COMP2020 Project 1: ALU, Design Document

Nguyen Dinh Cuong, V202000025

1 Overview

This is a 32-bit ALU with the following features: equal, not equal, and, or, nor, xor, less than or equal to, greater than, add, subtract, right shift arithmetic, and left/right shift logical. The ALU is part of a RISC V processor, which is the latter project that we have to do.

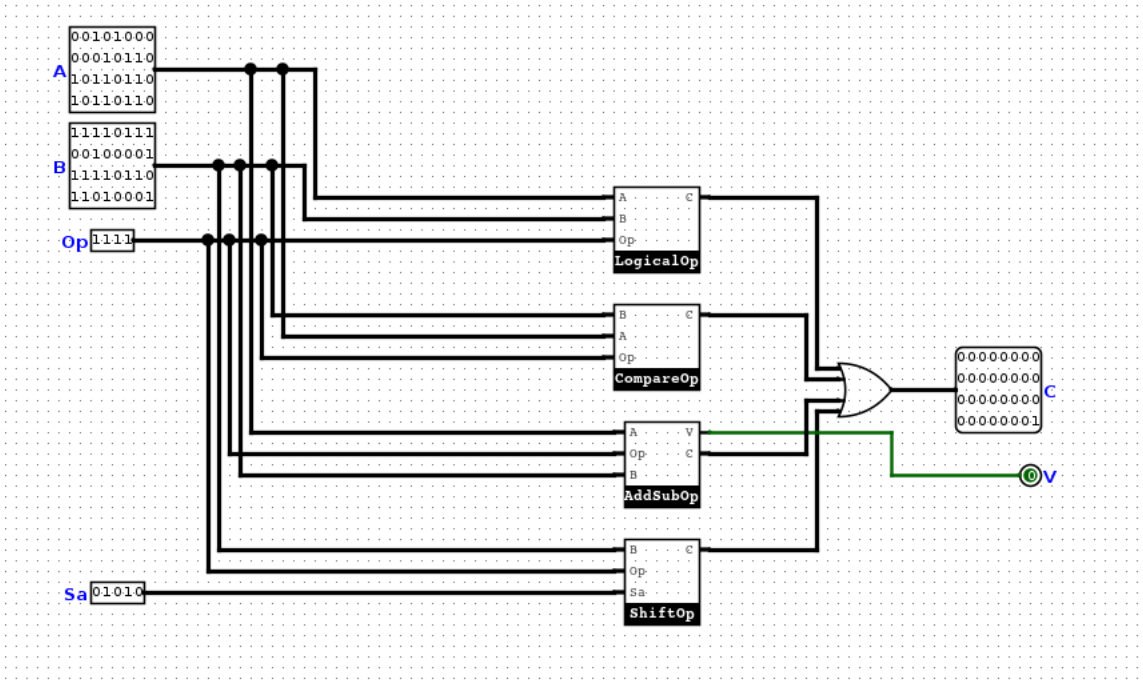

Figure 1: Final Product - The 32-bit ALU

2 Logical Operators

2.1 Implementation Details

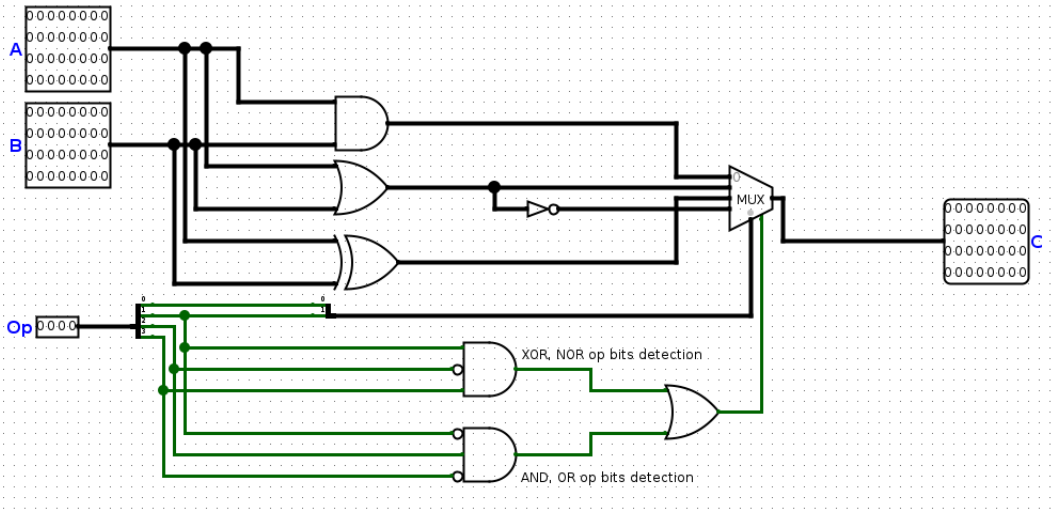

Figure 2: Logical Operators Implementation - LogicOp

AND, OR, XOR, and NOR are the four logical operators that the ALU must execute. Since NOR could be replaced using the NOT of OR, we can use three gates only. Op bits 0 and 1 represent the control for these operations. These two Op bits are combined into a single wire and fed into a 2x4 multiplexer as the input signal.

Control Op Bits Handling Truth Table - Logical Operators		
Op Bit 0	Op Bit 1	Output
0	0	A AND B
0	1	A XOR B
1	0	A OR B
1	1	A NOR B

Another crucial part of this circuit is the logical operation Op bits detection. We utilize the minimum terms and sum of products approach that we learned in Lecture 2 to create the detection circuit. According to the project description, the last three bits of logical operations must be "010" or "101", thus, we come up with the design above. Other detection circuits will be built using the same method, with the exception of the Adder/Subtractor, which will be built using a different technique.

