CPU Project (Part 2): CPU Design

Acknowledgement: UC Berkeley 61C

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!
- **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 16-bit two-cycle processor based on RPIS. This project is meant to give you a better understanding of the actual MIPS datapath. In fact, after this project you would have everything you needed to know in order to build a MIPS CPU in Logisim that can understand the output of assembler/linker phase.

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 download the starter package here.

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 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	16	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	16	Driven with the contents of $s0. FOR TESTING
$s1	16	Driven with the contents of $s1. FOR TESTING
$s2	16	Driven with the contents of $s2. FOR TESTING
$ra	16	Driven with the contents of $ra. FOR TESTING
$sp	16	Driven with the contents of $sp. FOR TESTING
FETCH_ADDRESS	16	This output is used to select which instruction is presented to the processor on the INSTRUCTION input.

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.

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	16	Address to read/write to in Memory
D: WRITE DATA	In	16	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	16	Driven by the data stored at the specified address.

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 0x0000 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.

2.5) Controls

You can probably guess that control signals will play a very large part in this project. Figuring out all of the control signals may seem intimidating. Remember that there is not a definitive set of control signals--walk through the datapath with different types of instructions, and when you see a mux or other component think about what selector/enable value you will need for that instruction.

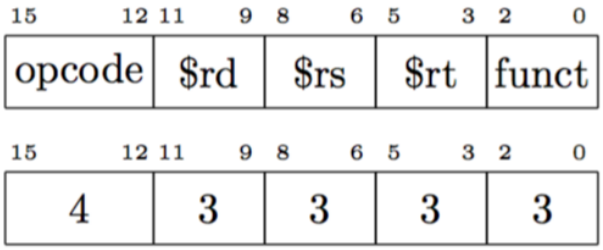
Additionally, implementing the control signals can be done in many ways, including implementing the corresponding truth tables and using comparators. Since you are welcome to use any built-in Logisim circuits, we suggest using whichever component makes the most sense to you, whether performing logical operations on instruction bits and/or comparing fields to certain values.

3) RPIS: The Instruction Set Architecture

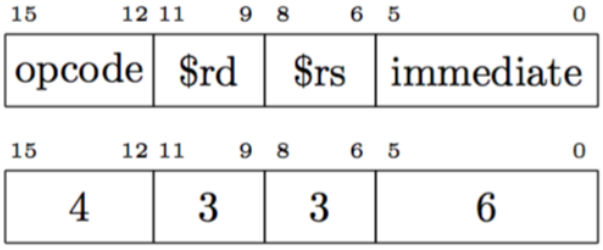
Your CPU will support the instructions listed below. Since you can't use the Green Sheet to reference for these instructions, the same information has been reproduced for RPIS below. There are some important differences to understand between MIPS and RPIS, and they are explained below (please make sure that you understand the new format of instructions before you start implementing it!)

