ALU Design Doc

Overview

In this project I designed and built a 32-bit ALU for the RISC-V instruction set. The two main components of the ALU are a 32-bit signed adder and a 32-bit left shifter. Since it was required that only one “copy” of each of these components be present, I then created additional subcircuits that, by transforming the inputs and outputs to these components, implemented subtraction and right arithmetic/logical shifting. I then created a comparator subcircuit to implement the four comparison operations. Finally, I combined these subcircuits along with the operators that corresponded directly to basic logic gates into a unified ALU. I used a decision tree of 1-bit MUXs that selected on one bit of the opcode at a time to select the right value for output. Finally, I also had to check whether an add or subtract was being performed to decide whether to assert the overflow output from the adder.

Components

32-bit Signed Adder

The 32-bit signed adder was constructed directly from what was presented in lecture. First, I had to create a 1-bit full adder. I did this by using the Combinatorial Analysis feature of Logism to generate the circuit from the following truth table.

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

Table 1: Truth Table for 1-bit full adder

I then created 2-bit, 4-bit, 8-bit, and 16-bit adder subcircuits using this 1-bit adder. To create the 32-bit adder, I couldn't just combine two 16-bit adders because then there would be no way to check for overflow. Instead, I combined a 16-bit, 8-bit, 4-bit, 2-bit, and two 1-bit adders to make my 32-bit adder. I captured the Cout of the two 1-bit adders and xor'ed them to check for overflow, as covered in lecture.

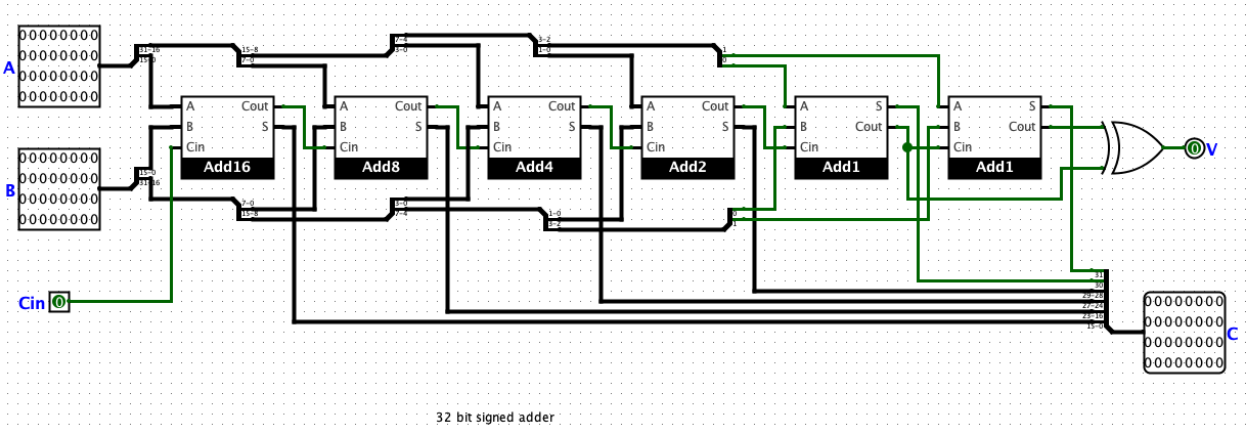

Figure 1: 32-bit Signed Adder

Combined Adder/Subtractor

Since the assignment specified that we only use one 32-bit adder subcircuit, I needed to implement subtraction by transforming the inputs to the adder. I did this by inverting B and setting Cin=1 when a subtraction was being performed. This has the effect of "flipping the bits and adding one" which can be used to negate a two's complement number. To see whether or not a subtraction or add is being performed, I added a 1-bit input that is 1 for subtraction and 0 otherwise (this directly corresponds to the 2nd bit of the opcode if an add or subtraction is being performed). This input is then sign-extended and xor'ed with B to invert B if a subtraction is taking place. The Cin is directly connected to this signal that will be 1 for a subtraction. These steps are labeled in Figure 2. The outputs of the adder do not need to be transformed.