2.2 Evaluation

The most significant design decision was to divide the wire after OR rather than implementing a separate NOR gate. While adding another NOR gate to the circuit may improve its clarity, employing a NOT gate is more efficient.

3 Value Comparisons

3.1 Implementation Details

This circuit is divided into two sections, the upper half comparing A and B. A XOR B equals a 32-bit zero if A is equal to B, thus, A XNOR B equals a 32-bit one if A equals B. Check if A XOR B equals zero using AND gates. If the answer is yes, A equals B; if the answer is no, A does not equal B. We verify that A is greater than zero in the lower section. If A is positive, the 31st bit will be zero, and at least one of the remaining bits will not be zero. A is positive if it meets this criterion; else, it's less than or equal to zero.

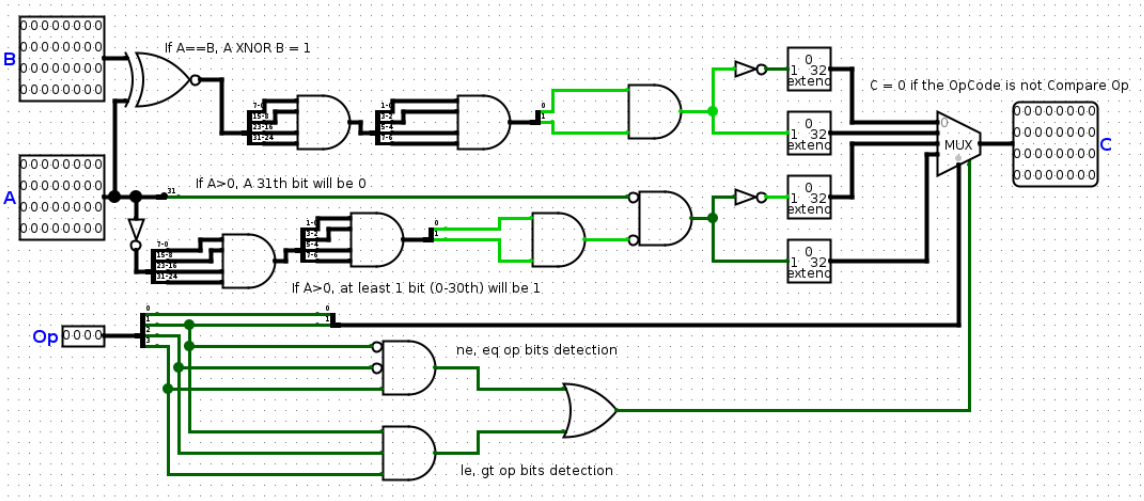

Figure 3: Comparing Operation Implementation - CompareOp

The control for these operations is likewise represented by Op bits 0 and 1. As the input signal, the two Op bits are merged into a single wire and sent through a 2x4 multiplexer. We also implement the comparison operation Op bits detection circuit using the same technique as LogicOp.

