Lab 6: CPU, Pipelining

Deadline: Monday, March 20, 11:59:59 PM PT

Setup

> You must complete this lab on your local machine (not hive machine). See Lab 0 if you need to set up your local machine again.

In your `labs` directory on your local machine, pull any changes you may have made in past labs:

```
$ git pull origin main
```

Still in your `labs` directory on your local machine, pull the files for this lab with:

```
$ git pull starter main
```

If you run into any `git` errors, please check out the common errors page.

Like Lab 5, all the work in this lab will be done using the digital logic simulation program **Logisim Evolution**.

Some important warnings before you begin:

- Logisim is a GUI program, so it can't easily be used in a headless environment (WSL, SSH, etc.). We recommend running it in a **local environment** with a GUI, Java 9+, and Python 3.6+. If your local system is macOS or Linux you're probably all set. If you're on Windows, use `Git Bash`, which runs on Windows with GUI support.
- Please use the version of Logisim that we distribute, since it is different from other versions on the internet (bugfixes and course-specific stuff)
- Don't move the staff-provided input/output pins; your circuit can't be tested properly if the pins move. If your circuit doesn't pass the tests and you think it is correct, check that your circuit fits in the corresponding harness in `tests/ex#-test.circ`
- Logisim doesn't auto-save your work. Remember to save (and commit) frequently as you work!

Exercise 1: Constructing Immediates

As we have seen in lecture, there are five types of immediates in RISC-V: I-type, S-type, B-type, U-type, and J-type. In this exercise, you will be implementing the **S-type** immediate generator. `ex1.circ` will take as an input a 32-bit RISC-V store instruction. It should output the sign-extended 32-bit immedate value of the instruction. Constructing the immediate

value will require the use of splitters. You can find the summary of how splitters work under the Advanced Logisim Features section in Lab 5.

This exercise will be helpful for Project 3!

1. **Open** `ex1.circ`. This circuit takes in a 32-bit RISC-V **store instruction**.
2. **Modify** the circuit so that it outputs the 32-bit sign-extended immediate value of the instruction. You may assume that the instruction is always a store instruction.

Testing

1. **Open** a local terminal session and navigate to your `lab06` folder.
2. **Run** the provided tests with `python3 test.py`.
   - Your Exercise 1 circuit is run in a test harness circuit (`tests/ex1-test.circ`).
   - Your output is located at `tests/out/ex1-test.out`
   - The reference output is located at `tests/out/ex1-test.ref`
   - In the output file, each column corresponds to an input/output pin on the main circuit, and each row shows a set of inputs and the corresponding outputs the circuit produced.
   - If your circuit output is different, you can check it against the reference output file; the `diff` command may help.

Exercise 2: Constructing the BrUn Control Signal

The `BrUn` control signal is used to tell the branch comparator whether the branch comparison is being performed on signed or unsigned numbers. The following table summarizes the expected value of `BrUn`.

Type of Instruction	`BrUn`	**Notes**
Unsigned branch comparison	`1`	-
Signed branch comparison	`0`	-
`beq` or `bne`	Don't care	The sign of the numbers are not needed when determining if the numbers are equal
Non-branch	Don't care	We do not use the output of the branch comparator for non-branch instructions

If the value is listed as "don't care", this means that you can set `BrUn` to `0` or `1`.

Remember that both of the inputs will either be signed or unsigned. You cannot have the case where one number is signed and one is unsigned. Our hardware does not support comparing an unsigned number to a signed number.

We've provided you with the following hints to help you with your implementation.

<details>
<summary>▼ Which field of the instruction identifies the type of branch?</summary>

funct3
</details>

<details>
<summary>▼ Does the opcode field matter?</summary>

You can use the `opcode` and `funct3` fields to set `BrUn` to `1` whenever the instruction is performing an unsigned branch comparison.
However, the opcode field does not matter when implementing `BrUn` since it is "don't care" for all non-branch instructions. We can simply output a `1` if the `funct3` field corresponds to the `funct3` field of an unsigned branch instruction. It is ok if `BrUn` is `1` for a non-branch instruction because the non-branch instruction does not use the output of the branch comparator.
</details>

Depending on your implementation, constants and comparators will be very helpful for this exercise. The constants can be found in the Wiring library. You can choose the value of the constant in the Properties section in the bottom left window. You can also change the number of bits that are used to represent the constant in the Properties section.

Comparators have two inputs and are used to determine if the first input is less than, equal to, or greater than the second input. They can be found in the Arithmetic library. You can select the size of the inputs and whether the comparison is unsigned or 2's complement in the Properties section in the bottom left window.

This exercise will be helpful for Project 3!

