P1: ALU

Jack Ryan (jpr254)

February 13, 2024

1 Overview

This arithmetic and logic unit (ALU) is designed to perform many of the main computations required by a computer. This ALU takes in various inputs, including two 32-bit data inputs and a 4 bit operation code to tell the ALU which computations to perform. This results in a single 32-bit data output as well and an overflow indicator for addition/subtraction. The ALU can compute many operations, including addition, bit shifting, and bitwise comparison.

2 Component Design Documentation

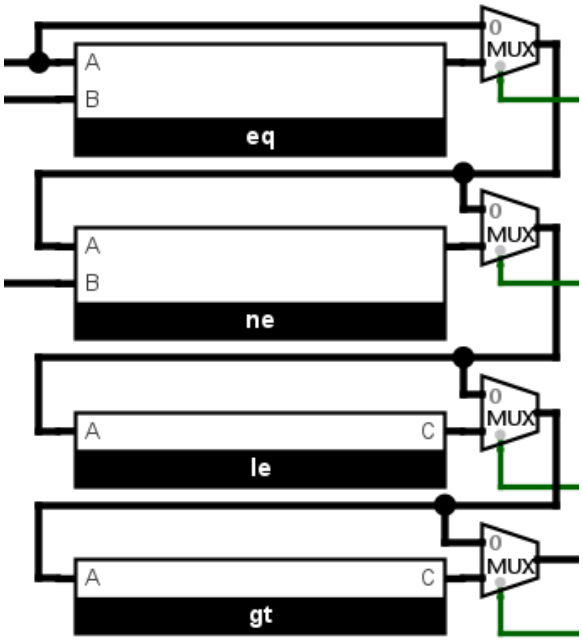
My ALU has 4 primary subcircuits.

- Add32 - handles addition of two 32-bit two's complement integers.
- GateControl - computes bitwise comparisons (AND, XOR, OR, NOR).
- Shift32 - handles logical right and left bit shifting, as well as right arithmetic shifts.
- ComparisionControl - in charge of the following comparisons (==, ! =, <=, >).

2.1 Operation codes

The ALU works off of the following list of operation codes. All op codes not on this have undefined functionality

Op	name	C	meaning	V
0010	and	C = A & B	*C gets the result of a bitwise **and** operation on inputs A and B*	V = 0
0011	or	C = A | B	*C gets the result of a bitwise **or** operation on inputs A and B*	V = 0
101x	shift left logical	C = B << Sa	*C gets the result of a shift left logical operation on input B*	V = 0
0000	xor	C = A ^ B	*C gets the result of a bitwise **xor** operation on inputs A and B*	V = 0
0100	nor	C = ~(A | B)	*C gets the result of a bitwise **nor** operation on inputs A and B*	V = 0
0001	shift right logical	C = B >>> Sa	*C gets the result of a shift right logical operation on input B*	V = 0
1100	shift right arithmetic	C = B >> Sa	*C gets the result of a shift right arithmetic operation on input B*	V = 0
0110	ne	C = (A != B) ? 000...0001 : 000...0000	*if A != B, C gets the value 000...0001, else 000...0000*	V = 0
0111	eq	C = (A == B) ? 000...0001 : 000...0000	*if A = B, C gets the value 000...0001, else 000...0000*	V = 0
0101	le	C = (A ≤ 0) ? 000...0001 : 000...0000	*if $A \leq 0$, C gets the value 000...0001, else 000...0000*	V = 0
1101	gt	C = (A > 0) ? 000...0001 : 000...0000	*if $A > 0$, C gets the value 000...0001, else 000...0000*	V = 0
100x	subtract	C = A - B	*C gets the result of A - B*	V = overflow
111x	add	C = A + B	*C gets the result of A + B*	V = overflow