Control Op Bits Handling Truth Table - Comparing Operation		
Op Bit 0	Op Bit 1	Output
0	0	(A != B) ?
0	1	(A == B) ?
1	0	(A $\leq$ 0) ?
1	1	(A $>$ 0) ?

3.2 Evaluation

The function of this circuit gets apparent and the implementation becomes simpler by splitting it into two smaller components. Using those bit extenders for calculating and gates for Op bits handling are effectively-sufficient methods in this case.

4 Adder/Subtractor

The sum of two 1-bit inputs with a carry bit is supported by a one-bit complete adder. This circuit is required for more complex full adders to be built.

4.1 Implementation Details

4.1.1 One-Bit Adder

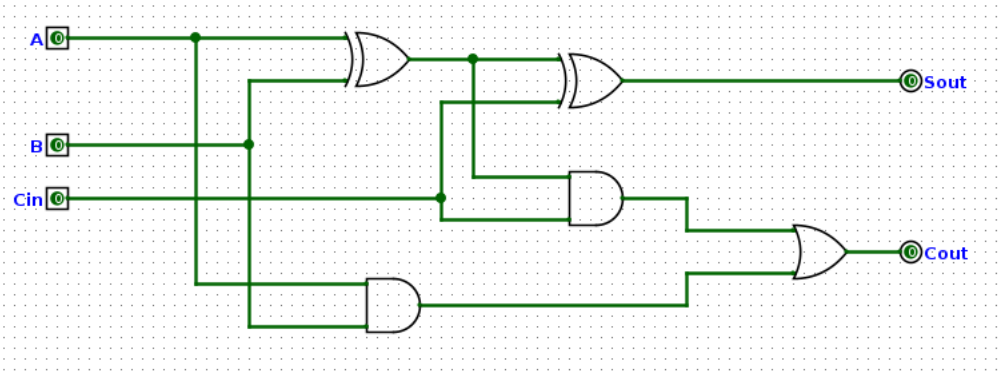

Figure 4: One-Bit Complete Adder Implementation

A and B are the 1-bit inputs in the figure above, whereas Cin is the carry-in bit. The output bit is S, and the carry-out bit is Cout. It is the most optimized circuit for the table below:

Truth Table of One-Bit Adder Circuit				
A	B	Cin	S	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

4.1.2 Higher-order Adders

All additional higher-order adders are built using 1-bit adder. As this 32-bit adder requires an output V to indicate overflow, a circuit which is Add2V was created as a 2-bit adder that outputs V instead than Cout. This process is repeated for all adders up to and including 16-bit adders.

