Project 2.1: ALU and Regfile

Computer Architecture I ShanghaiTech University
Project 1.2 Project 2.1 Project 2.2

IMPORTANT INFO - PLEASE READ

The projects are part of your design project worth 2 credit points. As such they run in parallel to the actual course. So be aware that the due date for project and homework might be very close to each other! Start early and do not procrastinate.

Overview

- You are allowed to use any of Logisim's built-in blocks for all parts of this project.
- Save frequently and commit frequently! Try to save your code in Logisim every 5 minutes or so, and commit every time you produce a new feature, even if it is small.
- Tests for a completed ALU and RegFile have been included with the lab starter code. You can run them with the command `./test.sh`. See the Testing section for information on how to interpret your test output.
- Don't move around the given inputs and outputs in your circuit; this could lead to issues with the autograder.
- Because the files you are working on are actually XML files, they're quite difficult to merge properly. **Do not work on the project in two places and attempt to merge your changes!** If you are working separately from your partner, make sure that only one person is working on the project at any given time. We highly recommend pair programming on this project; understanding the nuts and bolts should help you experience your magical software-meets-hardware-I-actually-know-how-a-computer-works-now moment (and will prepare you to tackle datapath problems on exams).

Please read this document *CAREFULLY* as there are key differences between the processor we studied in class and the processor you will be designing for this project.

Getting started

Similarly to Project 1, we will be distributing the project files through gitlab. Make sure you have the access to gitlab. In the group CS110_22s_projects you should have access to your project 2.1 project. Also, in the group CS110_22s, you should have access to the p2.1_framework.

Obtaining the Files

1. Clone your p2.1 repository from gitlab:
```
git clone https://autolab.sist.shanghaitech.edu.cn/gitlab/cs110_22s_projects/p2.1_xxx_xxx.git
```
(replace xxx to your project name)

2. In the repository add a remote repo that contains the framework files:
   ```
   git remote add framework
   https://autolab.sist.shanghaitech.edu.cn/gitlab/cs110_22s/p2.1_framework.git
   ```
3. Go and fetch the files:
   ```
   git fetch framework
   ```
4. Now merge those files with your master branch:
   ```
   git merge framework/master
   ```
5. The rest of the git commands work as usual.

- Note: Some files are managed by Git Large File Storage, you need to install it to pull all the files. See the setup part of Lab 5 for reference.

Exercise 1: Arithmetic Logic Unit (ALU)

Your first task is to create an ALU that supports all the operations needed by the instructions in our ISA (which is described in further detail in the next section). Please note that we treat overflow as RISC-V does with unsigned instructions, meaning that we ignore overflow.

We have provided a skeleton of an ALU for you in `alu.circ`. It has three inputs:

INPUT NAME	BIT WIDTH	DESCRIPTION
A	32	Data to use for Input A in the ALU operation.
B	32	Data to use for Input B in the ALU operation.
ALUSel	4	Selects what operation the ALU should perform (see the list of operations with corresponding switch values below).

and one output:

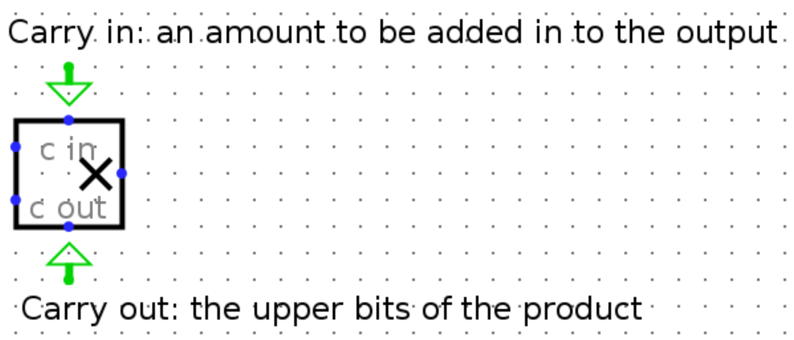
OUTPUT NAME	BIT WIDTH	DESCRIPTION
Result	32	Result of the ALU Operation.

Below is the list of ALU operations for you to implement, along with their associated ALUSel values. You are allowed and encouraged to use built-in Logisim blocks to implement the arithmetic operations.