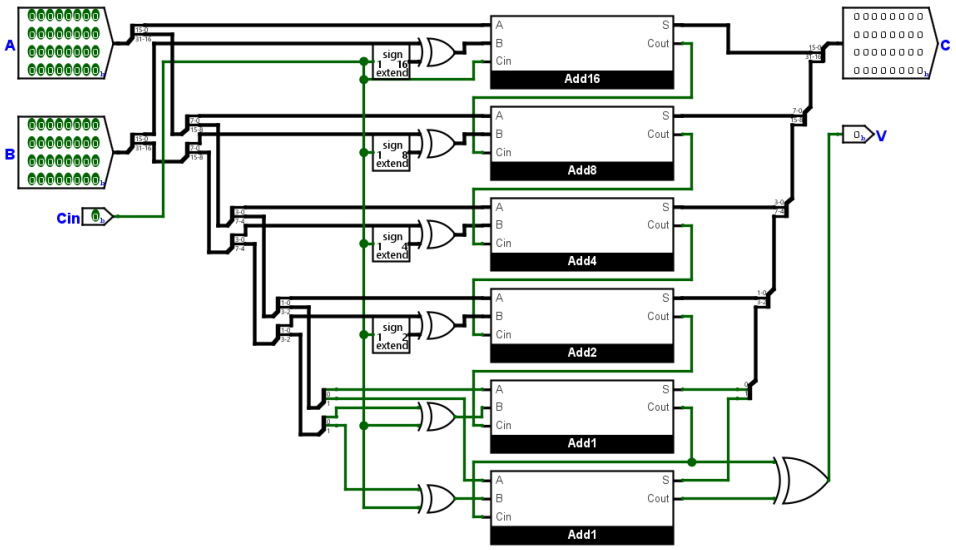


2.2 Design justification

Subcircuits were created to minimize the complexity (especially with wiring) of any given circuit. In an effort to reduce wiring complexity and avoid using multiplexers with many inputs, I made the decision to use a mux after each computation (example below) and then select between the altered and unaltered input based on the op code. The potential downside to this is a longer critical path. That being said I did notice that some operations, notably add/sub, took longer than others, which indicated to me that each computation was being run fully every time and that this downside is negligible.

3 Add32

A 32-bit adder that takes two two's complement inputs and outputs their sum and a indication of overflow.



Add32

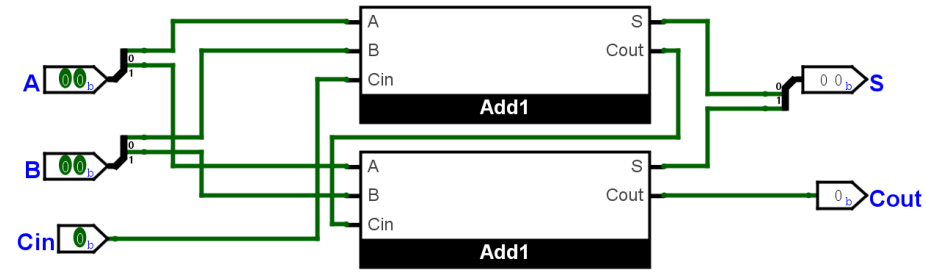
3.1 Implementation Details

In the circuit above A and B are the 32-bit inputs, while C_{in} is the carry-in bit. C is the output bit, while V is the overflow indicator. This circuit uses multiple subcircuits to complete the addition. Add16, Add8, Add4, and Add2 are all almost identical. The two one-bit adders at the end are used to determine if overflow has occurred. Together Add16, Add8, Add4, Add2, Add1, and Add1 combine to add up add the two 32-bit inputs into one 32-bit output. If C_{in} is one the values of B are negated using a bit extender and XOR gate and $C_{in} = 1$ is passed into the subcircuits. This has the effect of negating the value of B as a two's complement integer, which also means we can use this circuit to compute subtraction.

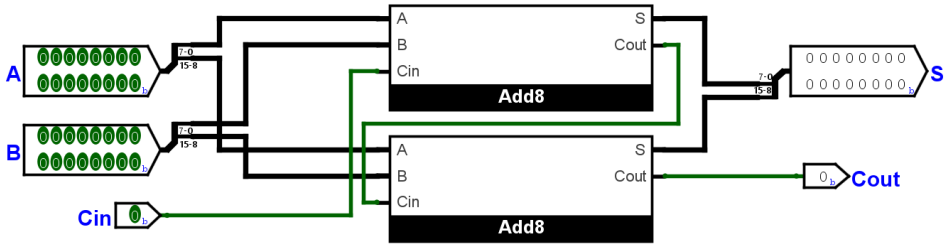
3.2 Add16 - Add2

Each of these circuits follows a the same pattern. Add2 (shown below) takes in two two-bit inputs along with a carry-in bit, includes two one-bit adders, then has a two-bit output and a carry-out bit. Add4 takes in two four-bit inputs along with a carry-in bit, includes two two-bit address, then has a four-bit output and a carry-out bit. This recursive pattern continues, finishing with Add16 which is also shown below (notice it is almost identical

to Add2).