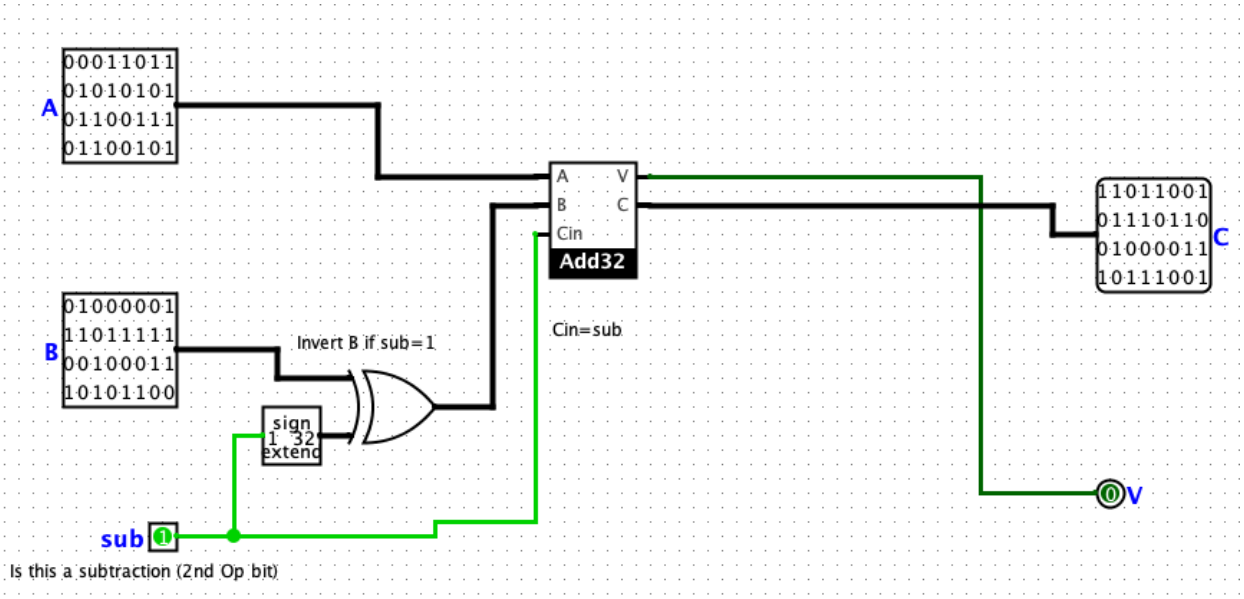

Figure 2: Adder/Subtractor Subcircuit

32-bit Left Shifter

To create a 32-bit left shifter, I started by creating a 1-bit left shifter as shown in Figure 3. This circuit uses a MUX to choose between the original signal and a version that is shifted by 1-bit and filled with the value of Cin.