SWITCH VALUE	INSTRUCTION
0	and: Result = A & B
1	or: Result = A | B
2	xor: Result = A^B
3	add: Result = A + B
4	sub : Result = A - B
5	mult: Result = (signed) A*B[31:0]
6	mulhu: Result = A*B[63:32]
7	mulh: Result = (signed) A*B[63:32]
8	divu: Result = (unsigned) A / B

SWITCH VALUE	INSTRUCTION
9	div: Result = (signed) A / B
10	remu: Result = (unsigned) A % B
11	srl: Result = (unsigned) A >> B
12	sra: Result = (signed) A >> B
13	slt: Result = (A < B) ? 1 : 0
14	sir: Result = (A[31:0] == B[0:31]) ? 1 : 0
15	cnto: Result = Number of 1s in both A and B

When implementing `mult` and `mulh`, notice that the multiply block has a "Carry Out" output (the adder block also has this, but you will not need this) located here:


Carry in: an amount to be added in to the output
c in
c out
Carry out: the upper bits of the product

Experiment a bit with it, and see what you get for both the result and carryout with negative and positive 2's complement numbers. You should realize why we made mulh extra credit.

Hints:

- `cnto` takes the value in both input registers and counts the number of ones existing in the input. The calculated number of ones will be stored in `$rd`.
- `mulh` is a little difficult, you should deal with upper 32 bits carefully using the built-in block "Multiplier".
- You can hover your cursor over an output/input on a block to get more detailed information about that input/output.
- You might find bit splitters or extenders useful when implementing `sra` and `srl`.
- Use tunnels! They will make your wiring cleaner and easier to follow, and will reduce your chances of encountering unexpected errors.
- A multiplexer (MUX) might be useful when deciding which block output you want to ouput. In other words, consider simply processing the input in all blocks, and then outputing the one of your choice.

NOTE: Your ALU must be able to fit in the provided harness `alu_harness.circ`. Follow the same instructions as the register file regarding rearranging inputs and outputs of the ALU. In particular, you should ensure that your ALU is correctly loaded by a fresh copy of `alu_harness.circ` before you submit.

Testing

When you run `./test.sh`, the ALU tests will produce output in the `tests/student_output` directory. We've provided a python script called `binary_to_hex_alu.py` that can interpret this output in a readable format for you. To use it, do the following: (you may have to use "py" or "python3" as is appropriate on your machine)

```
$ cd tests
$ python binary_to_hex_alu.py PATH_TO_OUT_FILE
```

For example, to see reference_output/alu-add-ref.out in readable format, you would do this:

```
$ cd tests
$ python binary_to_hex_alu.py reference_output/alu-add-ref.out
```

If you want to see the difference between your output and the reference solution, put the readable outputs into new `.out` files and diff them. For example, for the alu-add test, take the following steps:

```
$ cd tests
$ python binary_to_hex_alu.py reference_output/alu-add-ref.out > alu-add-ref.out
$ python binary_to_hex_alu.py student_output/alu-add-student.out > alu-add-student.out
$ diff alu-add-ref.out alu-add-student.out
```

Exercise 2: Register File (RegFile)

As you learned in class, RISC-V architecture has 32 registers. However, in this project, **You will only implement 9 of them (specified below)** to save you some repetitive work. This means your rs1, rs2, and rd signals will still be 5-bit, but we will only test you on the specified registers.

Your RegFile should be able to write to or read from these registers specified in a given RISC-V instruction without affecting any other registers. There is one notable exception: your RegFile should NOT write to x0, even if an instruction try. Remember that the zero register should ALWAYS have the value 0x0. You should NOT gate the clock at any point in your RegFile: the clock signal should ALWAYS connect directly to the clock input of the registers without passing through ANY combinational logic.

The registers and their corresponding numbers are listed below.

REGISTER #	REGISTER NAME
x0	zero
x1	ra
x2	sp
x5	t0
x6	t1
x7	t2
x8	s0
x9	s1
x10	a0

You are provided with the skeleton of a register file in `regfile.circ`. The register file circuit has six inputs:

INPUT NAME	BIT WIDTH	DESCRIPTION
Clock	1	Input providing the clock. This signal can be sent into subcircuits or attached directly to the clock inputs of memory units in Logisim, but should not otherwise be gated (i.e., do not invert it, do not "and" it with anything, etc.).
RegWEn	1	Determines whether data is written to the register file on the next rising edge of the clock.
Read Register 1 (rs1)	5	Determines which register's value is sent to the Read Data 1 output, see below.
Read Register 2 (rs2)	5	Determines which register's value is sent to the Read Data 2 output, see below.
Write Register (rd)	5	Determines which register to set to the value of Write Data on the next rising edge of the clock, assuming that RegWEn is a 1.
Write Data	32	Determines what data to write to the register identified by the Write Register input on the next rising edge of the clock, assuming that RegWEn is 1.

