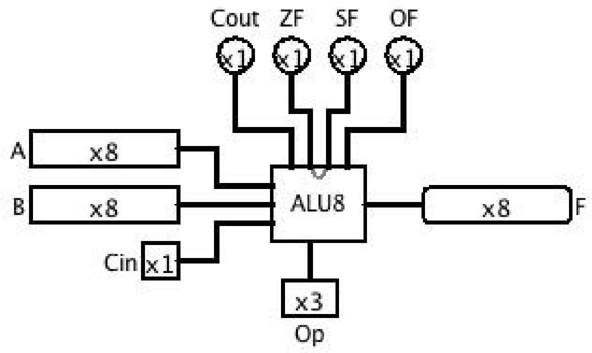
cs281: Introduction to Computer Systems

Project Lab 2: The Arithmetic Logic Unit (ALU)

Assigned: Sept. 19, Due: Sept. 29

1. The purpose of this laboratory is to:
   - reinforce our experience in designing and implementing combinational logic circuits,
   - gain experience in abstraction at the hardware level and assemble larger wholes from smaller parts,
   - develop expertise with Logisim, an educational software package for designing and simulating digital circuits.
2. This lab is to be completed individually. You will only submit a copy of your Logisim file; there is no accompanying lab report. You will be given a template `ALU.circ` file so that you have a framework from which to build your ALU, and it will help ensure that the external interface (pin definitions and placement) will be consistent with that expected for our grading of your work.
3. We start with a view from outside the ALU circuit you will create. The following picture shows a Logisim main circuit that provides inputs and outputs to a circuit named ALU8. It is the ALU8 circuit that you will design and build in this project.



   The inputs are shown on the left and bottom of the ALU8 circuit and the outputs are on the right and top of the circuit. As a matter of convention, we use pins on the left of a chip for general inputs, the bottom of the chip for *control/select* inputs, the right for outputs, and here we use the top of the chip for a set of single-bit outputs that comprise the condition code flags of ALU8. Note that A and B are 8-bit-wide inputs, Op is a 3-bit-wide input, and Cin is a 1-bit-wide input. F is an 8-bit-wide output, and $Cout$, ZF, SF, and OF are single bit outputs.
4. The Op determines a particular operation for the combinational logic of the ALU8. So, in general, we can think of the functionality as follows:
$$(F, Cout, ZF, SF, OF) = Op(A, B, Cin)$$
   So the set of five outputs are a function, determined by Op, of the other three inputs, A, B, and Cin.
5. Op is a three bit-wide input, so in the table below, we give the bit pattern for Op, a mneumonic for the operation, and then the computation specifying the F output in terms of A, B, and Cin. Since we all know C/C++ bitwise operations now, we will borrow that notation.

Op	**Mneumonic**	**Semantics**
000	add	`F = B + A + Cin`
001	notadd	`F = B + ~A + Cin`
010	and	`F = A & B`
011	xor	`F = A ^ B`
100	shiftR	`F = A >> 1;` $F_7 = Cin$
101	shiftL	`F = A << 1;` $F_0 = Cin$

Note that the `add` and `notadd` also add in the value of Cin. Cin is also used as the bit to "fill in" for the shift right (as the most significant bit) and the shift left (as the least significant bit). The reason for this design is to allow this ALU8 to be chained together with three more ALU8 chips to build a 32 bit ALU. Note also that we do not need explicit subtraction, because we can obtain that operation by using the `notadd` operation and setting Cin to 1, effectively getting the "complement and adding one" semantics we need for two's complement.

6. The following table gives the meanings for the four single bit condition codes.

Flag	**Operations**	**Description**
ZF	*all*	Zero Flag: If the result of the current ALU operation, F, is zero, then ZF is 1. If the result of the operation is not zero, then ZF is 0.
SF	*all*	Sign Flag: Reflects the most significant bit of the result of the current ALU operation (i.e. F_7). When interpreted as an 8-bit two's complement, the sign bit indicates a negative result.
OF	`add` `notadd`	Overflow Flag: This bit is asserted if the current operation caused a two's complement overflow–either positive or negative, and is deasserted otherwise.
	`and` `xor` `shiftR` `shiftL`	The OF flag is 0 for all four of these other operations.
Cout	`add` `notadd`	Carry Flag: This bit is 1 if the addition caused a carry-out from the most significant bit position, so an unsigned overflow.
	`shiftR`	This is the bit "shifted off" from operand A, aka A_0.
	`shiftL`	This is the bit "shifted off" from operand A, aka A_7.
	`and` `xor`	Always 0 for these two operations.