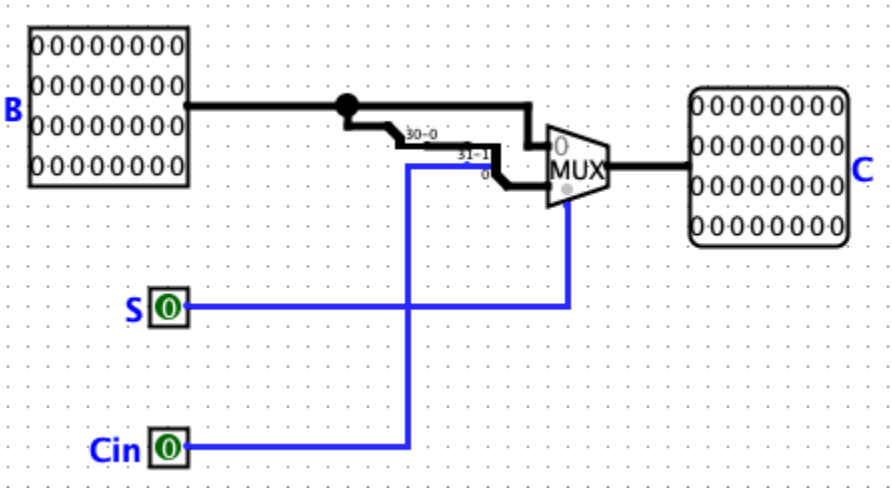

Figure 3: 1-bit Left Shifter

I used a similar process to create 2-bit, 4-bit, 8-bit, and 16-bit shifters, each of which perform a shift by the respective amount (and fill with Cin) if S=1 and do nothing otherwise. To perform a 32-bit shift by an arbitrary amount (between 0 and 31), I used each bit of the shift amount to control the appropriate subshifter. I.e. the least significant bit was used to determine if a shift by 1 should occur, then the next least significant bit was used to control a shift by 2, etc.

In this way, it is possible to shift by any amount that can be represented as a 5-bit binary number (0 through 31). The Cin is also propagated to each sub-shifter.

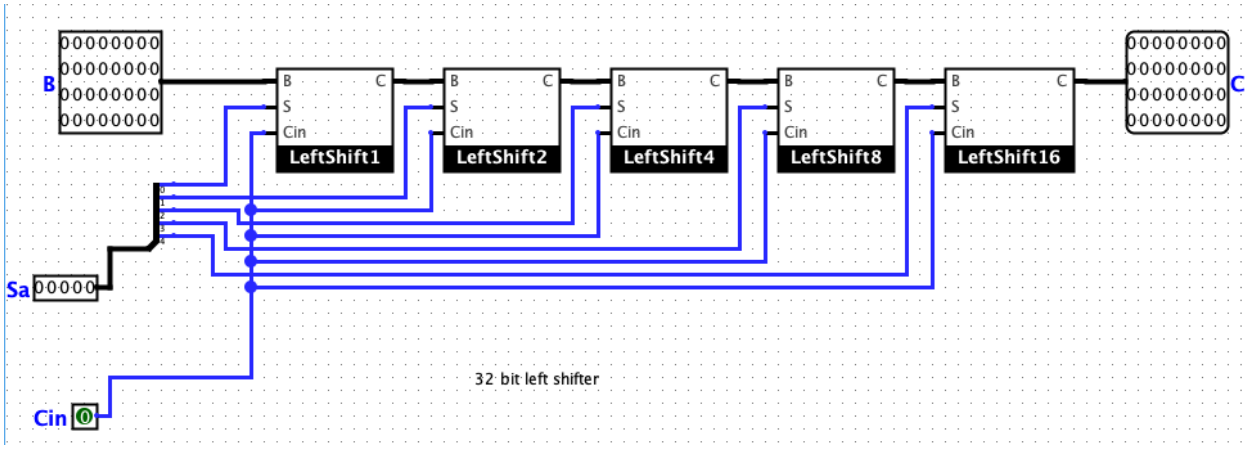

Figure 4: 32-bit Left Shifter

Combined Left/Right Shifter

Since the project disallowed the creation of multiple shifting components, it was necessary to implement right arithmetic and logical shifting by transforming the inputs to the left shifter. A right shift can be implemented by reversing the input, performing a left shift of the appropriate amount, and then reversing the output. Additionally, I had to select Cin based on whether an arithmetic shift was being performed (in which case it should be the sign bit) or not (in which case it should always be 0). This control is depicted in Figure 5. Note that one bit is used to indicate whether a left shift is being performed (which corresponds to the 2nd bit of the opcode if a shift is being performed) and another is used to indicate whether or not the shift is arithmetic (which corresponds to the 0th bit of the opcode if a right shift is being performed).