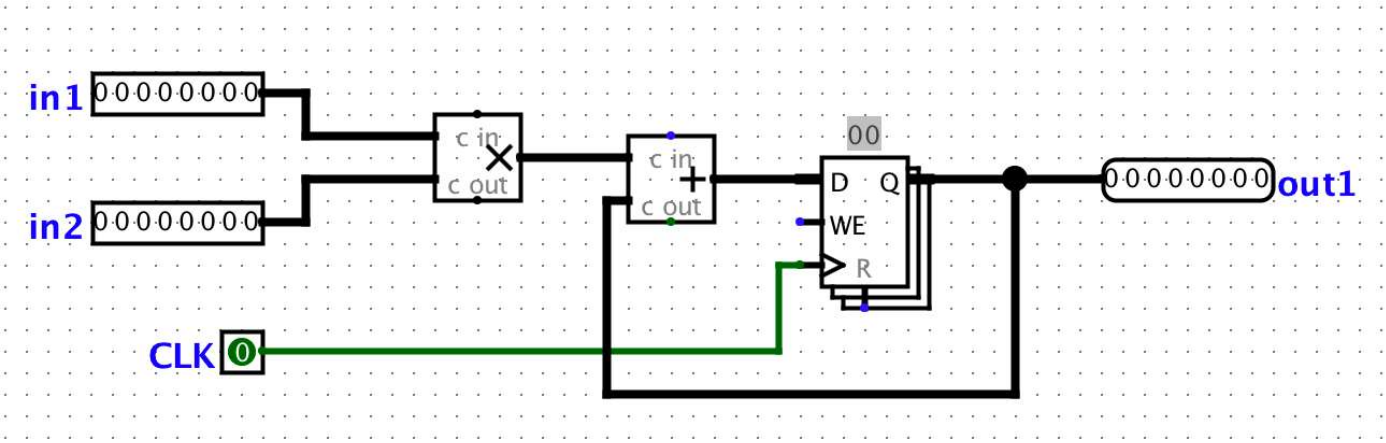
1. **Open** `ex2.circ`. This circuit takes a 32-bit RISC-V instruction as input and outputs the value of the `BrUn` control signal.
2. **Modify** the circuit so that it properly generates the value of `BrUn`.
    - Note: The input instruction may or may not be a branch instruction.

Testing

Refer to exercise 1's testing section, except the output file prefixes are now `ex2-test` instead of `ex1-test`.

Exercise 3: Inefficiencies Everywhere

For this exercise, we can assume that registers initially carry the value zero. We will be using the lab file `ex3.circ`, which should have a subcircuit called `non_pipelined` which looks something like this:



This circuit simply takes two inputs, multiplies them together, and then adds the result to the current state value. For this circuit, let the propagation delay of an adder block be 45ns and the propagation delay of a multiplication block be 60ns. The register has a CLK-to-Q delay of 10ns, setup time of 10ns, and hold time of 5ns. Assume that both inputs receive their data from registers (so the inputs arrive CLK-to-Q after the rising edge).

This exercise will ask you to write down your answers in `ex3_answers.txt`.

- Question 1: What is the length of the critical path of this circuit in **ns**. The answer should be an integer without any units.

Exercise 4: Pipe that Line

We want to improve the performance of this circuit and let it operate at a higher clock rate. In order to accomplish this, we want to have two stages in our pipeline: a multiplication stage and an addition stage, in that order.

To pipeline the circuit, we need a register to hold the intermediate value of the computation between pipeline stages. This is a general theme with pipelining.

In order to check that your pipelining still produces correct outputs, we will consider the outputs from the circuit "correct" if and only if it corresponds to the sequence of outputs the non-pipelined version would emit, but now the circuit will have a leading zero. This leading zero occurs because the second stange of the pipeline is "empty" in the first cycle.

To view the inputs to the circuit and the corresponding outputs for each cycle, take a look at `out/ex4-test.ref`. The output is `00000000` for cycle zero because nothing has moved through the pipeline yet. Cycle zero is essentially just printing out the starting state of the circuit. The output is `00000000` for cycle one because the second stage of the pipeline is "empty" in the first cycle.

We discussed that if an instruction depends on the output of a previous instruction, we need to either insert a pipeline "bubble" (or several) or include forwarding logic to ensure that the output of the first instruction is ready to be an input to the second. As a reminder, a bubble purposely delays an instruction in the pipeline.

<details>
<summary>▼ Why are such "bubbles" unnecessary for this particular circuit?</summary>

For this exercise, ONE instruction consists of BOTH an addition and a multiplication. The addition and multiplication are NOT two separate instructions.

Instead, they are two separate stages in the pipeline, similar to how EX and MEM are two separate stages in a pipeline that operates on the same instruction with EX operating on the instruction before MEM operates on the instruction second.

We do not have any bubbles in this particular circuit because none of the individual instructions depend on each other.
</details>

This exercise will ask you to write down your answers in `ex4_answers.txt`. The question numbers may be different from the step numbers, please be careful!

1. **Open** `ex4.circ` and **pipeline** the circuit found in `ex3.circ`
    - Question 1: What is the critical path of this pipelined circuit in **ns**. The answer should be an integer without any units.
    - Question 2: What is the maximum clock rate for this pipelined circuit in **MHz**. The answer should be a decimal number without any units.

Testing

Refer to exercise 1's testing section, except the output file prefixes are now `ex4-test` instead of `ex1-test`.

Submission

Save, commit, and push your work, then submit to the **Lab 6** assignment on Gradescope.