Add2

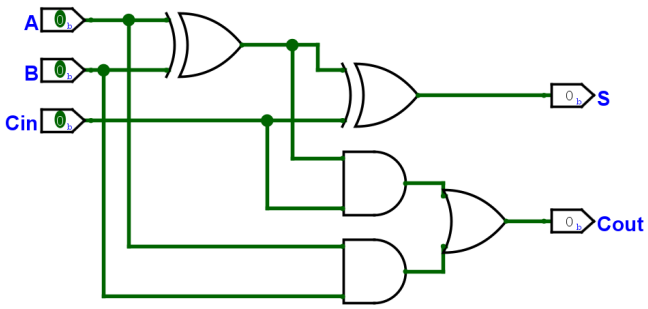


Add16

3.3 Add1

Add1 is a one-bit full adder designed to take in two one-bit inputs with a carry-in input, return the one-bit sum along with a carry-out bit. The Adder is based off of the following truth table, for which the circuit is optimal.

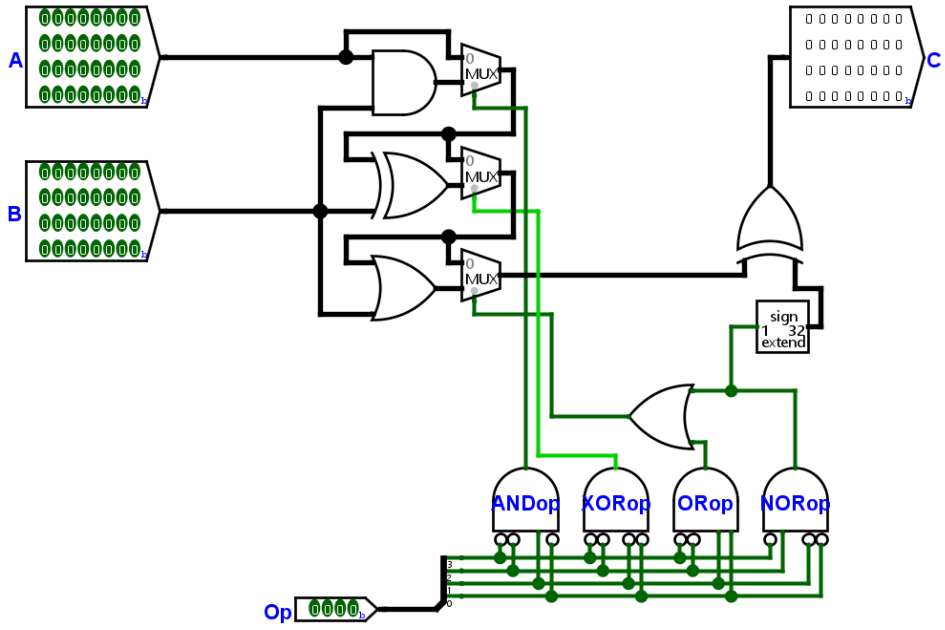
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Add1

4 GateControl

This circuit handles the AND, XOR, OR, and NOR bitwise logical operations.