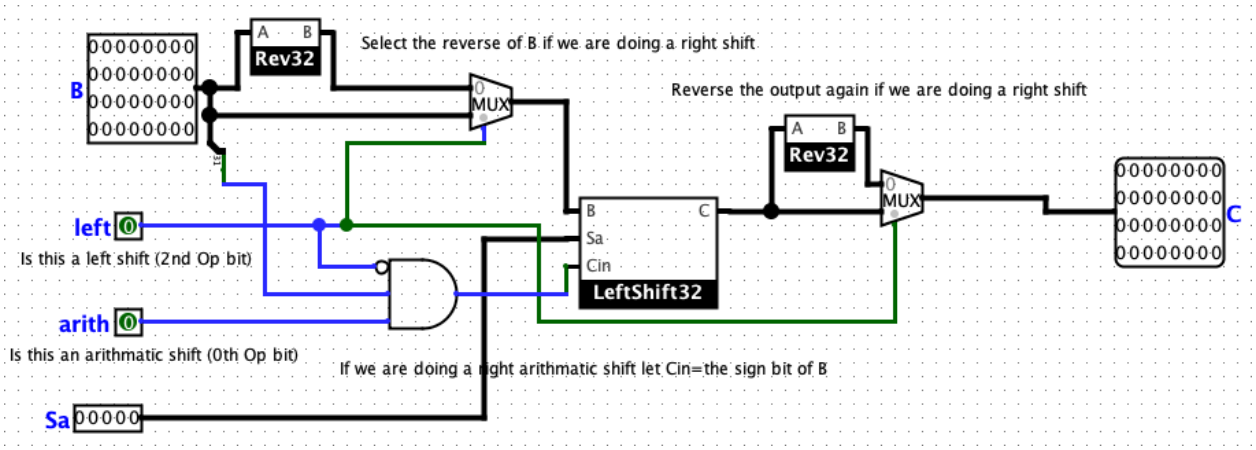

Figure 5: Left/Right Shifter Subcircuit

Combined Comparator Unit

I decided to implement the four comparison operations (A != B (ne), A == B (eq), A <= 0 (le), A > 0 (gt)) in one subcircuit. To select which comparison to perform, I had one signal that corresponded to it being an eq or gt (the 0th opcode bit if a comparison is being performed), and one signal that corresponded to it being an le or gt (the 1st opcode bit if a comparison is being performed). These inputs are labeled in Figure 6. To perform the equality tests, I xor'ed the two inputs, which should yield 0 if both inputs are the same and some arbitrary non-0 number otherwise. To check if this multibit signal is indeed 0 I decided to or all 32 bits together. Later I invert this result if we are doing an eq. To perform the greater/less than tests on A, I check both the sign bit of A and whether A=0. To avoid unnecessary duplication, I used a MUX to select the inputs to the huge multi-input or so that I can use it both to check if A nor B and if A are 0. After the result of the comparison has been computed, I need to 0-extend it to the required 32-bit value.