The register file also has the following outputs:

OUTPUT NAME	BIT WIDTH	DESCRIPTION
Read Data 1	32	Driven with the value of the register identified by the Read Register 1 input.
Read Data 2	32	Driven with the value of the register identified by the Read Register 2 input.
ra Value	32	Always driven with the value of ra (This is a DEBUG/TEST output.)
sp Value	32	Always driven with the value of sp (This is a DEBUG/TEST output.)
t0 Value	32	Always driven with the value of t0 (This is a DEBUG/TEST output.)
t1 Value	32	Always driven with the value of t1 (This is a DEBUG/TEST output.)
t2 Value	32	Always driven with the value of t2 (This is a DEBUG/TEST output.)
s0 Value	32	Always driven with the value of s0 (This is a DEBUG/TEST output.)
s1 Value	32	Always driven with the value of s1 (This is a DEBUG/TEST output.)
a0 Value	32	Always driven with the value of a0 (This is a DEBUG/TEST output.)

The DEBUG/TEST outputs are present for testing and debugging purposes. If you were implementing a real register file, you would omit those outputs. In our case, be sure they are included correctly--if they are not, you will not pass.

You can make any modifications to `regfile.circ` you want, but the outputs must obey the behavior specified above. In addition, your `regfile.circ` that you submit *must* fit into the `regfile_harness.circ` file we have provided for you. This means that you should take care not to move inputs or outputs. To verify changes you have made didn't break anything, you can open `regfile_harness.circ` and ensure there are no errors and that the circuit functions well. (The tests use a slightly modified copy of `regfile_harness.circ`.)

Hints:

- Take advantage of copy-paste! It might be a good idea to make one register completely and use it as a template for the others to avoid repetitive work.
- Because of the naming conventions that Logisim Evolution requires, because the outputs are named ra, sp, etc., you will not be able to name your registers ra, sp, etc. We suggest that you instead name them with numerical name of the register, e.g. x1, x2.
- I would advise you not to use the enable input on your MUXes. In fact, you can turn that feature off. I would also advise you to also turn "three-state?" to off. Take a look at all the inputs to a logisim register and see what they all do.
- Again, MUXes are your friend, but also DeMUXes.
- Think about what happens in the register file after a single instruction is executed. Which values change? Which values stay the same? Registers are clock-triggered--what does that mean?
- Keep in mind registers have an "enable" input available, as well as a clock input.
- What is the value of `x0`?

Testing

When you run `./test.sh`, the RegFile tests will produce output in the `tests/student_output` directory. We've provided a python script called `binary_to_hex_regfile.py` that can interpret this output in a readable format for you. To use it, do the following commands: (you may have to use "py" or "python3" as is appropriate on your machine)

```
$ cd tests
$ python binary_to_hex_regfile.py PATH_TO_OUT_FILE
```

For example, to see reference_output/regfile-x0-ref.out in readable format, you would do this:

```
$ cd tests
$ python binary_to_hex_regfile.py reference_output/regfile-x0-ref.out
```

If you want to see the difference between your output and the reference solution, put the readable outputs into new `.out` files and diff them. For example, for the regfile-x0 test, take the following steps:

```
$ cd tests
$ python binary_to_hex_regfile.py reference_output/regfile-x0-ref.out > regfile-x0-ref.out
$ python binary_to_hex_regfile.py student_output/regfile-x0-student.out > regfile-x0-student.out
$ diff regfile-x0-ref.out regfile-x0-student.out
```

Submission

Similarly to Project1, upload your `autolab.txt` to Autolab to submit your project.

You will NOT be submitting extra files. If you add a helper circuit, please place the circuit only in `alu.circ` and `regfile.circ`. Only changes to the files `alu.circ` and `regfile.circ` will be considered by the autograder. Besides, we only grade code on the `master` branch. If you do not follow these requirements, your code will likely not compile and you will get a zero on the project.

The last time of your submission to the git repo will count towards your submission time - also with respect to slip days. So do not commit to your git repo after the due date, unless you want to use slip days or you are OK with getting fewer points.

TAs: *Cunhan You, Yida Zhao*
Based on UC Berkely CS61C
Last Modified: 2022-03-2