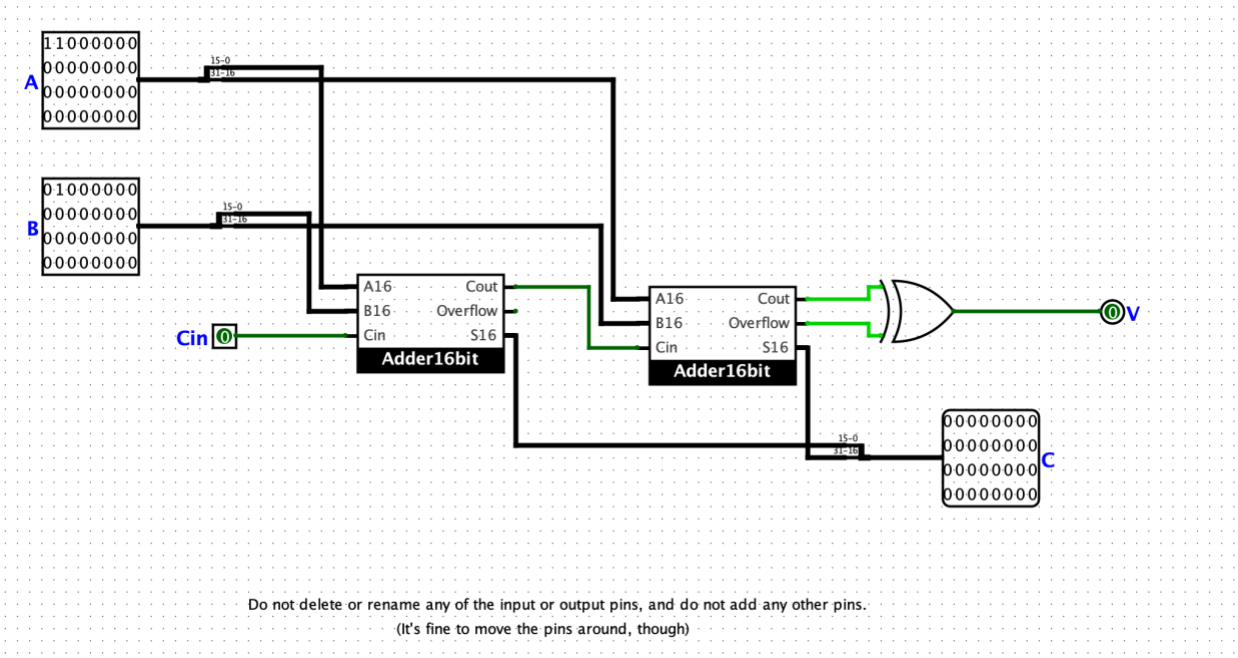
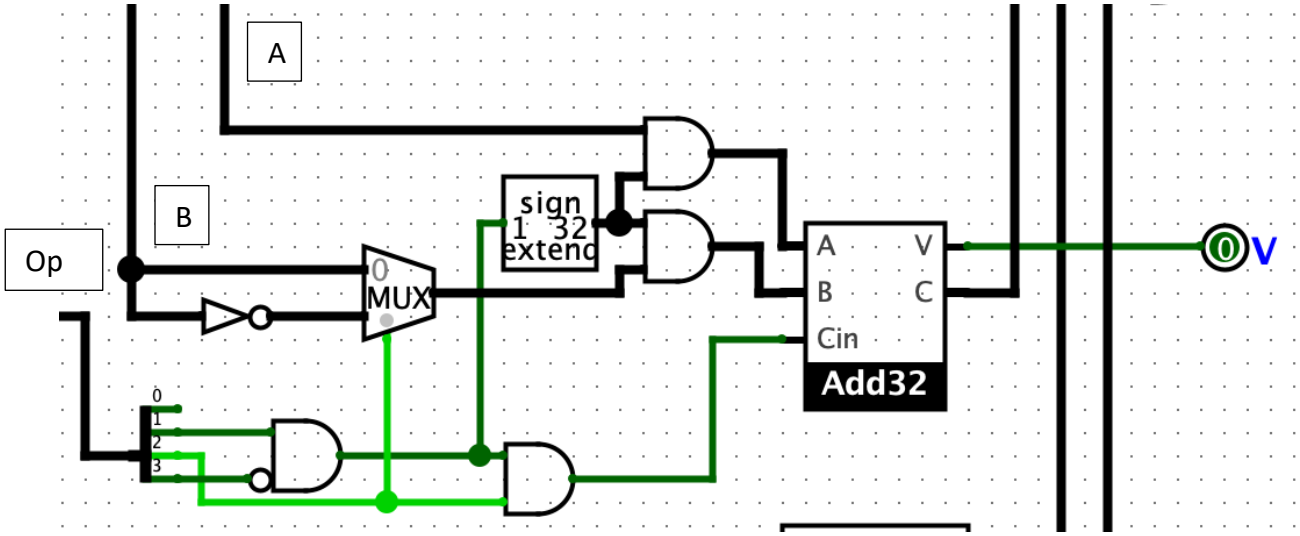


