CS61C Spring 2016 Project 3-2: CPU

TAs: Rebecca Herman, William Huang, Howard Mao

Due Tuesday, March 29th, 2016 @ 11:59 PM

IMPORTANT INFO - PLEASE READ

- You are allowed to use any of Logisim's built-in blocks for all parts of this project.
- **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, Regfile, and CPU have been included in the proj3-starter - just run `make p1` for the ALU and Regfile tests or `make p2` for the CPU test. 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 and you will need to do further testing on your own.
- **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.**

Overview

In this project you will be using Logisim to implement a simple 32-bit two-cycle processor. Throughout the implementation of this project, we'll be making design choices that make it compatible with machine code outputs from MARS and your Project 2! When you're done, you'll be able to run MIPS code through your assembler and linker, and then on your very own CPU :D

In part II, you will complete a 2-stage pipelined processor!

0) Obtaining the Files

We have added the CPU template (cpu.circ) and harness (run.circ), the data memory module (mem.circ), and a basic assembler (assembler.py) to help you test your CPU. Please fetch and merge the changes from the proj3-2 branch of the starter repo. For example, if you have set the `proj3-starter` remote link:

```
cd proj3-XX                  # Go inside the project directory
git fetch proj3-starter
git merge proj3-starter/proj3-2 -m "merge proj3-2 skeleton code"
```

If you do not have the `proj3-starter` remote link from part I, you can run:

```
git remote add proj3-starter https://github.com/cs61c-spring2016/proj3-starter.git
```

If you do have some other inorrect value for the `proj3-starter` remote link, delete it first by running:

```
git remote rm proj3-starter
```

1) Getting Started - Processor

We have provided a skeleton for your processor in `cpu.circ`. Your processor will contain an instance of your ALU and Register File, as well as a memory unit provided in your starter kit. You are also responsible for constructing the entire datapath and control from scratch. Your completed processor should implement the ISA detailed below in the section Instruction Set Architecture (ISA) using a two-cycle pipeline, specified below.

Your processor will get its program from the processor harness `run.circ`. Your processor will output the address of an instruction, and accept the instruction at that address as an input. Inspect `run.circ` to see exactly what's going on. (This same harness will be used to test your final submission, so make sure your CPU fits in the harness before submitting your work!) Your processor has 2 inputs that come from the harness:

INPUT NAME	BIT WIDTH	DESCRIPTION
INSTRUCTION	32	Driven with the instruction at the instruction memory address identified by the FETCH_ADDRESS (see below).
CLOCK	1	The input for the clock. As with the register file, this can be sent into subcircuits (e.g. the CLK input for your register file) 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.).

Your processor must provide 6 outputs to the harness:

OUTPUT NAME	BIT WIDTH	DESCRIPTION
$s0	32	Driven with the contents of $s0. FOR TESTING
$s1	32	Driven with the contents of $s1. FOR TESTING
$s2	32	Driven with the contents of $s2. FOR TESTING
$ra	32	Driven with the contents of $ra. FOR TESTING
$sp	32	Driven with the contents of $sp. FOR TESTING
FETCH_ADDRESS	32	This output is used to select which instruction is presented to the processor on the INSTRUCTION input.

ONE MORE THING: In addition to these inputs and outputs, you also need to have an **LED unit** which lights up to signify signed overflow. This indicator should be wired to the signed overflow port of your ALU. This should be viewable in your main circuit.

Just like in part I, be careful not to move the input or output pins! You should ensure that your processor is correctly loaded by a fresh copy of `run.circ` before you submit. You can download a fresh copy from the starter repo website.

1.5) Getting Started - Memory

The memory unit is already fully implemented for you! Here's a quick summary of its inputs and outputs:

OUTPUT NAME	IN- OR OUT-PUT?	BIT WIDTH	DESCRIPTION
A: ADDR	In	32	Address to read/write to in Memory
D: WRITE DATA	In	32	Value to be written to Memory
En: WRITE ENABLE	In	1	Equal to one on any instructions that write to memory, and zero otherwise
Clock	In	1	Driven by the clock input to cpu.circ
D: READ DATA	Out	32	Driven by the data stored at the specified address.

One important caveat is that the memory is only 64 MB in size, due to the limitations of Logisim's built-in memory block. If you try to read or write addresses greater than or equal to 2^{26}, they will alias to lower addresses. That is, the address 2^{26} + X will appear to be the same as X.

2) Pipelining

Your processor will have a 2-stage pipeline:

1. **Instruction Fetch:** An instruction is fetched from the instruction memory.
2. **Execute:** The instruction is decoded, executed, and committed (written back). This is a combination of the remaining stages of a normal MIPS pipeline.

