Instructions

Instructions

There is one deliverable for this assignment: a LOGISIM circuit file named `alu.circ`. Commit and push your `alu.circ` to GitHub, and submit via Gradescope. Only one member of your team needs to submit on Gradescope, though you should remember to attach your partner's name to the submission. You are expected to practice pair programming, where both partners are working together on a shared screen to complete this assignment. See the explanation of `pair_programming.pdf` in the syllabus for more-detailed guidelines.

Designing an ALU

The Arithmetic Logic Unit (ALU) combines a variety of mathematical and logical operations into a single unit and forms the heart of most computer systems. For example, a typical ALU might perform addition, subtraction, magnitude comparison, AND, and OR operations. On this assignment, you will be building a simple ALU.

The interface of the circuit you will design is shown in figure 1. The inputs and outputs are described in table 1.

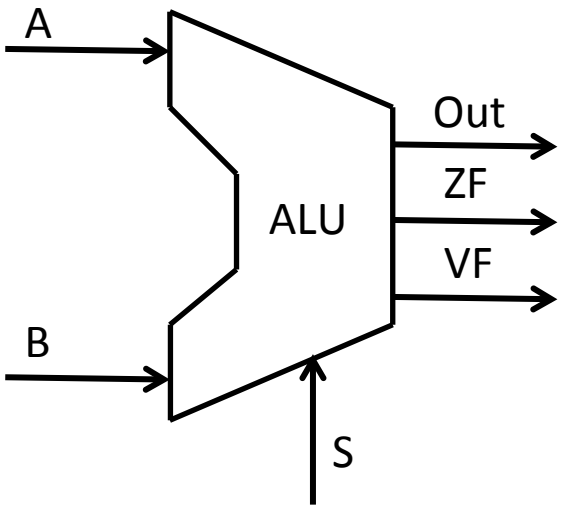

Figure 1: ALU interface

Table 2 provides a mapping from the settings of $S_2 : S_0$ to the result produced by the ALU. For example, when $S = 000$, the output of the ALU (i.e., the value on the *Out* pin) should simply mirror the value of the input A. When $S = 011$, the value of *Out* should be the result of incrementing the value of B by 1, and so on.

Pin Name	Type	Bit Width	Description
S	Input	3	Selects what arithmetic/logical operation must be performed on the two operands. The descriptions of each of the operation codes are provided in table 2.
A	Input	8	The first operand to the ALU.
B	Input	8	The second operand to the ALU.
Out	Output	8	The result produced by performing the operation specified by S on inputs A and B.
ZF	Output	1	This is a *zero flag* — this output should be high if and only if the result (Out) is 0.
VF	Output	1	This is an *overflow flag* — this output should be high if and only if an addition, increment or subtraction was performed and this resulted in a **signed** overflow.

Table 1: Input/output pin descriptions

Notes and Constraints

- Here's the general idea behind how to design your ALU: you should perform all the possible computations in parallel, irrespective of the value of S. Then, use the bits of S to multiplex between all the different results and forward only the correct answer to the Out pin.

- You should use built-in digital logic components in the Logisim libraries whenever possible. For example, you do not have to build your own 8-bit adder — you should just use the 8-bit adder that is part of Logisim. You already know how to build an adder from class, so let's take advantage of some abstraction.

- **This is important:** to receive full credit, your ALU design must only use **one** instance of an adder circuit. You may *not* use subtractors, comparators, negators or any other circuits that you may find under the "Arithmetic" tab in Logisim. This constraint may seem unnecessarily restrictive, but this is in fact the reality of how hardware is often designed; circuits like adders and subtractors use a large number of gates and are thus expensive. Repurposing the same hardware to perform multiple operations can be an important cost-saving measure. Use your single 8-bit adder to implement the addition, increment and subtraction operations. You can use some strategic multiplexing with

S_2	S_1	S_0	Output
0	0	0	A
0	0	1	B
0	1	0	$A + 1$
0	1	1	$B + 1$
1	0	0	$A + B$
1	0	1	$A - B$
1	1	0	A AND B
1	1	1	A OR B

Table 2: ALU opcodes

the bits of S to control what operands get fed to this adder.

- If you find yourself needing to supply a constant value like a 0 or a 1 to a specific input, do not use an input pin for this purpose: use a constant instead (under the "Wiring" tab in Logisim).

- Debugging tips:

  - Test your ALU as you go. Add a little functionality, and then test that functionality for a variety of inputs using the Poke tool. Then add some more functionality.
  - Keeping your circuit schematic neat can help avoid bugs. Tunnels are handy for eliminating large numbers of crisscrossing wires. You can read about tunnels in the Logisim library reference.
  - A common source of bugs is wires getting crossed *underneath* other components. Neatness can help with this, but it's good to check from time to time that nothing has gotten crossed by dragging your components around on the canvas.
  - Probes enable you to examine the value of a specific wire and are handy for debugging. These too are documented in the Logisim library reference.

Submission

Commit and push your `alu.circ` to GitHub, and submit via Gradescope. Only one member of your team needs to submit on Gradescope, though you should remember to attach your partner's name to the submission. Please pay attention to the given specifications to avoid significant autograder failures; in particular, make sure that your submission preserves all of the following (as is the case in the starting point circuit):

- Ensure that your file is named `alu.circ`.
- Ensure that the input and output pins in your circuit are named exactly as specified in this handout, including capitalization.
- Ensure that your circuit is named `ALU` within Logisim.
- Ensure the "appearance" of your circuit in Logisim matches figure 1.