Figure 4: 32-bit Adder

- Same approach as 16-bit adder but in terms of the 32-bit adder, we combine both two 16-bit adders.
- Therefore, the detection of an overflow is much easier as we just consider the V (overflow) of the 2nd 16-bit adder block, which consists of the msb of the adding operation.

- From the ALU32, I decide to perform the Op of the Add32, which consist addition and subtraction



- From the Operation, we design the internal circuit based in the logic of addition (001x) and subtraction (011x).
  - If the Op is 0x1x, it allows A or B/NOT B to go into the Add32
  - If the Op is x0xx, it implies the addition
  - If the Op is x1xx, it implies the subtraction.

- If the Op is 0x1x and x1xx, going through the AND gate below is to ensure the subtraction and it gives Cin = 1. Simultaneously, it infers to the MUX and get the value of !B (NOT B).

- Truth table

Op	A	B	Cin
001x	A	B	0
011x	A	!B	1
Otherwise	Disable	Disable	0

2.3.LogicGate32

- Objective: We must combine the logic gate AND (0100), OR (0101), XOR (1010), NOR (1011)
- To determine the organization of the LogicGate32, I analyze the operation to set up the wires.
  - If the Op is not 01xx or 10xx and the Op is not 0x0x and 1x1x, the MUX will be disabled. Otherwise, it can be enabled and choose one of the logic gates in order below.
  - From the 0^{th} and 1^{st} bit of the Op, I differentiate the order of the logic gate to input for the MUX. Hence, the table below show the order of it

Op	Logic Gate	The order
xx00	AND	1
xx01	OR	2
xx10	XOR	3
xx11	NOR	4

  - By this order, MUX can recognize the Logic Gate to choose by the different number of the Op