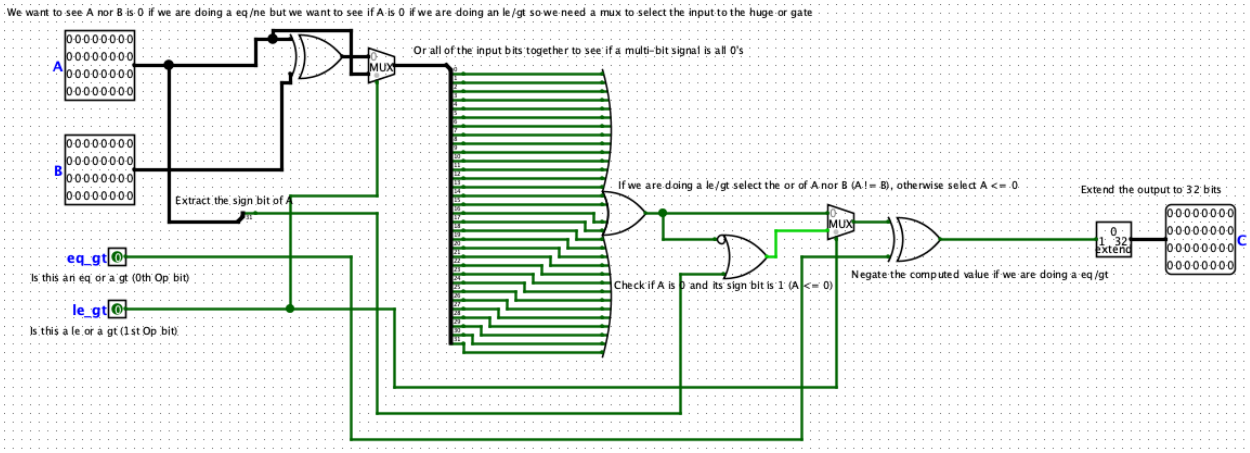

Figure 6: Comparator Subcircuit

Control Logic

Once all of the required values have been computed, the correct one needs to be selected based on the opcode. To do this using only 1-bit MUXs (as required by the assignment) and with minimal gates I decided to build a decision tree in which each step selects on one bit of the opcode. In this tree, shown in Figure 7, each opcode is paired up with another that differs from only one bit. The appropriate value is then selected by that bit of the opcode. In the following stages of the tree, we combine the results of multiple different operations by selecting on one bit at a time. At the end of the tree, we have no more remaining bits to select on, but the correct value will have been selected.

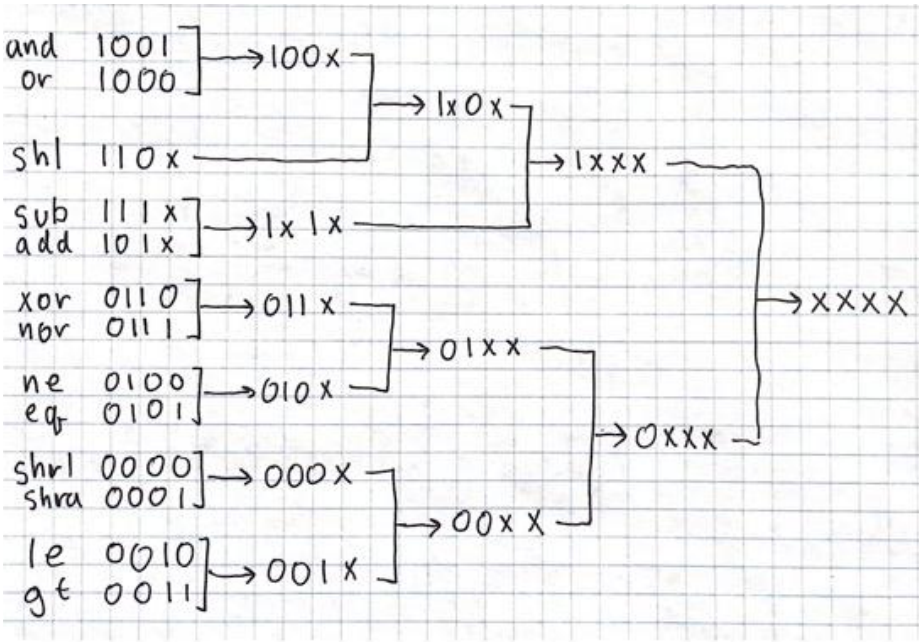

Figure 7: Value Selection Tree

When implementing this decision tree in my ALU (as shown in Figure 8) I had to make a few modifications. Since some of the subcircuits I had created contained values that were not immediately combined in my decision tree (e.g. the left shift versus the two right shifts) I had to split the output of several subcircuits and then re-select them. This works because the output of each subcircuit has already been selected to be the correct variant of each operation. Additionally, I had to assert V iff we are executing an add or subtract operation. To do this, I used an and gate to combine the V from the adder/subtractor and the 1st and 3rd bits of the opcode (which are only both true for adds and subtracts). Therefore V will be asserted only when appropriate.

