CS 252: Computer Organization

Sim #2
Logic Networks and Adders
(see class website for due date)

1 Purpose

Complex logic circuits, such as those inside a CPU, are implemented from simpler parts. In this project, you will implement some Java classes which represent simple hardware components, and then assemble them into a single large object which implements a multi-bit adder.

1.1 Required Filenames to Turn in

Name your files

```
Sim2_HalfAdder.java
Sim2_FullAdder.java
Sim2_AdderX.java
```

2 Even More Limitations!

Like Sim 1, you are not allowed to use any addition operator except for `++`, and even that operator is only legal as part of your `for()` loops. You **may** use subtraction, but only for calculating your indices for carrying logic. (It is very nice to be able to copy a value from column `i-1` into column `i`. Similarly, you can use it to find the **last** of the bits in the `AdderX` class, in order to calculate `carryOut` and `overflow`.)

In the same way, you must not use the `==` or `!=` operators; if you want to perform these comparisons, you'll have to figure out to do it with logic gates only.

Second, in this project, you **also** cannot use any `if()` statements[1] in any of your classes. This is because, in hardware, you don't have `if()` - instead, all hardware is running all the time. Instead, build your logic from gates: AND/OR/NOT/XOR etc.

Why am I placing such restrictions on you? Because I'm hoping to show you that we don't need fancy operators - or even something as simple as `if()` statements - in order to implement complex logic. **Everything** in your computer is just complex arrangements of `AND, OR, NOT` (plus memory).

---

[1] The ternary operator, if you know of it, counts as an `if()` statement, in this context! And so does switch/case.

3 No Testcases This Time

This time, I won't provide any **interesting** testcases - you'll have to write them yourself. (As always, sharing testcases on Discord is **encouraged.**)

I have provided a single testcase, however. It doesn't 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!

4 Tasks

Like with Sim 1, you will be implementing a few Java classes (no C code this time) to model a few simple logic circuits. Unlike Sim 1, each of the classes in this project will require that you build logic from simpler elements.

Unlike Sim 1, I won't be providing any Java files for you to modify; instead, you'll write your own from scratch.

All of the classes that you write will have an `execute()` method, which will be called when it's time to perform the logic of this object.

4.1 Common Requirements

Each of the classes you write will build (somewhat) complex logic out of simpler parts. As you did in Sim 1 (and as you'll see in the NAND example I've provided), you must create objects inside the constructor of your class - never inside `execute()`.

Basically, the **only** things that your `execute()` methods should do is to (a) copy values around, and (b) call `execute()` on other objects.

4.2 Testcase 00

Testcase 00, which I've provided, doesn't really check any logic. Instead, it simply exists to double-check the types of the inputs and outputs. If you can link with that testcase, then you'll be able to link with any other testcases I write.

4.3 Half Adder

Write a class named `Sim2_HalfAdder`. This class has the same inputs as AND/OR/XOR (two `RussWire` objects, named `a,b`). However, it has two outputs, `RussWire` objects named `sum` and `carry`. This class will implement a Half Adder: remember that this is a 1-bit adder, which does **not** have a carry-in bit.

4.4 Full Adder

Write a class named `Sim2_FullAdder`. This must have three inputs: `a,b,carryIn` and two outputs `sum,carryOut`.

You **must** implement the full adder by linking two half adders together.

4.5 Multi-Bit Adder

Write a class named `Sim2_AdderX`, which implements a multi-bit adder by linking together many full adders. (This is known as a "ripple carry" adder.) **Your adder must be composed of many different full adders; don't try to get more fancy than this[2].**

The constructor for this class must take a single `int` parameter; this is the number of bits in the adder. You may assume that the parameter is `>=2`; other than that, you must support any size at all. (We'll call this parameter `X` in the descriptions below.)

The inputs to this class must be two arrays of `RussWire` objects, each with exactly `X` wires. Name the inputs `a,b`. The outputs from this class are a `X` wire array named `sum`, and two single bits: `carryOut,overflow`.

This class has the same restrictions as all the rest. Since you're doing it `X` times, I'll allow you to use a `for()` loop in this method. Also note that subtraction is allowed for copying a carry bit from one column to another, so you can copy from `i-1` into `i`. However, remember that `if()` is still banned!

NOTE: This class doesn't perform subtraction - that's why there isn't any control bit to select it. But based on Sim 1, you know that it wouldn't be too hard to add one, if we had wanted.

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.

---

[2]Sometimes students want to use a bunch of full adders, and a single half adder for the least-significant-bit. I'll allow this, but frankly, it's not worth your time.

- 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

You can find all of the (non-secret) testcases in the zipfile, which is on the class website.

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_sim2`. Place this script, all of the testcase files (including their `.out` files if assembly language), and your program files in the same directory (in your Docker container).

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 Discord. 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 to GradeScope. Turn in only your program; do not turn in any testcases.