CS 252 : Computer Organization

Sim #3

An ALU

due at 3pm, Fri 6 Mar 2020

1 Purpose

In this project, you'll implement an ALU. You will build three new classes: one to implement a MUX, one to implement a 1-bit ALU element, and one to implement the entire ALU.

1.1 Limitations

Hopefully, I've convinced you (in Simulation Project 2) that any sort of logical expression can be encoded as logic gates. So to simplify the code, I'll let you use Java logical operators (`&&, ||, ~, &, |, !`) again. Likewise, you may use the `==, !=` operators. However, you **may not** use `<, <=, >, >=` (except in `for()` loops). As before, you may use `++` in any `for()` loops that you write, and you may use addition or subtraction (to figure out array indices) when copying your carry bits from one element to the next. But you **must not** use addition or subtraction anywhere else.

Just like in Simulation 2, you are not allowed to use `if()` anywhere - not even in the MUX! You will need to compose your MUX using **only** logical operators.

1.2 Required Filenames to Turn in

Name your files

```
Sim3_MUX_8by1.java
Sim3_ALUElement.java
Sim3_ALU.java
```

I have released a few utility classes in the project directory. If you want to use them in your solution, you may. However, you must use those classes and **only** those classes - and don't modify them in any way! (If you turn in copies of these to Gradescope, I'll ignore them.)

2 No Testcases This Time

This time, I won't provide any testcases besides a very basic one detailed below. (As always, sharing testcases on Piazza is **encouraged.**)

The test case does not do anything interesting, and doesn't print anything out - it simply double-checks that you have the right input and output fields for each of the 3 required classes.

Use the one testcase I give you to make sure that your fields are named correctly. Use testcases that you have written to make sure that your **results** are correct!

NOTE: We will not test any input for the LESS testcase where overflow would be an issue. So you don't have to worry about that!

3 Tasks

Like in Simulation 2, you will be implementing Java classes to model various logical components - and you will do it mostly by composing smaller pieces into larger ones.

The MUX and ALU classes will each have a single `execute()` method, just like in previous Simulations. But the ALU Element class will be a little different: it will have two different methods - representing the two-pass nature of how the ALU works.

3.1 MUX

Write a class named `Sim3_MUX_8by1`. It must have a 3-bit `control[]` input, and an 8-bit `in[]` input; both are arrays of `RussWire` objects. It must have a single `RussWire` output, which is named `out` (not an array).

This class must have a single `execute()` method.

This class models an 8-input MUX, where each input is a single bit wide. Since it has 8 inputs, there are 3 control bits. (As with our adders, treat element 0 of the control array as the LSB of the control input.)

Something to think about:
How could you adapt this class to represent a 2-input MUX as well - without adding any new control bits?

3.2 ALU Element

Write a class named `Sim3_ALUElement`. It must have several inputs:

- `aluOp` (3 bits)

  As we normally do, element 0 is the LSB of this field. It has 5 possible values:

  - 0 - AND
  - 1 - OR
  - 2 - ADD
  - 3 - LESS
  - 4 - XOR

(You may assume that this input will never be set to 5,6,7.)

Of course, XOR is not a standard ALU operation according to the design in the textbook - I'm adding it just for fun.

- `bInvert` (1 bit)
- `a,b` (1 bit each)
- `carryIn` (1 bit)
- `less` (1 bit)

  This input is the value that this ALU Element should give as the result if `aluOp==3`. (Obviously, this input will not be set before the first pass, so it should only be read during the second pass.)

The class must also have several outputs:

- `result` (1 bit)

  This is the output from this ALU element. It might be the result of calculating AND, OR, ADD, or LESS.

- `addResult` (1 bit)

  This is the output from the adder. It should always be set - no matter what the `aluOp` is set to. This is the add result for **this bit only**.

  (If `aluOp==2`, then `result` and `addResult` will - eventually - be the same.)

- `carryOut` (1 bit)

3.2.1 ALU Element - Two Passes

The `Sim3_ALUElement` class must have **two** execute methods, named `execute_pass1()` and `execute_pass2()`.

`execute_pass1()` represents (surprise!) the first pass through the ALU Element. When this is called, all of the inputs to the element will be set **except** for `less`. Your code must run the adder (including, of course, handling the `bInvert` input), and must set the `addResult` and `carryOut` outputs. It **must not** set the `result` output yet - because, at this point in time, the value of the `less` input might not be known.[1]

`execute_pass2()` represents the second pass through each ALU Element. When this function is called, all of the inputs will be valid (including `less`), and you must generate the `result` output.

When should you generate the AND value and OR value, and when should you copy the AND, OR, ADD values into the inputs of the MUX? You get to choose.

[1] Tempted to use an `if()` statement, and set the `result` sometimes? Remember that `if()` is banned!

3.3 ALU

Write a class named `Sim3_ALU`, which represents a complete ALU. It **must** use an array of ALU Element objects internally.

You must write this class so that it can handle inputs with any number of bits (not just 32). We'll pass the required size of the ALU as a parameter to the constructor of your class; you can assume that it will be ≥ 2. (We'll call this value X below.)

This class must have the following inputs:

- `aluOp` (3 bits)

  See the ALU Element description above to see how the ALU operation is encoded.

- `bNegate` (1 bit)

- `a,b` (X bits each)

This class must have a single X bit output, named `result`.

This class must have a single `execute()` method. It must deliver the inputs to the various ALU Elements, call `execute_pass1()` on them, set up the `less` inputs for each element, and then call `execute_pass2()` on each. (There are a few variations in the order in which you perform these steps. You may use the order that makes most sense to you. But write comments to explain what you're doing!)

4 How to Add

As I've noted above, you are **not** allowed to use any addition or subtraction in your classes (except for `for()` loops, and for copying carry bits from one element to the next).

However, like I did with Simulation Project 2, I've provided utility classes: a `FullAdder`, plus the smaller classes that it makes use of. I encourage you to use this inside each `ALUElement`; however, if you prefer to re-implement the adder using Java logical operators, that is also OK.

5 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).

- **Not** use any other files (unless allowed by the project spec) - since our grading script won't know to use them.

- Follow the spec precisely (don't change any names, or edit the files I give you, unless the spec says to do so).

- (In projects that require output) match the required output **exactly!** Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I **strongly recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

5.1 Testcases

For assembly language programs, the testcases will be named `test_*.s` . For C programs, the testcases will be named `test_*.c` . For Java programs, the testcases will be named `Test_*.java` . (You will only have testcases for the languages that you have to actually write for each project, of course.)

Each testcase has a matching output file, which ends in `.out`; our grading script needs to have both files available in order to test your code.

For many projects, we will have "secret testcases," which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcases of your own, in order to better test your code.**

5.2 Automatic Testing

We have provided a testing script (in the same directory), named `grade_sim3`. Place this script, all of the testcase files (including their `.out` files if assembly language), and your program files in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It **might** also work on your Mac, but no promises!)

5.3 Writing Your Own Testcases

The grading script will grade your code based on the testcases it finds in the current directory. Start with the testcases I provide - however, I encourage you to write your own as well. If you write your own, simply name your testcases using the same pattern as mine, and the grading script will pick them up.

While you normally cannot share code with friends and classmates, **testcases are the exception.** We encourage you to share you testcases - ideally by posting them on Piazza. Sometimes, I may even pick your testcase up to be part of the official set, when I do the grading!

6 Turning in Your Solution

You must turn in your code using gradescope. Turn in only your program; do not turn in any testcases or other files.