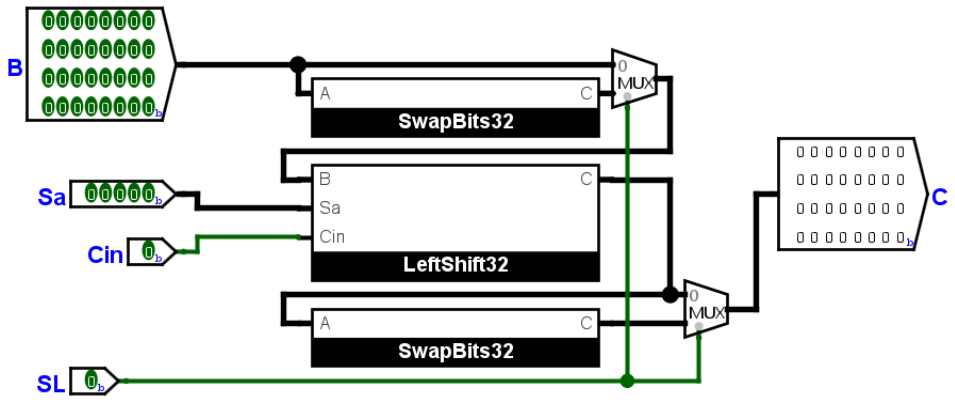
GateControl

4.1 Implementation Details

The above circuit takes in two 32-bit data inputs (A and B) along with Op, a four-bit code that determines which if any of the logical operations to compute. Input A is routed in series through the 32-bit AND, XOR, and OR gates. If Op corresponds with one of the correct op codes listed in the able in section 2.1, it will apply that bitwise logical operation. Otherwise it will simply pass the value of A through. This is done using a mux after each of the gates. NOR is handled but activating the mux for OR and then negating the resulting value though the use of a bit extender and 32-bit XOR gate. The 32-bit result C represents the computation of one or none of these bitwise logical operations.

5 Shift32

A bit shifter that can compute logical left shifts, logical right shifts, and arithmetic right shifts between 0 and 31 bits.



Shift32

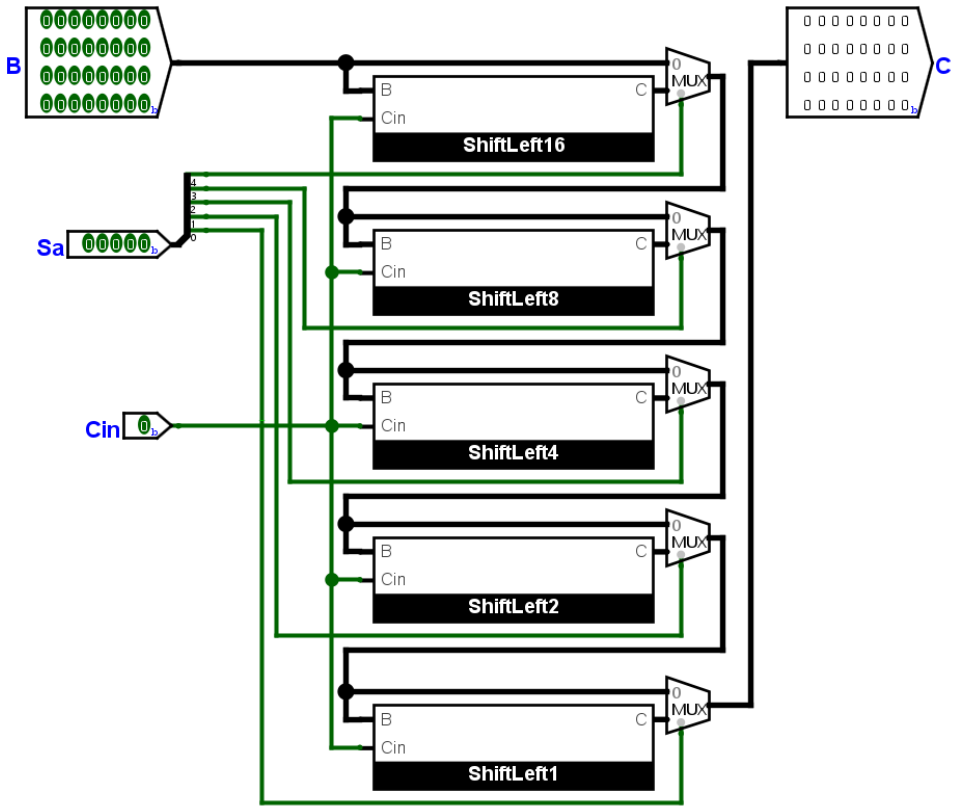
5.1 Implementation Details

This circuit takes in one 32-bit data input B, Sa the amount to shift in range 0-31, C_{in} the bit (1 or 0) to fill in, and SL to determine if the shift is left or right (0 = left, 1 = right. If SL is one I use a mux to swap the order of B. I then left shift by the specified amount, Sa, and fill in the empty bits with C_{in}. By swapping these bits again we can use LeftShift32 to do the work of a right shifter. If SL is zero the swaps do not happen, and in both

