Assigned: Tuesday 15 September

Due: Monday 21 September 10:30 pm

Summary: You will build a 4-bit ALU in Logisim that implements standard arithmetic and logical operations: bitwise AND, OR, NAND, and NOR; addition and subtraction; and less-than comparison.

Collaboration

For this assignment, you may work with one or two partners from the class section. That is, you may work alone or in teams of up to three students. Teams of two are preferable. **Solutions submitted by teams of four or more students or with students from the other section will receive an automatic zero.**

This assignment should be a learning experience for everyone in the team, so do not "divide and conquer." You should use a pair programming approach as you build your circuits.

To reiterate the honesty policy of the syllabus: The work submitted by your team must be your team's own work. You may discuss (with words) your design with other groups, but you must not share or show your circuit files for any of the exercises in this assignment.

Getting Started

The testing functionality used for grading (and your own testing) requires a modified version of Logisim available from Cornell University's CS 3410 course (download).[1]

Copy three files from `~weinman/courses/CSC211/assignments/alu` to begin:

`4-bit-alu.circ` The Logisim scaffold where you will build your ALU.

`1-bit-alu-tests.txt` A limited set of tests for the 1-bit ALU

`4-bit-alu-tests.txt` A limited set of tests for the 4-bit ALU

Save frequently as you work. The provided `.circ` file includes most of the inputs and outputs you will need, but you will be adding a few subcircuits and inputs later in the assignment. **You must give these precisely the names (including upper and lower case) specified in the assignment.**

The following command syntax is used to test circuits:

```
java -jar /path/to/logisim.jar -test "subcircuit name" tests circuit
```

The arguments after `-test` are the name of the subcircuit being tested, the text file listing test cases, and the circuit file that contains the subcircuit to test.

Problems

This section contains the following tasks

- Problem 1: Building a 1-bit ALU
- Problem 2: Extending the 4-bit ALU
- Problem 3: Implementing `slt`
- Problem 4: Detecting overflow
- Problem 5: Correcting `slt` for overflow

[1] The MathLAN version from the Logisim lab uses this version.

Problem 1: Building a 1-bit ALU

In the "1-bit ALU" subcircuit, build a 1-bit ALU with the following inputs and outputs:

Inputs

`a` and `b` Inputs operands that should be ANDed, ORed, added, etc.

`operation` The 2-bit operation selector:

`00`	AND
`01`	OR
`10`	addition
`11`	nothing (for now)

`a_invert` and `b_invert` These inputs invert `a` and `b`, respectively, before they are connected to any of the ALU's operations. Asserting both `a_invert` and `b_invert` allows you to compute NOR and NAND (using AND and OR operations according to DeMorgan's laws) without any additional gates. Asserting just `b_invert` (with `c_in`) numerically negates the `b` input, which allows you to perform subtraction (and `slt` in Problem 3).

`c_in` Carry input from a chained adder or an added bit used when negating a number. Recall that in two's-complement $-b = \bar{b} + 1$. Asserting `c_in` lets you easily add one to an inverted `b` (via `b_invert`).

Outputs

`result` Result of the selected operation after all inversions, carry logic, etc.

`c_out` Carry output of `a` and `b`. Note this should be `1` if `a` plus `b` plus `c_in` is two or larger, even when operation is set to something other than addition or subtraction.

Do not build your own adder or multiplexers; use Logisim's built-in multiplexers, adders, and any logic gates you may need (found under the Gates, Plexers, and Arithmetic categories of the explorer pane). You should only use one adder in the 1-bit ALU; with `b_invert` and `c_in` asserted you can perform subtraction without any additional logic.

Testing

To test your 1-bit ALU, run the following command, assuming your `.circ` and `.txt` files are in the current directory.

```
java -jar /path/to/logisim.jar \
     -test "1-bit ALU" 1-bit-alu-tests.txt 4-bit-alu.circ
```

This command reports how many test cases passed. For any failing tests, it will report the test number and the outputs that did not match the expected values.

Examine the test file and add additional test cases you need to be confident that your circuit works. Your grade depends on how well your circuit works (as well as out organized it looks), so write extensive tests.

Problem 2: Extending the 4-bit ALU

In the "4-bit ALU" subcircuit, chain four of your 1-bit ALUs together to build a 4-bit ALU. (Hover over a pin in a placed subcircuit to see its label.) The inputs and ouputs provided in this "4-bit ALU" subcircuit are the same, except `a`, `b`, and `result` are now four bits and have been connected to *splitters* that break each bit off into a separate wire. The pin numbered "0" is the rightmost (least significant) bit of the binary number in the pin's value.

Note: You should leave at least six breadboard rows between your 1-bit ALUs to leave room for the additional connections you will make in parts 3–5.

Testing

To test your 4-bit ALU, run the following command:

```
java -jar /path/to/logisim.jar \
     -test "4-bit ALU" 4-bit-alu-tests.txt 4-bit-alu.circ
```

You should add your own test cases because those provided do not use all of the ALU's basic operations. Try to write tests for untested operations and each special case (adding positive numbers, adding negative numbers, overflowing on addition, subtracting a negative number, etc).