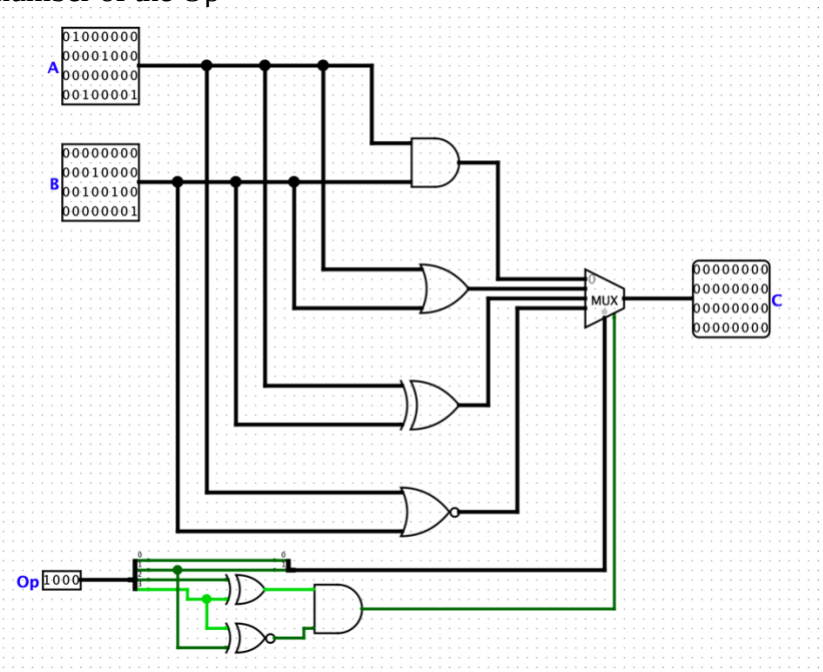


Figure 5: LogicGate32

2.4.Compare32

- To perform the comparison in the Compare32, I have to perform ne (1000), eq (1001), le (1110), gt(1111).
    - With the upper half of the diagram, it performs ne (not equal) and eq (equal) operation.
        - Since the output C is only 1 digit performed (0 or 1) under the 32-bit output, I have to scale down each 32-bit by splitter of 8 data bit. Then, one more time scaling down it into 2 data bit and finally down it into 1 bit. Later, extend the number under 32-bit output.
        - To compare A & B, we compare each element using XNOR gate (if the same element, gives out 1)
    - With the second half of the diagram, it performs le (less & equal to 0) and gt (greater than 0).
        - The procedure scaling down the 32 data-bit is the same as the previous one.
        - Before then, at the upper wire, it checks A is positive or not by checking the 31^{st} number of the A.
        - At the lower wire, it tests the value lower than or greater than 0. After going through the procedure scaling down, if A is positive and at least 1 bit from 0^{th} to 30^{th} is 1. Then, it should be greater than 0. Otherwise, it is less or equal to 0.
- For connecting the wires logically, we defined using the operation to evaluate:
    - If the Op is not 1xxx and for each operation does not contain x00x and x11x, the MUX is disabled.
    - From the 0^{th} and 1^{st} bit of the Op, I differentiate the order of the comparison gate to input for the MUX. Hence, the table below show the order of it

Op	Logic Gate	The order
xx00	ne	1
xx01	eq	2
xx10	le	3
xx11	gt	4

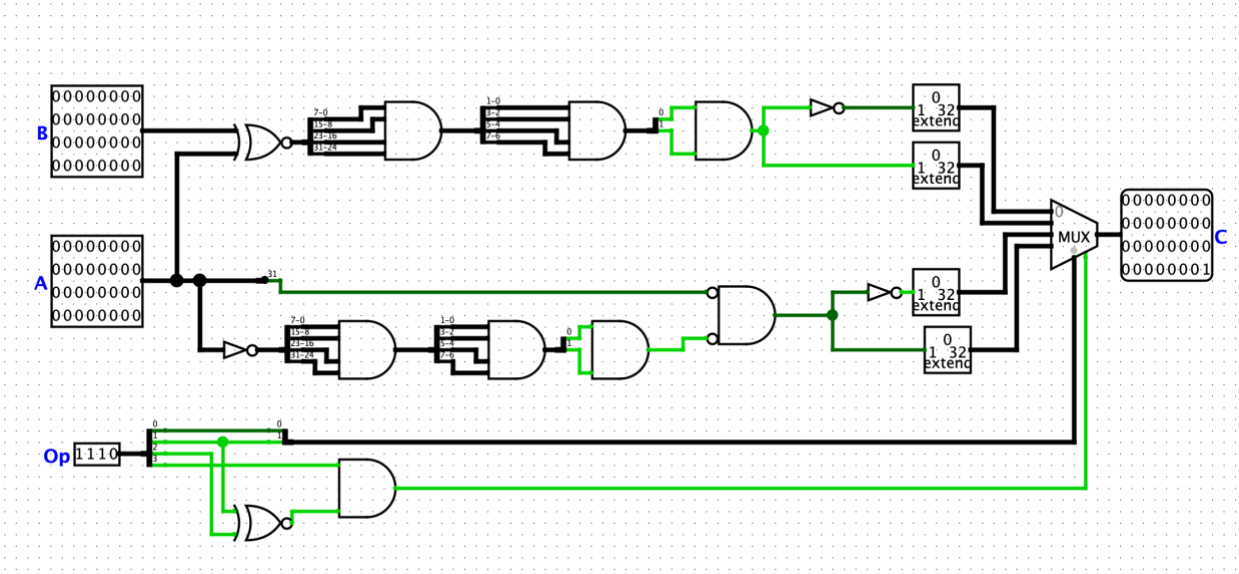


Figure 6: Compare32

C. ALU32 by combining all four parts

From the Implementation Details of Section B, we combine all four key components (LogicGate32, Add32, Compare32, ShiftCircuit). Then, connecting the inputs (A,B,Sa) and Op into the input towards four components.