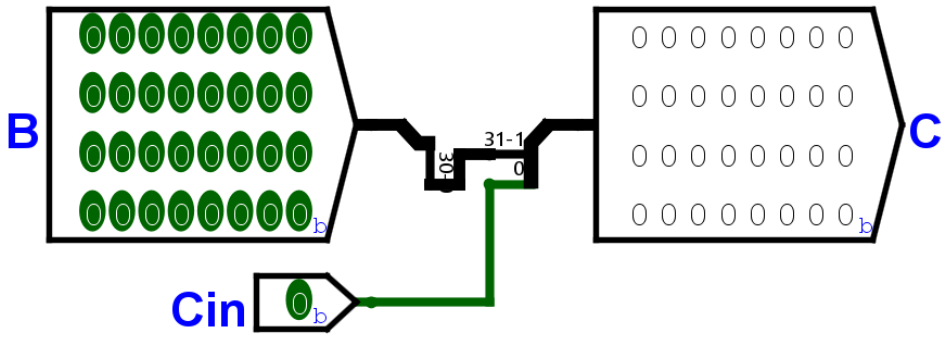
cases the shifted result is outputted in C. Sa is split into its components, each of which controls a mux to shift B the corresponding amount of bits, which together can handle and shift of 0-31 bits.

5.2 LeftShift32

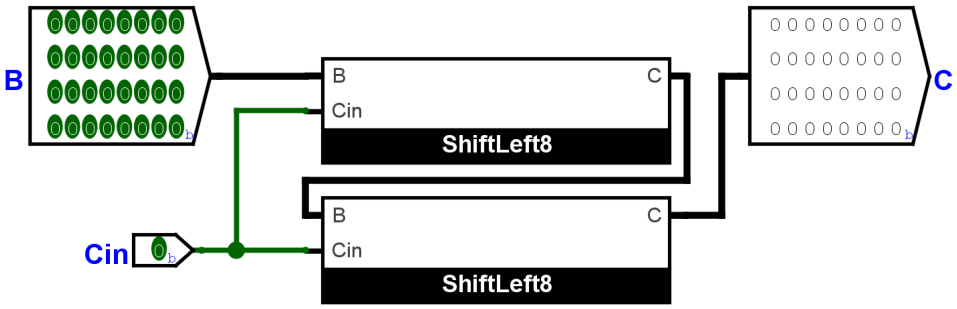
This circuit was designed in a similar way as Add32. It uses multiple recursive subcircuits, ShiftLeft16 through ShiftLeft2 (example of one below, all follow the same pattern). These circuits are built off of ShiftLeft1, which is also shown below. ShiftLeft1 works by using a splitter to section off the first 31 bits which are passed to the output C, and fills in the remaining bit with the value of C_{in}.



