CPU Project (Part 1): ALU and Regfile

Acknowledgement: UC Berkeley 61C

Overview

In this project you will be using Logisim to implement a simple 16-bit two-cycle processor, with an ISA that uses a subset of MIPS instructions on a slightly different architecture. This reduced ISA is called RPIS (Reduced Processor Instruction Set) and utilizes 8 registers that hold 16 bits of data, and a 16 bit memory address space. The datapath for this new instruction set follows almost exactly like the MIPS datapath you have been learning in class, with a few modifications in order to accommodate our reduced number of registers and address space.

In part I, you will make the Regfile and ALU.

IMPORTANT INFO - PLEASE READ

- You are allowed to use any of Logisim's built-in blocks for all parts of this project.
- **Use tunnels!** They will make your circuits much cleaner.
- **Save often.** Logism can be buggy and the last thing you want is to lose some of your hard work. There are students every semester who have had to start over large chunks of their projects due to this.
- Approach this project like you would any coding assignment: construct it piece by piece and test each component early and often!
- Sample tests for a completed ALU and Regfile have been included in the starter kit - just run the script `./run-sanity-test.sh`. We recommend running the sample tests locally, but they only work with **python 2.7**. As always, keep in mind that these tests are NOT comprehensive (see the Testing section for more details).
- **MAKE SURE TO CHECK YOUR CIRCUITS WITH THE GIVEN HARNESSES TO SEE IF THEY FIT! YOU WILL FAIL ALL OUR TESTS IF THEY DO NOT.**
  (This also means that you should not be moving around given **inputs** and **outputs** in the circuits).
- Because the files you are working on are not plain code and circuit schematics, they can't really be merged. **DO NOT WORK ON THE SAME FILE IN TWO PLACES AND TRY TO MERGE THEM. YOU WILL NOT BE ABLE TO MERGE THEM AND YOU WILL BE SAD.**

---

0) Obtaining the Files

Please download the skeleton code here.

1) Register File

Your task is to implement the 8 registers promised by the RPIS ISA (use the built-in Logisim registers!). Your regfile should be able to write to or read from any register specified in a given MIPS instruction without affecting any other registers. There is one notable exception: your regfile should NOT write to $0, even if an instruction should 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. Their purposes are the same as in MIPS, but less exist.

REGISTER #	REGISTER NAME
0	$0
1	$ra
2	$s0
3	$s1
4	$s2
5	$a0
6	$v0
7	$sp

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.).
Write Enable	1	Determines whether data is written on the next rising edge of the clock.
Read Register 1	3	Determines which register's value is sent to the Read Data 1 output, see below.
Read Register 2	3	Determines which register's value is sent to the Read Data 2 output, see below.
Write Register	3	Determines which register to set to the value of Write Data on the next rising edge of the clock, assuming that Write Enable is a 1.
Write Data	16	Determines what data to write to the register identified by the Write Register input on the next rising edge of the clock, assuming that Write Enable is 1.

The register file also has the following six outputs:

OUTPUT NAME	BIT WIDTH	DESCRIPTION
Read Data 1	16	Driven with the value of the register identified by the Read Register 1 input.
Read Data 2	16	Driven with the value of the register identified by the Read Register 2 input.
$s0 Value	16	Always driven with the value of $s0 (This is a DEBUG/TEST output.)
$s1 Value	16	Always driven with the value of $s1 (This is a DEBUG/TEST output.)
$s2 Value	16	Always driven with the value of $s2 (This is a DEBUG/TEST output.)
$ra Value	16	Always driven with the value of $ra (This is a DEBUG/TEST output.)
$sp Value	16	Always driven with the value of $sp (This is a DEBUG/TEST output.)

The outputs $s0-$s2, $ra, and $sp are present because those registers are more special than the others - they are for testing and debugging purposes and will be used in the autograder tests! If you were implementing a real regfile, you would omit those outputs. In our case, be sure they are included correctly! If they are not, we won't be able to grade your submission (and you'll get a zero. :( ).