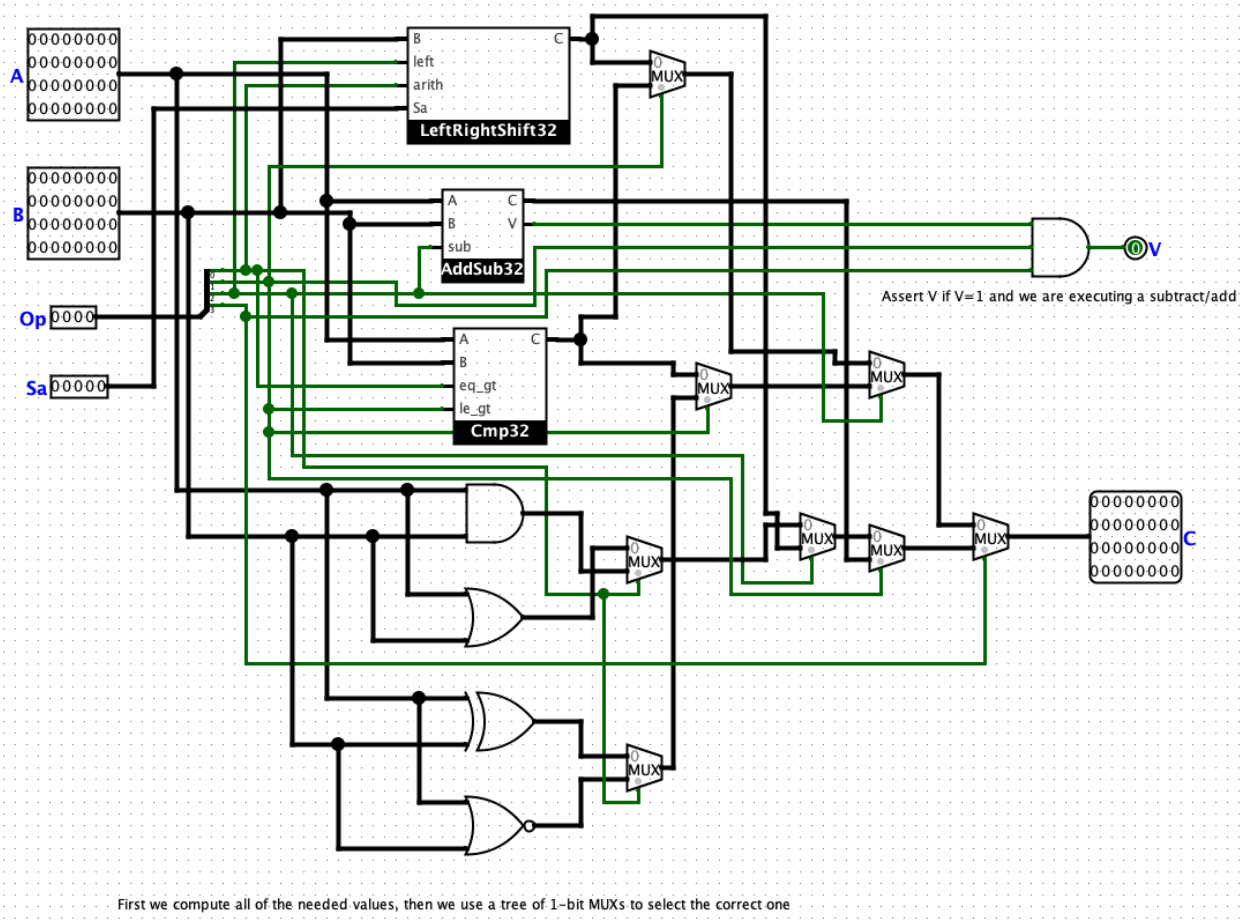

Figure 8: ALU Control Logic

Design Decisions/Tradeoffs

I made a variety of design decisions when constructing my ALU. Most of these decisions were constrained by the requirements of the assignment (mostly the requirements that there be only one copy of each subcircuit and that only 1-bit MUXs can be used). However, there is one major choice that I faced: Whether to group all of the comparison operators together or to separate the equality tests from the less than/greater than. In the end, I chose to combine them so that I could reuse the multi-input or that I built to check if 32-bit signals were all 0's.

Testing

I used a computer program to test my ALU and its various subcircuits. For each circuit, I wrote a function that took the same inputs as the circuit and returned the specified outputs. Then, I wrote code that would randomly generate test cases based on this function. In addition to purely random test cases, I used exhaustive test cases for all possible combinations of edge values (e.g. 0, -1, MIN_INT, MAX_INT) and opcodes.