LeftShift32



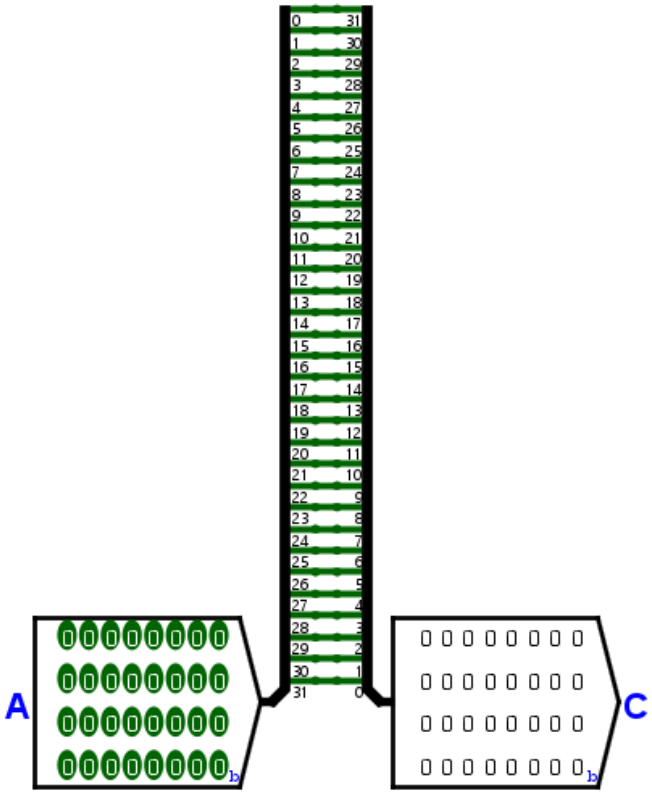
ShiftLeft16



ShiftLeft1

5.3 SwapBits32

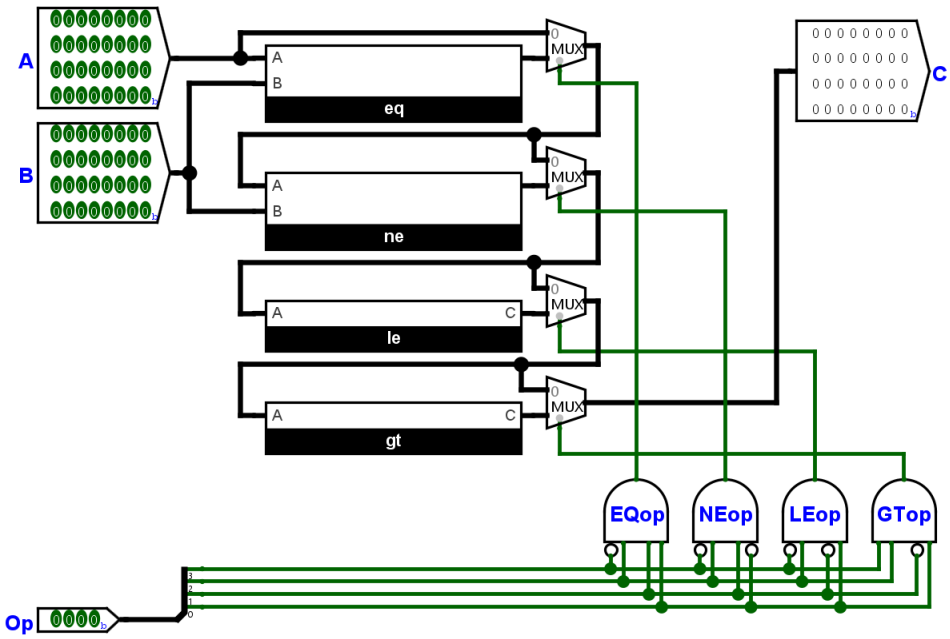
This circuit exists simply to swap the order of the 32 bits, 0 to 31, 1 to 30, and so on. It is used to reduce clutter, as the 32-bit splitters are quite large.



SwapBits32

6 ComparisonControl

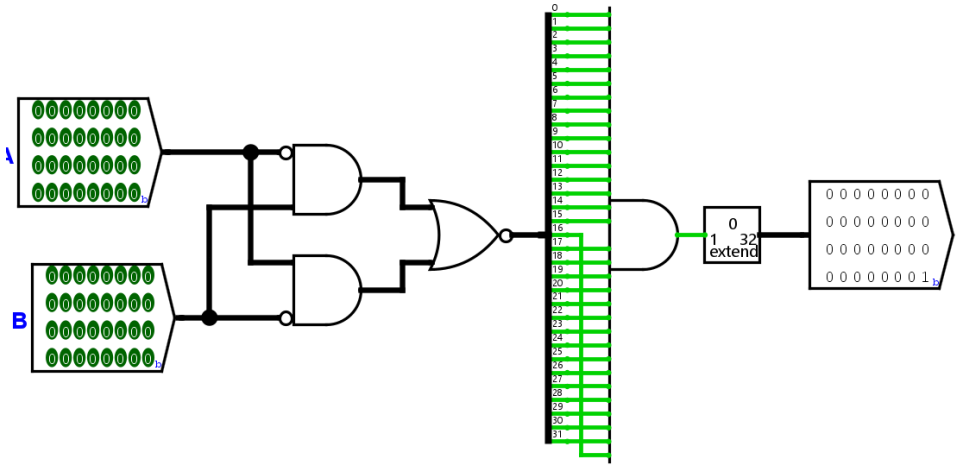
The circuit below handles the equal too, not equal too, less than or equal too, and greater than comparisons.