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 to not reorder inputs or outputs. If you need more space, however, you can move them around if you are careful and maintain their relative positioning to each other. To verify your changes didn't break anything, simply open `regfile-harness.circ` and ensure there are no errors and the circuit functions well. We will be using a similar file to test your register file for grading, so you should download a fresh copy of `regfile-harness.circ` and make sure your `regfile.circ` is cleanly loaded before submitting.

HINTS: Muxes and demuxes are your friends. I would advise you not to use the enable input on your Muxes. In fact, you can turn that feature off. I would advise you to also turn "three-state?" to off. Take a look at all the inputs to a logisim register and see what they do.

2) Arithmetic Logic Unit (ALU)

Your second task is to create an ALU that supports all the operations needed by the instructions in our ISA. The ISA mostly consists of instructions from the MIPS ISA, though there may be additional "new" instructions and formats... Fortunately, you don't have to try to figure out all the required operations as we provide the list to you. The instructions in RPIS will be described in detail in part 2, but for now, you can trust that the operations listed below will cover all of the instructions in RPIS. Please note that we treat overflow as MIPS does with unsigned instructions, meaning that we ignore overflow (i.e. there is only one version of add/addi that does not have any special behavior if overflow occurs).

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

INPUT NAME	BIT WIDTH	DESCRIPTION
X	16	Data to use for X in the ALU operation.
Y	16	Data to use for Y in the ALU operation.
Switch	3	Selects what operation the ALU should perform (see the list of operations with corresponding switch values below).

... and three outputs:

OUTPUT NAME	BIT WIDTH	DESCRIPTION
Equal	1	High iff the two inputs X and Y are equal; low otherwise.
Result	16	Result of the ALU Operation.
Result2	16	Contains the upper 16 bits of mult when the function is mult. Contains some unspecified 16-bit value on other instructions.

And as previously promised, here is the list of operations that you need to implement (along with their associated Switch values). You are allowed and encouraged to use built-in logisim blocks to implement the arithmetic operations.

SWITCH VALUE	INSTRUCTION
0	sll: Result = X << Y
1	srl: Result = X >>> Y
2	add: Result = X + Y
3	and: Result = X & Y
4	or: Result = X | Y
5	xor: Result = X^Y
6	slt: Result = (X < Y) ? 1 : 0 Signed
7	mult: Result = X*Y[15:0]; Result2 = X*Y[31:16]

If Result2 is not specified, its behavior is undefined, and it can have any value. It should still have some defined value (X's are never OK!) but it doesn't matter what that value is, or if that value is constant or varying.

For `sll` and `srl`, the shifter circuits in Logisim will only shift up to the number of data bits, so for our 16 bit registers, you only have to worry about handling shifts of up to 15 bits. This entails modifying the Y input before inputting it into the shifter such that only the lower four bits of Y are used as the shift value while the higher order bits can be ignored.

NOTE: The multiplier circuit built into Logisim is signed (when operating on 16-bit numbers)! You are NOT expected to implement multiply from scratch. Use the built-in multiply blocks, and don't worry about it.

Some additional things to keep in mind:

- The output `Equal`, which is true iff X and Y are equal, must always output the correct comparison result regardless of the `Switch` value.

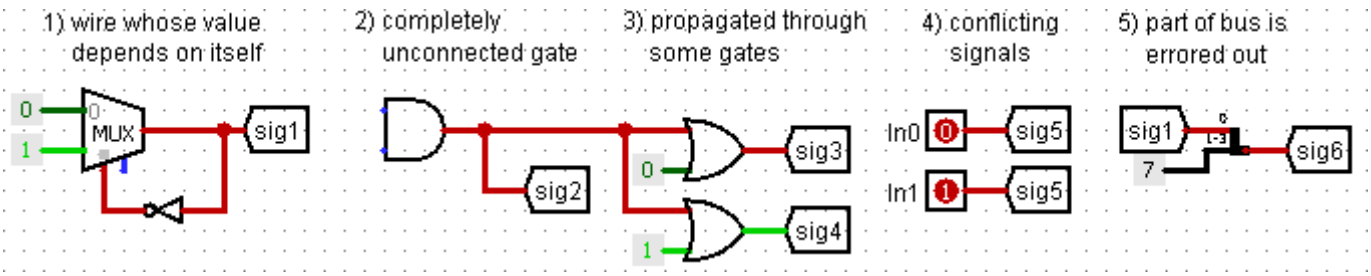
Note: Your ALU must be able to fit in the provided harness `alu_harness.circ`. Follow the same instructions as the register file regarding rearranging inputs and outputs of the ALU. In particular, you should ensure that your ALU is correctly loaded by a fresh copy of `alu-harness.circ` before you submit.