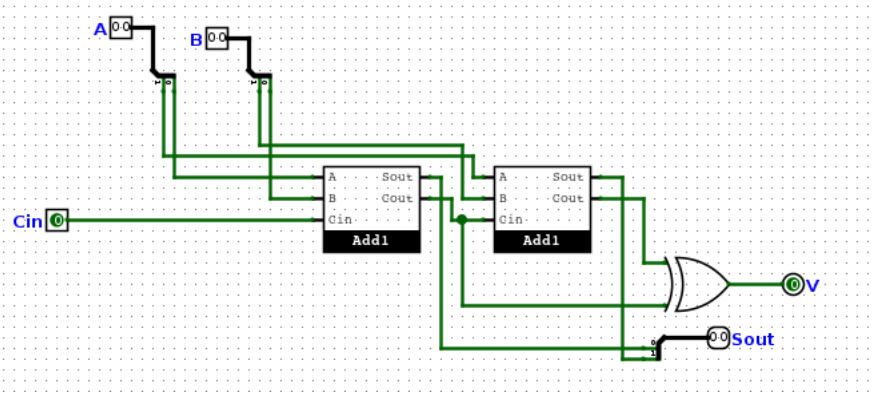

Figure 5: Two-Bit Adder Implementation With Overflow Detection - Add2V

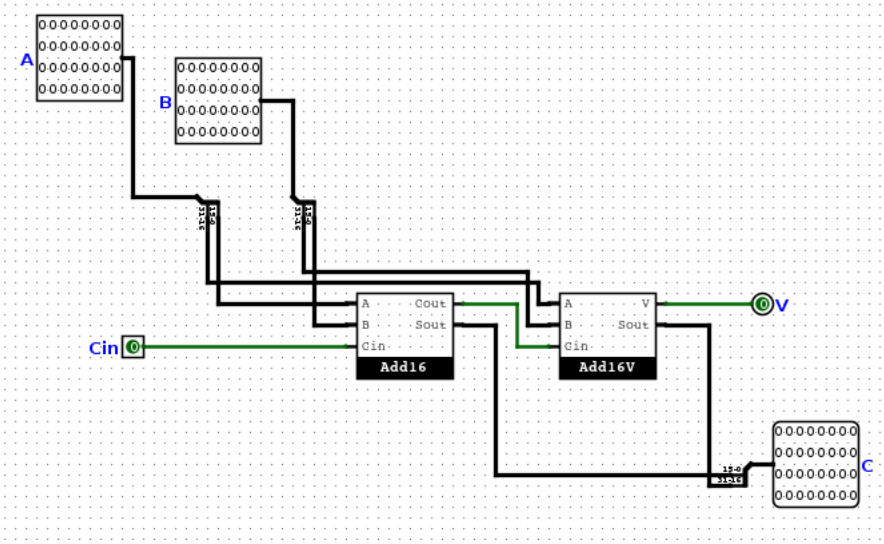

Figure 6: 32-bit Adder Implementation With Overflow Detection - Add32

4.1.3 The ALU Add/Sub Operation

The control for the Add/Sub operation comes from Bit 2 of the Op input. A-B is evaluated if Op Bit 2 is 1, and A+B if Op Bit 2 is 0. In case we need to use B is a two's complement number, the negative B is obtained by taking the inverse and adding one in subtraction. This may be done in circuit form using a NOT gate on B and a multiplexer, as well as 1 in Cin. I, on the other hand, have chosen a different approach. Instead of using a multiplexer with inverses as inputs, which would decrease the efficiency, we just need to use a bit extender and a XOR gate. They are logically equivalent since their truth tables are the same.

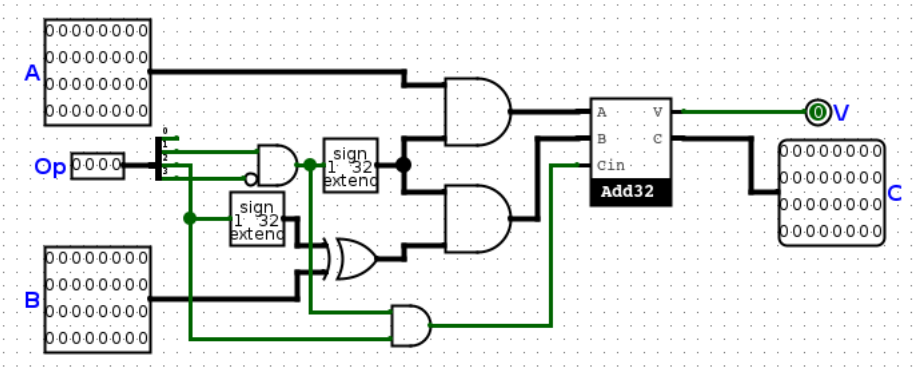

Figure 7: Adder/Subtractor Implementation - AddSubOp

Control Op Bits Handling Truth Table - Adder/Subtractor	
Op Bit 2	Output
0	A + B
1	A - B

For the sake of add/sub Op bits detection, things are also a bit different. Because the Op code is unique for this operation (011x for subtracting and 001x for adding), we don't need the use of the minimum terms and sum of products technique. All we need is an AND gate with one negated input and a bit extender.