You should note that data hazards do NOT pose a problem for this design, since all accesses to all sources of data happens only in a single pipeline stage. However, there are still control hazards to deal with. Our ISA does not expose branch delay slots to software. This means that the instruction immediately after a branch or jump is not necessarily executed if the branch is taken. This makes your task a bit more complex. By the time you have figured out that a branch or jump is in the execute stage, you have already accessed the instruction memory and pulled out (possibly) the wrong instruction. You will therefore need to "kill" instructions that are being fetched if the instruction under execution is a jump or a taken branch. Instruction kills for this project MUST be accomplished by MUXing a `nop` into the instruction stream and sending the `nop` into the Execute stage instead of using the fetched instruction. Notice that 0x00000000 is a `nop` instruction; please use this, as it will simplify grading and testing. You should only kill if a branch is taken (do not kill otherwise), but do kill on every type of jump.

Because all of the control and execution is handled in the Execute stage, **your processor should be more or less indistinguishable from a single-cycle implementation, barring the one-cycle startup latency and the branch/jump delays.** However, we will be enforcing the two-pipeline design. If you are unsure about pipelining, it is perfectly fine (maybe even recommended) to first implement a single-cycle processor. This will allow you to first verify that your instruction decoding, control signals, arithmetic operations, and memory accesses are all working properly. From a single-cycle processor you can then split off the Instruction Fetch stage with a few additions and a few logical tweaks. Some things to consider:

- Will the IF and EX stages have the same or different `PC` values?
- Do you need to store the `PC` between the pipelining stages?
- To MUX a `nop` into the instruction stream, do you place it *before* or *after* the instruction register?
- What address should be requested next while the EX stage executes a `nop`? Is this different than normal?

You might also notice a bootstrapping problem here: during the first cycle, the instruction register sitting between the pipeline stages won't contain an instruction loaded from memory. How do we deal with this? It happens that Logisim automatically sets registers to zero on reset; the instruction register will then contain a `nop`. We will allow you to depend on this behavior of Logisim. Remember to go to Simulate `-->` Reset Simulation (Ctrl+R) to reset your processor.

3) Instruction Set Architecture (ISA)

Your CPU will support the instructions listed below. In all of the instructions you recognize from MIPS, the instruction format, opcode, funct, and register numbers should be taken directly from your greensheet. Notice that not all of the native MIPS functions are listed in the CORE INSRUCTION SET section of the greensheet - you may also need to look in the ARITHMETIC CORE INSTRUCTION SET section, and in the OPCODES, BASE CONVERSION, ASCII SYMBOLS table on the back! There will also be an instruction that is foreign to MIPS: specifications for this new instruction will follow the table, as well as clarifications on some other selected instructions.

INSTRUCTION	FORMAT
`Add`	`add $rd, $rs, $rt`
`Add Immediate`	`addi $rt, $rs, immediate`
`Add Immediate Unsigned`	`addiu $rt, $rs, immediate`
`Add Unsigned`	`addu $rd, $rs, $rt`
`And`	`and $rd, $rs, $rt`
`And Immediate`	`andi $rt, $rs, immediate`
`Branch on Equal`	`beq $rs, $rt, label`
`Branch on Not Equal`	`bne $rs, $rt, label`
`Jump`	`j label`
`Jump and Link`	`jal label`
`Jump Register`	`jr $rs`
`Load Upper Immediate`	`lui $rt, immediate`
`Load Word`	`lw $rt, offset($rs)`
`Or`	`or $rd, $rs, $rt`
`Or Immediate`	`ori $rt, $rs, immediate`
`Set Less Than`	`slt $rd, $rs, $rt`
`Set Less Than Immediate`	`slti $rt, $rs, immediate`
`Set Less Than Immediate Unsigned`	`sltiu $rt, $rs, immediate`
`Set Less Than Unsigned`	`sltu $rd, $rs, $rt`
`Shift Left Logical`	`sll $rd, $rt, shamt`
`Shift Right Arithmetic`	`sra $rd, $rt, shamt`
`Shift Right Logical`	`srl $rd, $rt, shamt`
`Store Word`	`sw $rt, offset($rs)`
`Store Word and Increment` **NEW!**	`swinc $rt, offset($rs)`
`Sub`	`sub $rd, $rs, $rt`
`Sub Unsigned`	`subu $rd, $rs, $rt`

Store Word and Increment (swinc)