Logisim Notes

If you are having trouble with Logisim, ***RESTART IT and RELOAD your circuit!*** Don't waste your time chasing a bug that is not your fault. However, if restarting doesn't solve the problem, it is more likely that the bug is a flaw in your project. Please a message to your TA about any crazy bugs that you find, and we will investigate.

Things to Look Out For

- Do **NOT** gate the clock! This is very bad design practice when making real circuits, so we will discourage you from doing this by heavily penalizing your project if you gate your clock.
- **BE CAREFUL with copying and pasting from different Logisim windows.** Logisim has been known to have trouble with this in the past.
- When you import another file (Project --> Load Library --> Logisim Library...), it will appear as a folder in the left-hand viewing pane. The skeleton files should have already imported necessary files.
- Changing attributes *before* placing a component changes the default settings for that component. So if you are about to place many 32-bit pins, this might be desirable. If you only want to change that particular component, place it first before changing the attributes.
- When you change the inputs & outputs of a sub-circuit that you have already placed in `main`, Logisim will automatically add/remove the ports when you return to `main` and this sometimes shifts the block itself. If there were wires attached, Logisim will do its automatic moving of these as well, which can be extremely dumb in some cases. Before you change the inputs and outputs of a block, it can sometimes be easier to first disconnect all wires from it.
- Error signals (red wires) are obviously bad, but they tend to appear in complicated wiring jobs such as the one you will be implementing here. It's good to be aware of the common causes while debugging:



Logisim's Combinational Analysis Feature

Logisim offers some functionality for automating circuit implementation given a truth table, or vice versa. Though not disallowed (enforcing such a requirement is impractical), use of this feature is discouraged. **Remember that you will not be allowed to have a laptop running Logisim on the final.**

Testing

For part 1, we have provided you with a script for running tests on the ALU and regfile called "run-sanity-check.sh". Running `./run-sanity-check.sh` will copy your alu and regfile into the tests directory and run two ALU tests and two Regfile tests, which are stored in the tests directory as well. These tests drop in your work into a very slightly modified version of the harness and run it with a small set of inputs. The output is then compared against the provided reference. This sanity script assumes that python 2.7 is your default Python version (as it is on the `hive*` servers). If you have a more recent version of python, but still have python2.7 installed, then modify line 10 from `./sanity_test.py` to `python2.7 sanity_test.py` . Otherwise, you will need to download Python 2.7

If you fail a test and want to figure out what went wrong, you can go into your tests folder and open the harness corresponding to that test. Right click on your regfile or ALU and choose "view main." You can then see your refile with the inputs provided by the test. Make sure your regfile or ALU behaves the way it ought to!

The sanity tests provided *are not* completely comprehensive, and test some of the main behavior expected from the two circuits. We suggest that you first think through the possible use cases and edge cases for your ALU and RegFile before just running the tests to see if you pass them.

Custom Testing

We have also provided you with a script to help you create tests, called "make_alu_test.py". Take a look at the file called TESTING INSTRUCTIONS to see how you can make your own tests. Basically, you'll want to come up with the values to put inside the different memory units to exercise different behaviors of the ALU. If you run `python make_alu_test.py`, the script will create most of the necessary files for you in the proper format, and will tell you what to do. **Note:** the autograder only works with python 2.7!

Submission & Grading

We will be using Gradescope for submission and grading purposes.

Deliverables

```
regfile.circ
alu.circ
```

Make sure you correctly upload only the deliverable files modified (you may optionally submit a zip file with those files inside).

We will be using our own versions of the `*-harness.circ` files, so you do not need to submit those. In addition, you should not depend on any changes you make to those files. This project will be graded in large part by an autograder. If you would like a regrade, we will give you a chance to request one, but we will also automatically deduct 5% from your final grade, unless it is an error with our test cases.