Interlude: After you complete problems 1–3 but before continuing, save your circuit project as `complete-4-bit-alu.circ`.

Problem 3: Implementing `slt`

The `slt` (set if less than) operation produces an output of `0001` if `a` is less than `b`, otherwise it produces the output `0000`. To perform this operation, the ALU first subtracts `b` from `a`, then checks whether the most significant bit of the output is `1` (meaning this is a negative number, so `b` must have been larger than `a`). Another possible name for this operation would be "is `a` minus `b` negative?"

The following steps guide you in adding `slt` capabilities to `complete-4-bit-alu.circ`:

- Add a one-bit input pin labeled `less` (all lowercase) to your 1-bit ALU subcircuit.
- Connect `less` directly to the fourth input of your main multiplexer (controlled by `operation`).
- Return to the 4-bit ALU subcircuit. Adding the `less` pin may have moved pins on the small version of your 1-bit ALU subcircuit. You can just reconnect them, or move each of the pins by right clicking on the 1-bit ALU in the explorer pane and choosing "Edit Circuit Appearance". Clicking a pin in this view will show you which part of the circuit that pin is connected to.
- Once your 1-bit ALUs are all reconnected, add a one bit Constant from the Wiring category in the explorer pane. Set this constant to zero and connect it to the `less` input of each 1-bit ALU. You can create a `0` constant for each 1-bit ALU if this is easier to fit into your circuit.
- Create a new subcircuit for the most significant bit called "MSB ALU". Return to your 1-bit ALU, select all components (from the Edit menu), copy, then paste these into the MSB ALU subcircuit.
- Create a new one-bit output pin called `set` in the MSB ALU subcircuit. Connect the appropriate output of your `adder` to the set output. (Consult consult Figure C.5.10 in your text.)
- Return to the 4-bit ALU subcircuit, select the 1-bit ALU connected to the most significant bits (the pins labeled "3" on the splitters), and delete this ALU.
- Replace the ALU you deleted with your new MSB ALU.
- The least significant bit (LSB)—connected to "0" pins from the splitters—ALU's `less` pin should not be wired to constant `0`; instead, connect the `set` output of the MSB ALU to the `less` input of the LSB's ALU.
- Be sure all the other inputs and outputs for the MSB ALU adder are connected.

To execute an `slt` instruction, assert `b_invert` and `c_in` to negate `b`. Then set `operation` to `11` to select the `slt` output. All of the ALUs produce a `0` except the LSB, which echoes the output from the MSB's `set` pin.

Write additional test cases for your expanded 4-bit ALU. You may want to add additional test cases for the 1-bit ALU and the MSB ALU. Your `slt`-enabled ALU should still pass all prior tests.

Problem 4: Detecting overflow

Hardware typically reports whether an arithmetic operation resulted in overflow of a signed two's-complement number (adding to a number until it wraps around to a negative, or subtracting until it wraps around to a positive). Note that this overflow for signed numbers is not the same as unsigned overflow, which is indicated in the MSB ALU's `c_out`.

Create a new subcircuit called "overflow". In this subcircuit, create and label any inputs you need for detecting overflow and create a single output labeled `overflow`.

Drop an instance of "overflow" inside your MSB ALU and wire it appropriately, adding another output labeled `overflow` to the MSB ALU connected to your overflow detector. Finally, connect *that* to an `overflow` output in the 4-bit ALU subcircuit.

Problem 5: Correcting `slt` for overflow

Our implementation of `slt` is not quite correct. To see where this implementation fails, consider the following input settings:

- `a = 1001` (-7)
- `b = 0110` (6)
- `operation = 11` (for `slt`)
- `b_invert = 1` (to negate `b` for `slt`)
- `c_in = 1` (to negate `b` for `slt`)

The result of $-7 - 6$ is `1+1001+1001=0011`, a positive number, so `slt` will produce `0000`, even though $-7 < 6$.

Correct your circuit so the `slt` operation accounts properly for overflow by adding the appropriate logic either to the "4-bit ALU" subcircuit, or to the "MSB ALU" subcircuit.

What to turn in

- Files `4-bit-alu.circ` and `complete-4-bit-alu.circ`.
- Each subcircuit from both files—clearly labeled—as an image in a single PDF in the following order
  - `4-bit-alu.circ`
    - 1-bit ALU
    - 4-bit ALU
  - `complete-4-bit-alu.circ`
    - 1-bit ALU
    - MSB ALU
    - overflow
    - 4-bit ALU

I will use scripts to run tests on your `.circ` files and print your PDF to write comments. Failure to include either will result in significant grade penalties.

Acknowledgments

 was developed in collaboration with Charlie Curtsinger. Problem 5 is adapted from Patterson and Hennessy (2008), Exercise C.24 (p. C-83).[2]

[2] Patterson, D. A., & Hennessy, J. L. (2008). *Computer organization and design: The hardware/software interface* (Fourth Edition). Morgan Kaufmann: Burlington, MA.