FORMAT: I
OPERATION (Verilog): M[R[rs] + SignExtImm] = R[rt]; R[rs] = R[rs] + SignExtImm
SignExtImm = { 16{immediate[15]}, immediate}
OPCODE: 0x2c

Jump and Link (jal)

Since we only have a 2 staged pipeline, it does not make sense to store PC + 8 in $ra on a jal instruction! Instead we will store PC + 4

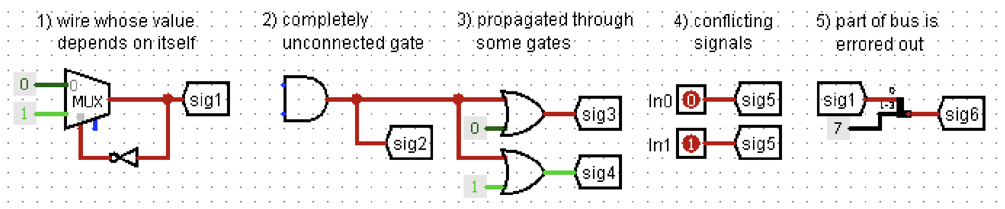
Logisim Notes

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 post to Piazza 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 desireable. 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:


1) wire whose value depends on itself
2) completely unconnected gate
3) propagated through some gates
4) conflicting signals
5) part of bus is errored out
MUX
sig1
sig2
sig3
sig4
sig5
sig6
In0
In1

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.