7. Using the **Logisim** digital logic design and simulator package, design and implement the above-described ALU8. Start with the `ALU.circ` file provided on the course web page. Using this template is covered in more detail in the guidance section below. Your implementation may **only** use 1-bit wide elements, although you are encouraged to build sub circuits that create multi-bit devices from 1-bit wide elements. You are also required to build your own MUXes, as needed, directly from gates.

8. This lab will be graded based on a total of 50 points. 40 of the points will derive from correct operation of the ALU for all of my test cases. I will be grading for both the output F as well as *all four condition codes* over a variety of test inputs, so make sure these are implemented correctly per the specification given above. The other 10 points will be based on your circuit design. Some of the grading criteria for the 10 points on these design elements:

   - Following conventions of inputs on the left, outputs on the right, and being able to read the "flow" of the circuit from left to right.
   - Abstraction – appropriate use of sub circuits with appropriate pin interfaces and labels so that a problem's decomposition is reflected in the design and hierarchical use of circuits.
   - Neatness – appropriate "clean" routing of wires and placement of sub circuits to convey an organized and orderly design.

9. *Graduated grading:* To allow prioritization of the elements of the design and implementation, we give two levels of deliverables corresponding roughly to B-level work versus A-level work. Both levels have in common the 10 points for circuit design criteria.

   - For the B-level work, 32 of the 40 correctness points will assess the operation of `add`, `notadd`, `and`, `xor` for the result F as well as the condition codes `ZF`, `SF`, `Cout` appropriate to these operations.
   - For the A-level work, the remaining 8 correctness points will assess the operation of `shiftR`, `shiftL` for the result F, the condition code `Cout` for these operations, and the `OF` for `add` and `notadd`.

Guidance

The guidance given here should help you to order and prioritize your development of the ALU8 circuit. It would be helpful for you to open the `ALU.circ` file and look at the template structure provided as you read this guidance.

1. When you open the template structure, the *current* circuit (the one with the magnifying glass icon) should be `ManualTop` and the design pane should correspond to the top-level picture above. **You should make NO changes to this circuit.** By extension, this means that you should make no changes to the interface for the `ALU8` circuit that would then result in needing to change `ManualTop`.

2. Notice that, in addition to `ManualTop`, the circuits named `ALU8`, `Adder8`, `And8`, and `Xor8` have been created. If you double-click any of these circuits to see their contents, you will observe that the only elements in each of these are the input pins and the output pins that define the external interface of the circuit. In Logisim, the placement of the pin devices on the design page determine their order and placement when the device is used. Pins on the top (and facing down) of the design panel are on the top of the device when it is used, and the order of the pins from left to right corresponds to the order on the design page. Likewise for pins on the left, right, and bottom of the design page. So for each of these already-created circuits, you must keep the top/bottom/left/right positioning of the pins as given to you, but as long as you keep them on the same "edge" and relative position, you can move them to accommodate your circuit design.

3. Before you can build an 8-bit wide adder to fill in the body of the Adder8 circuit, you will need to design and build a 1 bit full adder. You can use the design steps from the hardware lab to help this process, with inputs of 1-bit wide A, B, and Cin and 1-bit wide $Cout$ and S as outputs. This should be its own circuit and you should test it well before proceeding.

4. The other low level combinational circuit you may wish to implement is a multiplexer. In the hardware labs, we designed and built a 1-bit wide 2-1 mux. You can start by building this device. You will then need to extend your multiplexer, and there are two dimensions of extensions you may want or need:

   - Extending a 1-bit wide multiplexer to an 8-bit wide multiplexer. Note that this is the "width" of the inputs, but you still would have an A and a B, but each are 8 bits wide. For a 2-1, you still just have a single bit for the selector, S.
   - Extending from a 2-1 multiplexer to a 4-1 or 8-1. In this case, the width of the inputs stays the same, but the number of inputs (A, B to A, B, C, D, for instance) and the number of bits in the selector, which is always $\log_2$ of the number of inputs.

5. Once the building blocks are in place, focus one at a time on the 8-bit versions of the main building blocks, and make sure a single operation to the ALU works before complicating the design and adding in other high level operations. You should use appropriate width inputs (like 8-bit) and should use *splitters*, available in the Logisim *Wiring* folder, to break out the individual single-bit wide wires from the 8-bit aggregate.

6. For some of the condition code bits, the OF in particular, you will want to use our combinational logic design strategy of determining the inputs and building a truth table for the OF output and then designing the circuit to compute the OF.

7. Keep abstraction in mind and build sub-circuits as you go. Retro-fitting sub-circuits is much less helpful in terms of cleaner/simpler circuits and design, and the possibility of leveraging a sub-circuit in multiple places.