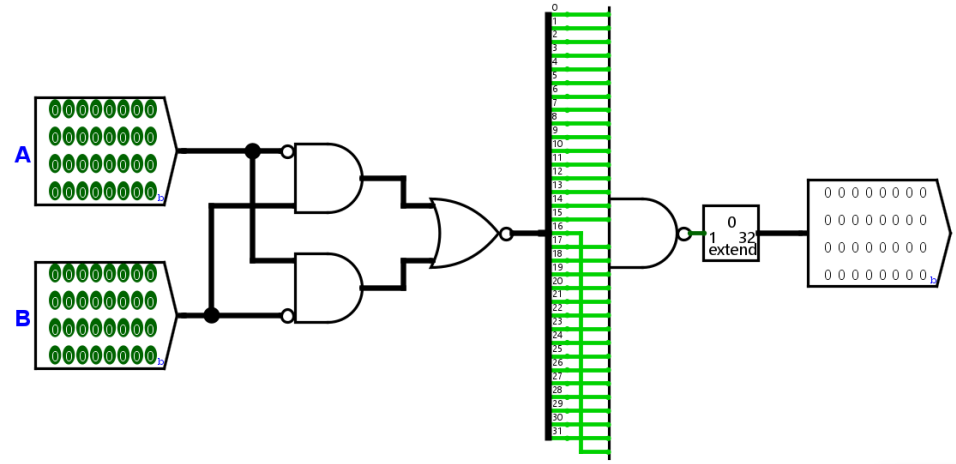
ComparisonControl

6.1 Implementation Details

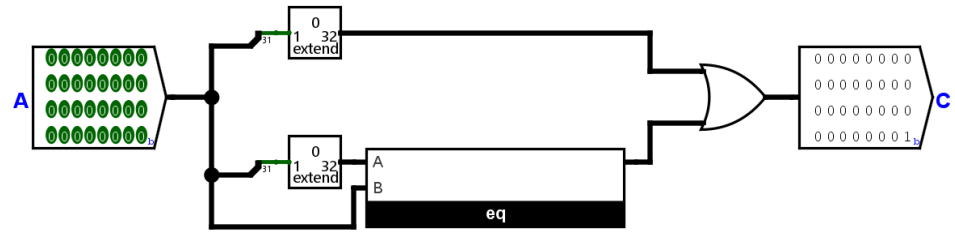
ComparisonControl takes in two 32-bit data inputs, A and B, and an op code Op. Like in previous cases, it uses multiple mux's to apply or not apply an operation. The output C is the single 32-bit result of one or none of the following computations, all of which are shown below. Note that for less than or greater too and greater than the comparison is between A and 0, not A and B like in the other cases.