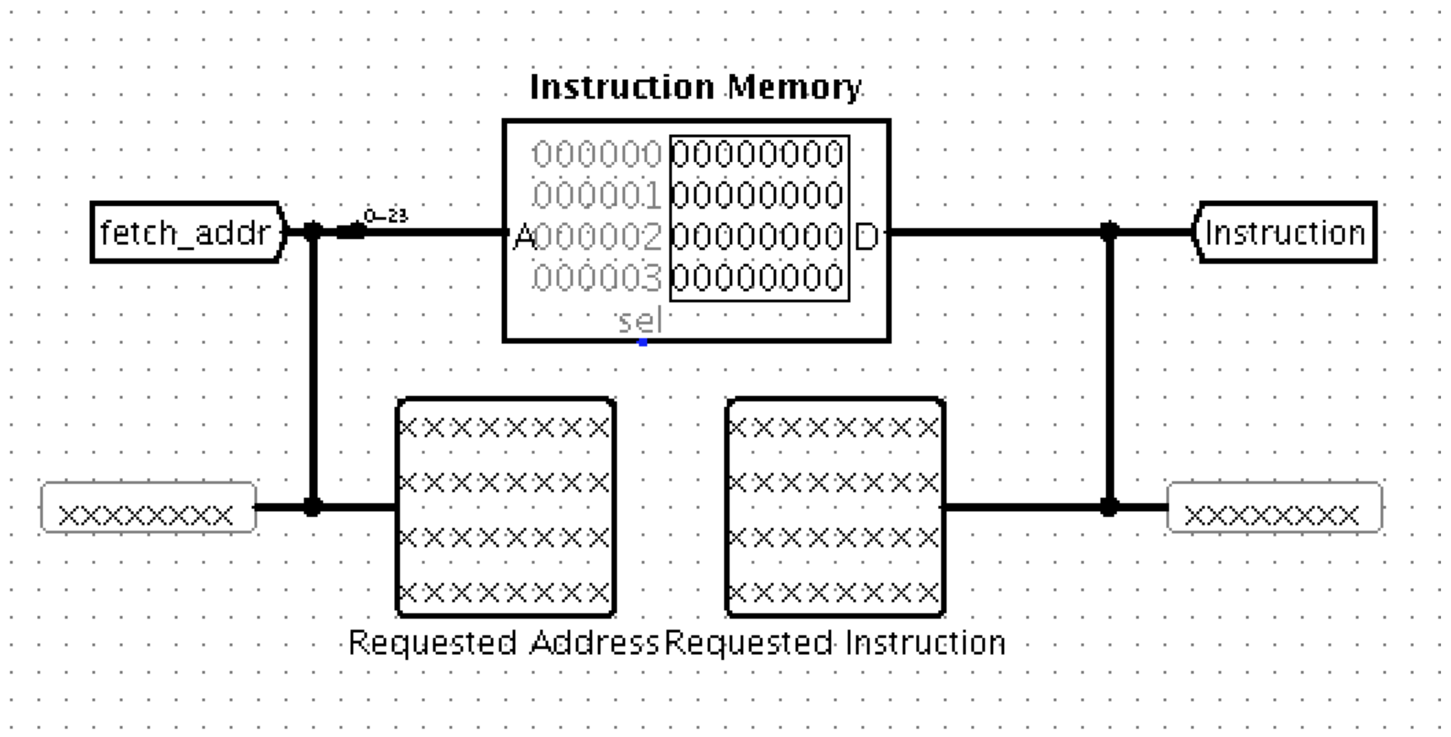
Testing

For part 2, it is somewhat difficult to provide small unit tests such as the ones from part 1 since you are completing the full datapath. As such, the best approach would be to write short MIPS programs and exercise your datapath in different ways. To facilitate this, we have provided you with a rudimentary MIPS assembler that functions similarly to your project 2. To assemble a MIPS file, simply run the assembler with python and pass in your input file as follows:

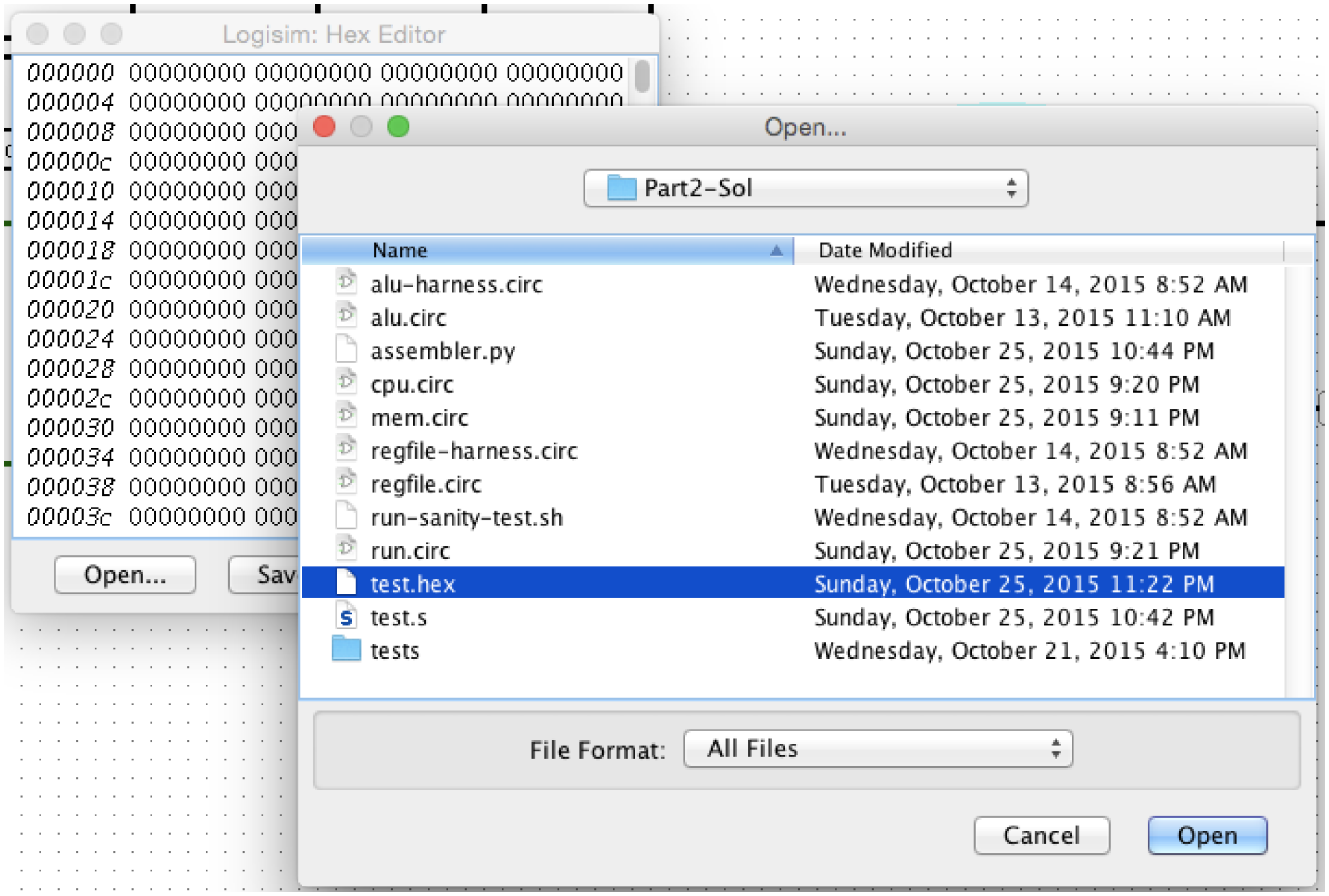
```
vim test.s  # create your assembly file. Remember to only use the instructions provide in the ISA above
python assembler.py test.s # This will generate an output file named test.hex
python assembler.py test.s -v # Same as above, but also provides some verbose output on command line.
```

Note: the assembler has only been tested with python 2.7, so it may be easier to run it remotely off of the `hive*` servers if you haven't set up your python environments.