Instruction Formats

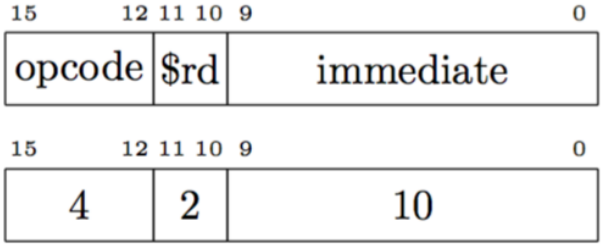
R-Type



I-Type



U-Type



The Instructions

OPCODE /FUNCT	TYPE	INSTRUCTION	FORMAT	BEHAVIOR
0x0	R	Shift Left Logical	sll $rd, $rs, $rt	R[rd] = R[rs] << R[rt][4:0]
0x1	R	Shift Right Logical	srl $rd, $rs, $rt	R[rd] = R[rs] >>> R[rt][4:0]
0x2	R	Add	add $rd, $rs, $rt	R[rd] = R[rs] + R[rt]
0x3	R	And	and $rd, $rs, $rt	R[rd] = R[rs] & R[rt]
0x4	R	Or	or $rd, $rs, $rt	R[rd] = R[rs] | R[rt]
0x5	R	Exclusive Or	xor $rd, $rs, $rt	R[rd] = R[rs] ^ R[rt]
0x6	R	Set Less Than (Signed)	slt $rd, $rs, $rt	R[rd] = (R[rs] < R[rt]) ? 1 : 0
0x7	R	Multiply (Unsigned)	mult $rs, $rt	R[HI] = (R[rs] * R[rt])[31:16]; R[LO] = (R[rs] * R[rt])[15:0];
0x1	U	Load Upper Immediate	lui imm	R[LO] = {imm, 0b000000} = imm << 6
0x2	U	Jump/Jump and Link	j $rd, label	PC = JumpAddr; R[rd] = PC + 2; JumpAddr = {PC[15:10], imm};
0x3	I	Jump Register /Jump and Link Register	jr $rd, $rs, imm	PC = R[rs] + SignExt(imm); R[rd] = PC + 2;
0x4	I	Branch on Equal	beq $rs, $rd label	if (R[rs] == R[rd]) PC = PC + SignExt(imm)
0x5	I	Branch on Not Equal	bne $rs, $rd label	if (R[rs] != R[rd]) PC = PC + SignExt(imm)
0x6	I	Add Immediate	addi $rd, $rs imm	R[rd] = R[rs] + SignExt(imm)
0x7	I	Set Less Than Immediate	slti $rd, $rs imm	R[rd] = R[rs] < SignExt(imm) ? 1 : 0
0x8	I	And Immediate	andi $rd, $rs imm	R[rd] = R[rs] & ZeroExt(imm)
0x9	I	Or Immediate	ori $rd, $rs imm	R[rd] = R[rs] | ZeroExt(imm)
0xa	I	Load HalfWord	lh $rd, offset($rs)	R[rd] = M[addr]; addr = R[rs] + SignExt(offset)
0xb	I	Store HalfWord	sh $rd, offset($rs)	M[addr] = R[rd]; addr = R[rs] + SignExt(offset)
0xc	I	Move From High	mfhi $rd	R[rd] = R[HI]
0xd	I	Move From Low	mflo $rd	R[rd] = R[LO]

Notable Differences from MIPS / Clarifications

Jumps and Linking

In RPIS, there are only two jump instructions, jump and jump register. For jump, the jump address is calculated by taking the 10 bit immediate (as it is a U-type instruction), and prepending the 6 most significant bits from the PC. In other words, using "," to indicate concatenation, JumpAddr = {PC[15:10], immediate}. For example, if the immediate was 0x1a7 = 0b0110100111 and the PC was 0xabcd = 0b1010101111001101, then JumpAddr = {101010, 0110100111} = 0b1010100110100111 = 0xa9a7.

For jump register, since it is now an I-type instruction the immediate is also used to calculate the address to jump to. Specifically, the PC will be set to the address in $rs plus immediate bytes, or PC = JumpAddr = R[rs] + immediate. If you just wanted to jump to the exact address in $rs, as with jump register in MIPS, then the immediate would just be 0.

Instead of having explicit link versions of these jump instructions, linking is done by using the register specified by $rd as the link register (in MIPS the link register would automatically be specified as $ra). That is, if $rd is say register $s0, then the address of the next instruction after the current PC (before the jump) would be saved in $s0 (if you wanted to have the same behavior as in MIPS, then you would always have $rs = $ra = 0b001). However, note that if $rd is the zero register, then the next instruction after the PC is not saved at all (and no linking is done), as the zero register cannot be written to.

Therefore, to execute the same behavior as `jalr $s0` in MIPS, you would call `jr $ra $s0 0`, such that R[ra] = PC + 2 and then PC = R[s0]. You may have noticed that the $rd field for U-type instructions is only 2 bits, which means that for jump, our $rd/link register can only be registers 0-3, or $0, $ra, $s0, and $s1.

Branches

In RPIS, the immediate/offset in branch instructions is interpreted as the number of *bytes* from PC to the label address. This is different from the use of word offset in MIPS, and you may notice that the LSB bit of the immediate/offset will always be 0, but this behavior should be more straightforward to implement along with `jr`.

Load Upper Immediate

Since our immediate for I-Type instructions is only 6 bits now, we would not be able to load a full 16 bit immediate if `lui` could only load 6 bits. Therefore, in RPIS, `lui` is now a U-Type instruction with a 10 bit immediate, where the 10 bit immediate value is loaded into the special $LO register (and `mflo` can then be used to load from the $LO register). The following sequence of instructions would then load a 16 bit immediate (e.g. 0xABCD = 0b1010,1011,1100,1101) into $s0:

```
lui 0b1010101111
mflo $s0
ori $s0 $s0 0b001101
```

For `lui`, the U-Type $rd field is unused and can be any value.

Multiplication

As in MIPS, the multiply instruction will store the upper bits of the product in $HI and the lower bits in $LO. Specifically, for RPIS, the upper 16 bits of the product will be stored in $HI and the lower 16 bits will be in $LO. `mfhi` and `mflo` can then be used in order to access the values of $HI and $LO, respectively.