A
B
0
1 32
extend

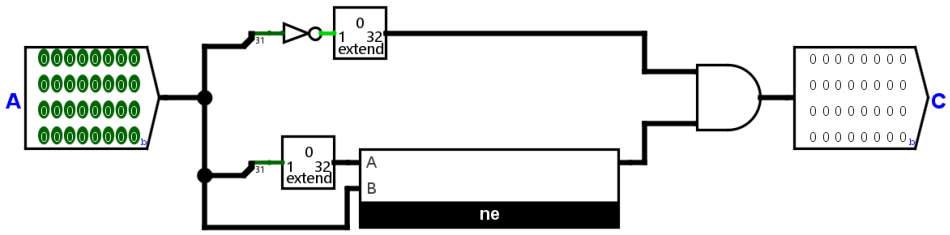
eq


A
B
0
1 32
extend

ne


A
0
1 32
extend
0
1 32
extend
A
B
eq
C

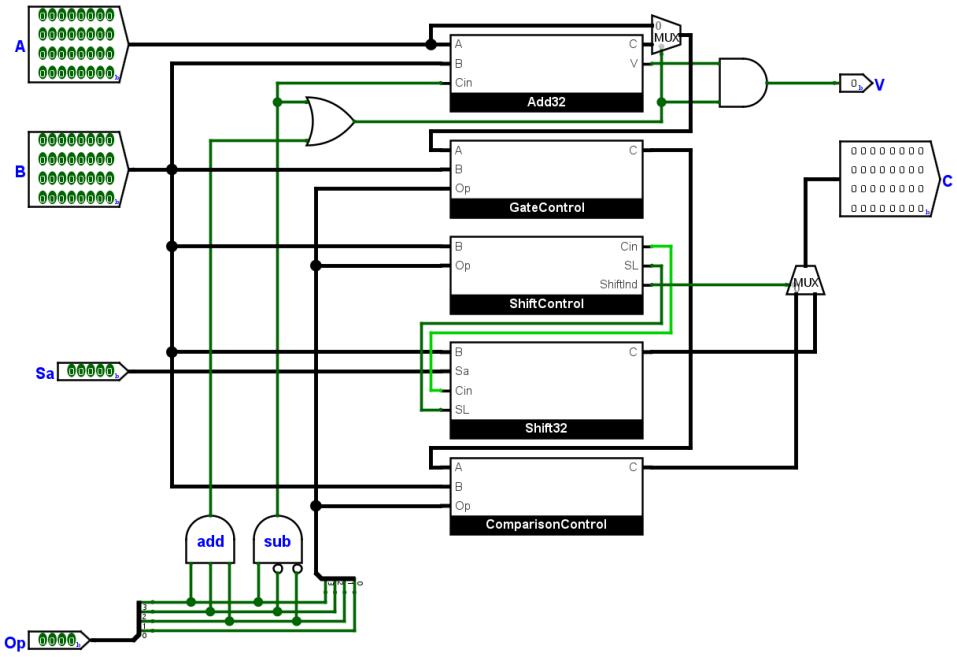
le



gt

7 ALU

Now we can finally consider the ALU. Many parts of the design will now feel familiar. The purpose of the ALU is to compute one of the operations listed in the table in section 2.1 based on a corresponding op code.



ALU

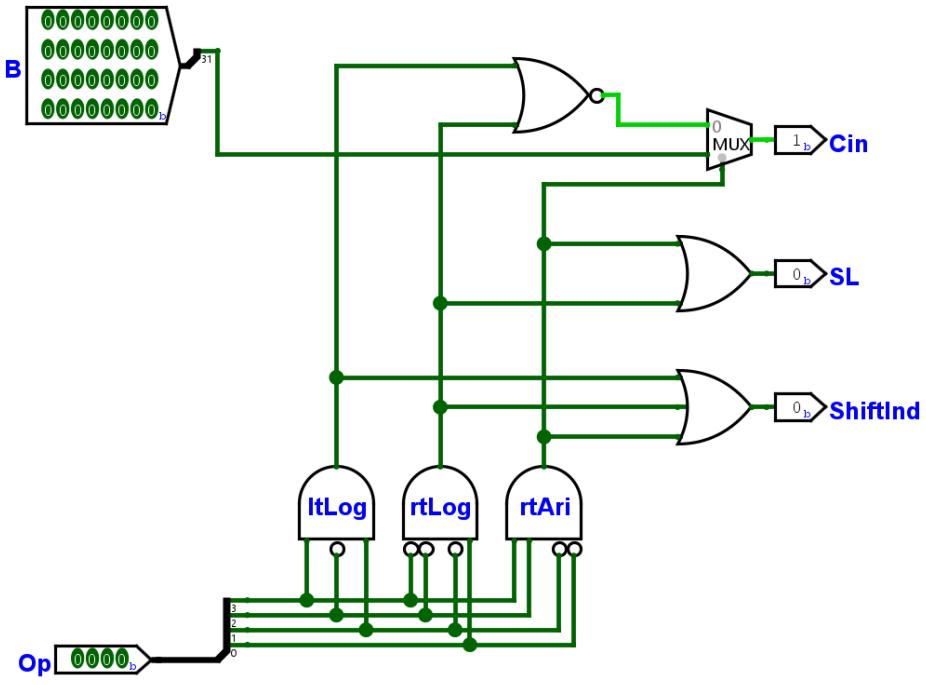
7.1 Implementation Details

ALU takes in two 32-bit data inputs, A and B, Sa an amount to shift in range 0-31, and Op a four-bit code to determine which operations to run.

Addition and subtraction trigger the same mux to tell the circuit to use the output of Add32. Subtraction is differentiated from addition by passing in 1 for C_{in} (the effect of which is discussed in section 3.1. Op is passed into GateControl and ComparisonControl to allow them to make the corresponding computations when necessary. It is also passed into ShiftControl, which is a helper circuit for Shift32 which will be explained separately below. Since shifting is applied to B instead of A, we have a mux to choose between A when 0 and B when 1 , which occurs if and only if a shift has taken place (represented with $ShiftInd = 1$). The desired computation is displayed in the output C.

7.2 ShiftControl

ShiftControl is a helper used to send the correct inputs into Shift32. If the op code for left logical shift (ltLog) is present we pass 0 as C_{in} and SL $= 0$. For right logical shift C_{in} is 0 and SL is 1. For right arithmetic shift C_{in} matches the most significant bit of B, and SL is 1. All of this behaviour is per definition of the corresponding shifts. In the case that any of the three shifts occur, ShiftInd is set to 1.



ShiftControl