Once you have generated the machine code, you'll have to load it into the instruction memory unit in `run.circ` and begin execution. To do so, first open `run.circ` and locate the Instruction Memory Unit.



Click on the memory module and then, in the left sidebar, click on the "(Click to edit)" option next to "Contents". This will bring up a hex editor with the option to open a previously created hex file. This is where we load the file outputted by the assembler earlier.



```
000000 00000000 00000000 00000000 00000000
000004 00000000 00000000 00000000 00000000
000008 00000000 000
00000c 00000000 000
000010 00000000 000
000014 00000000 000
000018 00000000 000
00001c 00000000 000
000020 00000000 000
000024 00000000 000
000028 00000000 000
00002c 00000000 000
000030 00000000 000
000034 00000000 000
000038 00000000 000
00003c 00000000 000
```

Name	Date Modified
alu-harness.circ	Wednesday, October 14, 2015 8:52 AM
alu.circ	Tuesday, October 13, 2015 11:10 AM
assembler.py	Sunday, October 25, 2015 10:44 PM
cpu.circ	Sunday, October 25, 2015 9:20 PM
mem.circ	Sunday, October 25, 2015 9:11 PM
regfile-harness.circ	Wednesday, October 14, 2015 8:52 AM
regfile.circ	Tuesday, October 13, 2015 8:56 AM
run-sanity-test.sh	Wednesday, October 14, 2015 8:52 AM
run.circ	Sunday, October 25, 2015 9:21 PM
test.hex	Sunday, October 25, 2015 11:22 PM
test.s	Sunday, October 25, 2015 10:42 PM
tests	Wednesday, October 21, 2015 4:10 PM

Once you've loaded the machine code you can tick the clock manually and watch your CPU execute your program! You can double click on the CPU using the poke tool to take a look at how your datapath is behaving under the given input. You can also fire up MARS at the same time and step through your code there simultaneously. This will allow you to compare the expected behaviour, as seen in MARS, with the behaviour your circuit is exhibiting. Of course, this approach is not valid for the new instructions - for those you'll have to rely solely on your own test assembly code.

Submission

There are **two** steps required to submit proj3-1. Failure to perform both steps will result in loss of credit:

1. First, you must submit using the standard unix submit program on the instructional servers. This assumes that you followed the earlier instructions and did all of your work inside of your `git` repository. To submit, follow these instructions after logging into your -XX class account:

   ```
   cd ~/proj3-XX                              # Or where your shared git repo is
   submit proj3-2
   ```

   Once you type `submit proj3-2`, follow the prompts generated by the submission system. It will tell you when your submission has been successful and you can confirm this by looking at the output of `glookup -t`.

2. Additionally, you must submit proj3-2 to your **shared** Bitbucket repository:

   ```
   cd ~/proj3-XX-YY                           # Or where your shared git repo is
   git add -u
   git commit -m "project 3-2 submission"     # The commit message doesn't have to match exactly.
   git tag "proj3-2-sub"                      # The tag MUST be "proj3-2-sub". Failure to do so will result in loss of credit.
   git push origin proj3-2-sub                # This tells git to push the commit tagged proj3-2-sub
   ```

Resubmitting

If you need to re-submit, you can follow the same set of steps that you would if you were submitting for the first time, but you will need to use the `-f` flag to tag and push to Bitbucket:

```
# Do everything as above until you get to tagging
git tag -f "proj3-2-sub"
git push -f origin proj3-2-sub
```

Note that in general, force pushes should be used with caution. They will overwrite your remote repository with information from your local copy. As long as you have not damaged your local copy in any way, this will be fine.

Deliverables

`cpu.circ`

We will be using our own versions of the `*-harness.circ` and `run.circ` files, so you do not need to submit those. In addition, you should not depend on any changes you make to those files.

Grading

This project will be graded in large part by an autograder. Readers will also glance at your circuits. If some of your tests fail the readers will look to see if there is a simple wiring problem. If they can find one, they will give you the new score from the autograder minus a deduction based on the severity of the wiring problem. For this reason, neatness is a small part of your grade - please try to make your circuits neat and readable.