4.2 Evaluation

For both the resultant bit and the carry-out bit, the circuit implements the simplest and most effective implementation. However,the fact that each sub-circuit for the adder required two distinct types of circuits would have been a weakness in this design. While this has no bearing on performance, a more simple architecture for circuits in general would be preferable. This circuit also employs a ripple carry adder, which is too inefficient for an actual microprocessor, as stated in the project description. The use of XOR gates and bit extenders instead of multiplexers for add/sub Op bits detection may be the most prominent feature of this design.

5 Shifter

5.1 Implementation Details

5.1.1 Mirror

The Mirror bit reverser reverses the order of 32 bits. Reversing bits, executing a left shift, and reversing bits again is identical to a right shift. With simply a left shifter, you can shift right and left. Making it a subcircuit, albeit a simple procedure, saves space.

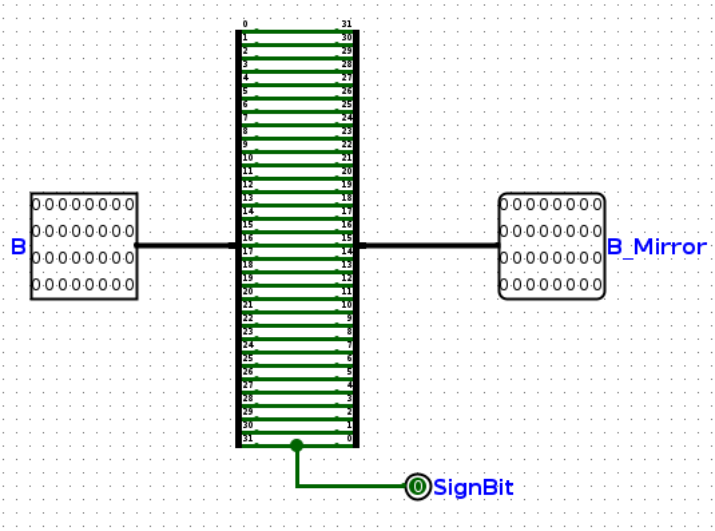

Figure 8: Mirror Bit Reverser Implementation

5.1.2 LeftShift32

The left shifter accepts a 32-bit value B, Cin, and Sa and returns C = (B << Sa) — carrybits. Sa is divided into five 1-bit components in this implementation. These 1-bit wires each indicate a shift of 16, 8, 4, 2, or 1 bit. B can shift up to 31 bits by combining these shifts.

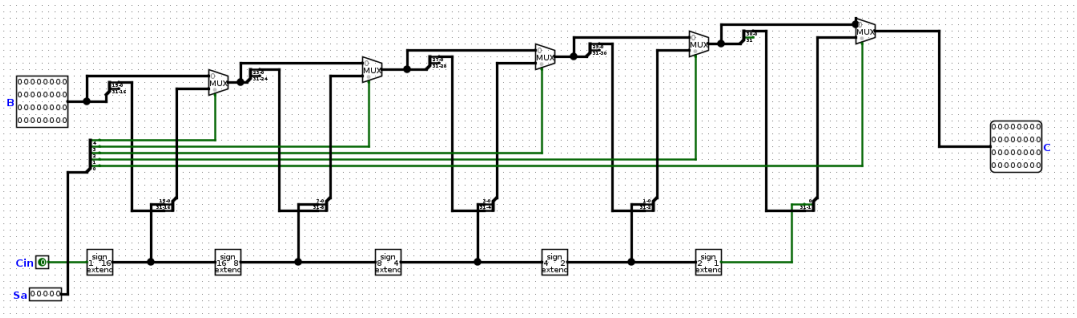

Figure 9: 32-bit Left Shifter Implementation - LeftShift32

5.1.3 The ALU Shift Operation

Two multiplexers are used to determine the logic for selecting the shift type. The left mux accepts input B as well as B's inverse. Op bit 2 and 3 both determine whether the operation is a left or right shift so I simply use Op bit 2 as the control bit. Hence, B is entered unchanged if the shift is to the left, and the inverted B is used when the operation is a right shift. After the shifter, there's a corresponding mux. If right shift was used, that right mux will re-reverse B. Not to mention, if we look back at the mirror circuit, we can see that in addition to reversing the input, it can also return the input's sign bit. We can utilize this sign bit to feed Cin into the LeftShift32 circuit when we need to execute an arithmetic right shift (to specify whether it's SRL or SRA, we must also use Op bit 0).