Set Less Than

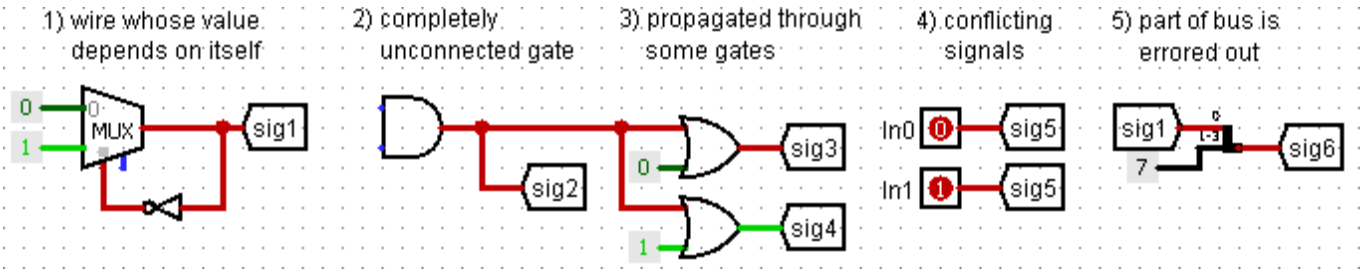
In RPIS, set less than is equivalent to `slt` in MIPS, where the values of $rs and $rd are interpreted as signed integers (you already implemented this in your ALU without any special handling).

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 16-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:



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.

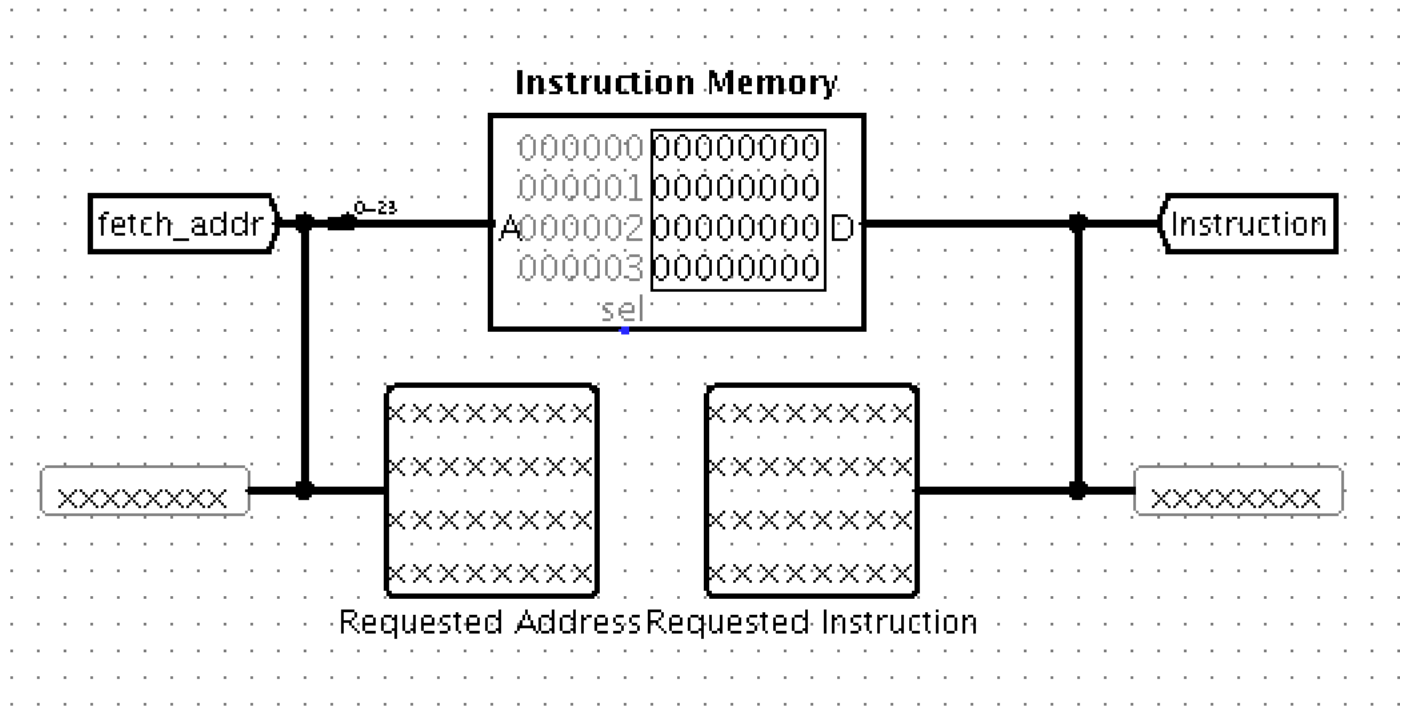
Testing

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 RPIS programs and exercise your datapath in different ways. To facilitate this, we have provided you with a rudimentary RPIS assembler. To assemble a RPIS file, simply run the assembler with python and pass in your input file as follows:

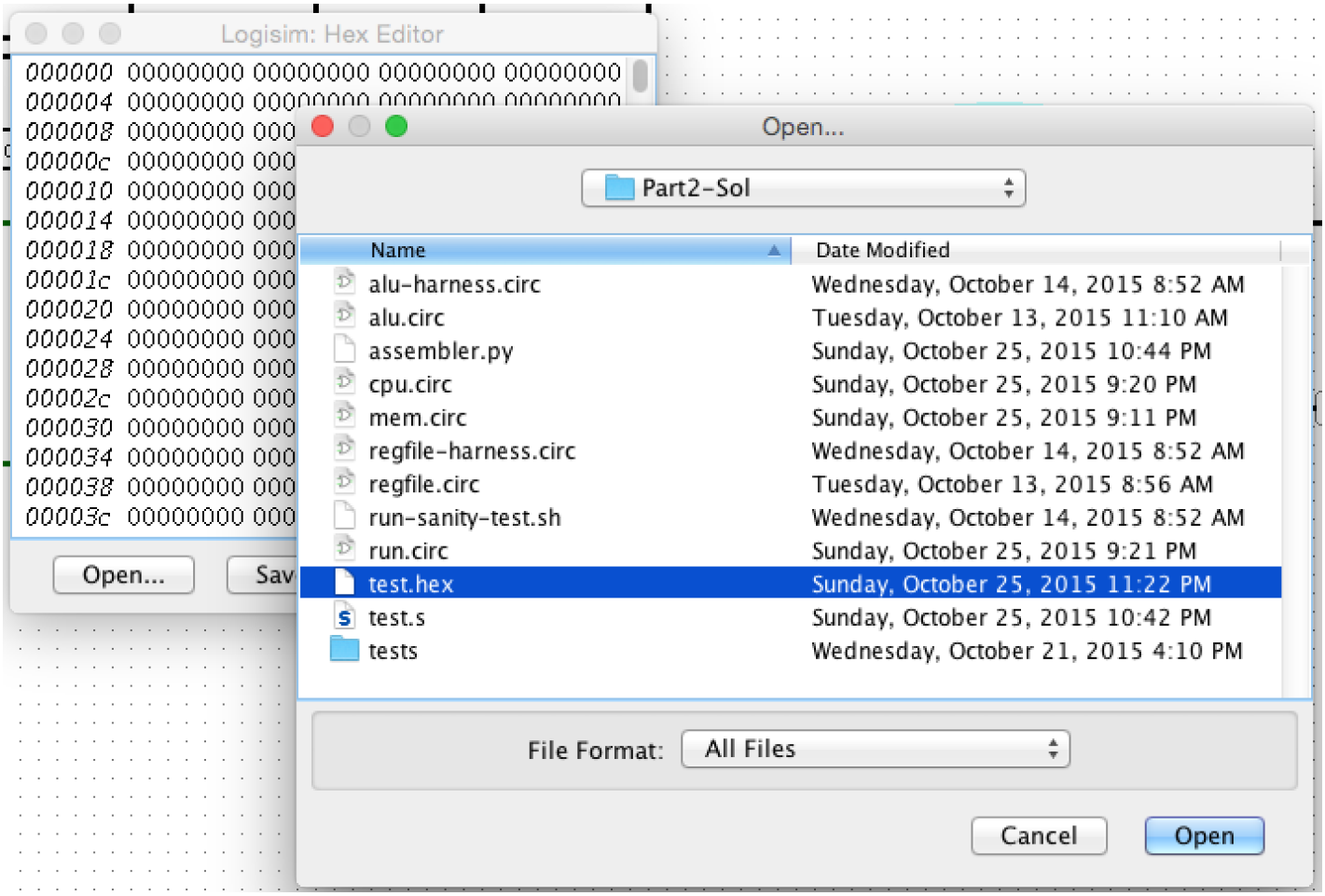
```
vim test.s  # create your assembly file. Remember to only use the instructions provide in
the ISA above
python assembler.py test.s # This will generate an output file named test.hex
python assembler.py test.s -v # Same as above, but also provides some verbose output on
command line.
```

Note: the assembler has only been tested with python 2.7

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.



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. Unfortunately, there is no equivalent MARS program for RPIS, so you will need to think through the behavior and execution of each instruction yourself.

Lastly, you may be wondering if your implementation will rely on your ALU and RegFile from part 1. You will be using your own implementations of the ALU and RegFile as you construct your datapath; however, for grading, we will test your CPU using both your versions of the ALU and RegFile as well as a staff version and take the maximum score of the two.

Submission & Grading

We will be using Gradescope for submission and grading purposes.

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. This project will be graded in large part by an autograder. If some of your tests fail, we will try to 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.