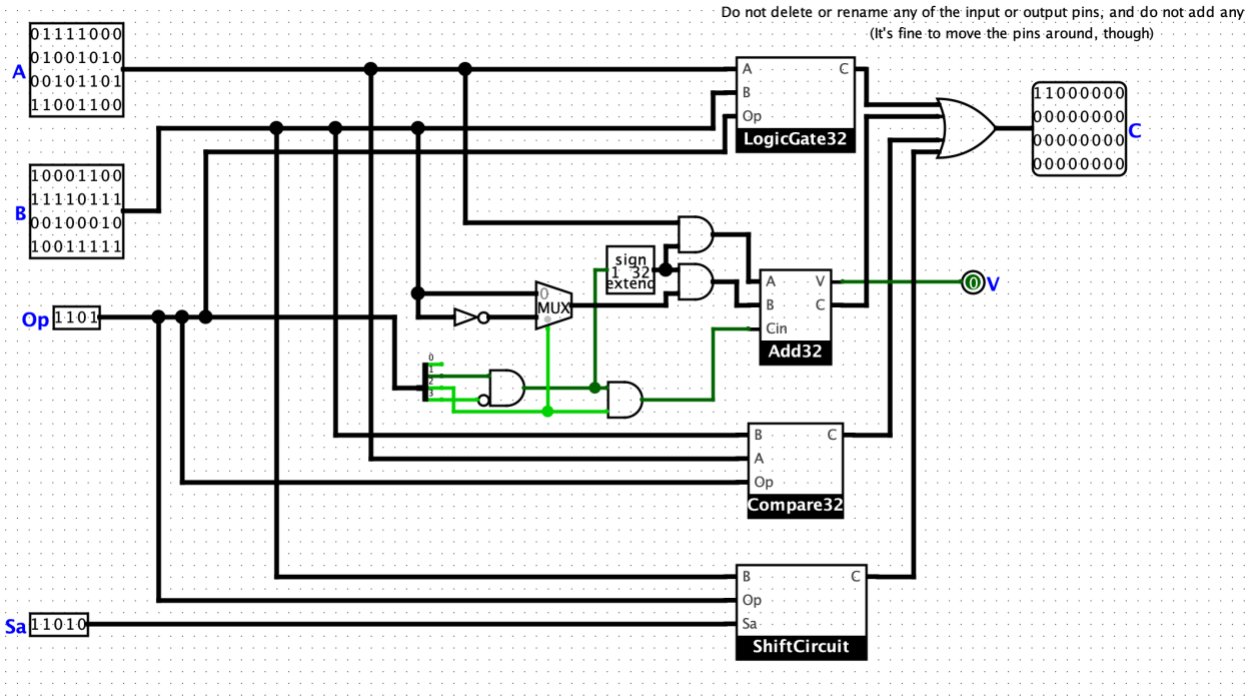


Figure 7: ALU32

3. Design Justification

- The decision led me to make the subcircuit and add to the ALU32 is that I saw the logic of the table and the grouping technique as a block is the essential strategy to build a concise and abstract component. (Professor Nam's lecture related to the build and box)

- From my perspective, the project is combined many techniques and skillset to analyze the circuit and give out the best solution to block and build in an effective way. The outcome for this project satisfies all of the requirements in the lecture since we can understand thoroughly how the circuit works and apply our logical thinking to solve the problem.

4. Reference

[1] P1: ALU of Canvas Page
[2] Wet Lab: 4-bit Adder and LeftShift32