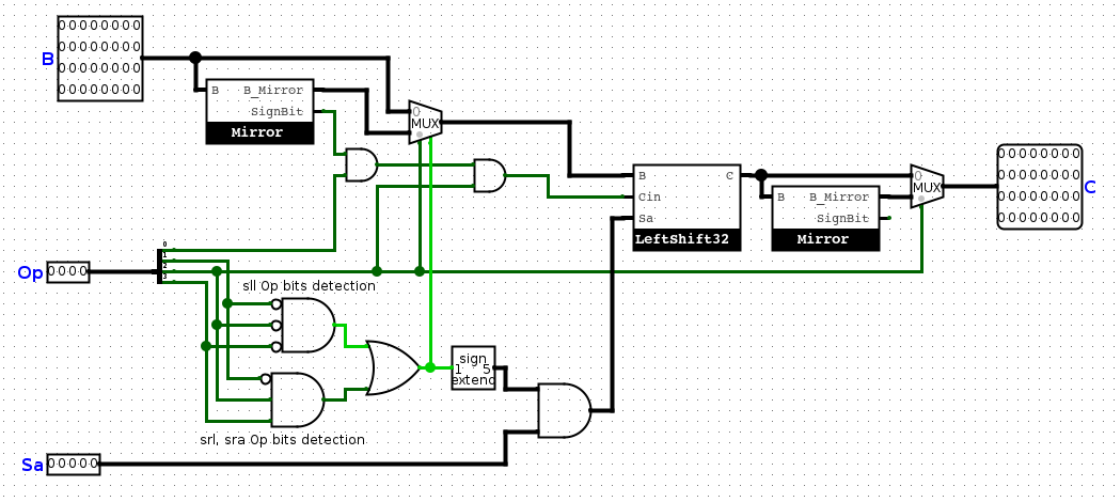

Figure 10: Shift Operators Implementation - ShiftOp

Control Op Bits Handling Truth Table - Shift Operators	
Op Bit 2	Output
0	A << B
1	A >> B

We apply the same approach for detecting shift Op bits - minimum terms and sum of products. The distinction here is that the detecting circuit's output is also extended and fed up to an AND gate alongside with the Sa. This is done to prevent LeftShift32 from executing if the Op bits do not match any provided Op bits, but the user inputs a number in the Sa.

5.2 Evaluation

Having a right and left shifter is inefficient, as we can ultilize only one left shifter for three separate tasks in this design. In addition, another remarkable feature of this circuit is the use of the bit extender and the AND gate to prevent the LeftShift32 from executing automatically.

6 Output C

Any portion of this ALU will only produce a non-zero result if and only if the operation is corresponding to this component, as shown in the figure below. As a consequence, the only non-zero result is output into C through an OR gate.

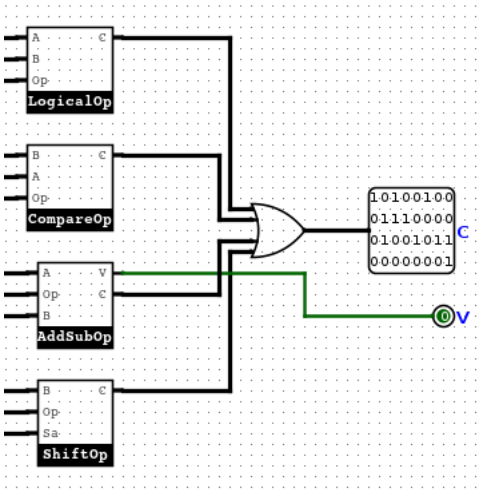

Figure 11: Gathering Four Parts