

CS2630: Computer Organization

Project 2, part 2

Single-cycle MIPS processor with I/O

Table of Contents

GOALS FOR THIS ASSIGNMENT	2
INTRODUCTION	2
READING THIS DOCUMENT.....	2
GETTING STARTED	3
THE PROCESSOR.....	5
INSTRUCTION DETAILS	6
CLZ INSTRUCTION	6
WHAT YOU MUST IMPLEMENT.....	6
TIPS ON BUILDING THE CONTROL UNIT	6
HOW YOU MUST TEST.....	7
TESTING TIPS.....	7
ASSEMBLING AND RUNNING NEW PROGRAMS.....	8
INPUT/OUTPUT.....	13
WHAT YOU MUST DO	14
REQUIREMENTS AND GRADING.....	14
RUBRIC.....	14
ADDITIONAL REQUIREMENTS.....	15
SUBMISSION CHECKLIST	15
RECOMMENDED APPROACH TO FINISHING THE PROJECT.....	16

TIPS	17
TEAMWORK TIPS	17
WHERE TO GET HELP	17
 ACADEMIC HONESTY.....	 18
 ACKNOWLEDGEMENTS	 18

Goals for this assignment

- Design and implement a substantial digital system
- Add new instructions to the datapath and control
- Use robust testing methodology in digital logic design
- Learn how to load binary code from the assembler into the instruction memory
- Create MIPS programs that adequately test the processor

Introduction

In project 2-1 you built two major components of a MIPS processor, and in project 2-2 you will build the rest of a processor, as well as IO to make it useful. As in part 1, we provide the top-level skeleton file and test circuits, and you will provide the implementation and additional tests.

Reading this document

There is a lot here. I recommend 3 experts: datapath, control, and testing. At some point you should read the whole document, but here's **what you should read first** if you want to get started fast but still be thorough.

- **Datapath expert:**
 1. *Getting started*, with emphasis on 3, 4a, and 4b
 2. *The Processor* including *What you must implement* and excluding *Tips on building the control unit*
 3. https://uiowa.instructure.com/courses/87896/assignments/760593?module_id=2174831 if you haven't yet
 4. *Recommended approach to finishing the project*
 5. *Teamwork Tips*
 6. *Where to get help*
- **Control unit expert:**
 1. *Getting started*, skimming what is after step 2
 2. *The Processor* including *What you must implement* and emphasizing *Tips on building the control unit*
 3. https://uiowa.instructure.com/courses/87896/assignments/760593?module_id=2174831 if you haven't yet

4. *Recommended approach to finishing the project*
 5. *Teamwork Tips*
 6. *Where to get help*
- **Testing expert:**
 1. *Getting Started*, emphasis on getting it to work with either your alu/regfile or the provided ones
 2. *The Processor*, emphasis on *How you must test* and *Testing tips*.
 3. https://uiowa.instructure.com/courses/87896/assignments/760593?module_id=2174831 if you haven't yet
 4. *Assembling and running new programs*, then go through those steps again for the func_test.s file.
 5. *Recommended approach to finishing the project*
 6. *Tips*
 7. *Teamwork Tips*
 8. *Where to get help*

That leaves out Input/Output, but we recommend you work on that second.

Getting started

1. Download the starter code from <https://github.com/bmyerz/proj3-starter/archive/proj2-part2.zip>

(or, if you are using git, you can instead clone <https://github.com/bmyerz/proj3-starter.git> and then switch to the branch proj2-part2)

2. Try running the tests

Use the same method for running Linux commands that you used in part 1 of the project.

- i. Run the tests

`make p2sc`

You should see output like

```
cp alu.circ regfile.circ mem.circ cpu.circ tests
cd tests && python ./test.py p2sc | tee ../TEST_LOG
Error loading circuit file: func_test.circ
Testing files...
Error in formatting of Logisim output (check ./CPU-starter_kit_test.circ):
    you have a non-integer in this list: ['xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx', 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx', 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'00000000000000000000000000000000', 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx']
FAILED test: CPU starter test (Error in the test)
Error in formatting of Logisim output (check ./func_test.circ):
```

```

    you have a non-integer in this list: ['']. If this is logisim output, are you sure you
    have a circ file for this test?
    FAILED test: func test (Error in the test)
    Passed 0/2 tests

```

The x's indicate that those bits are disconnected.

3. Copy your regfile.circ and alu.circ solutions from Project 2-1 into the new directory.
4. Alternatively, if your project 2-1 solution didn't fully work and you want to use a working alternative, we provide some with instructions below.

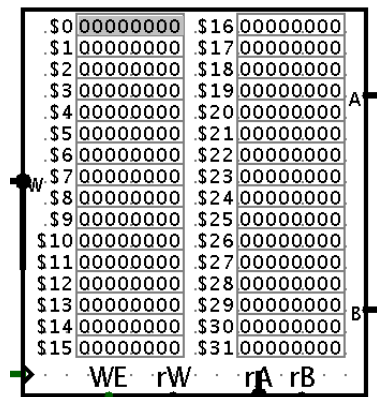
a. To use the register file replacement

download regfile.circ and cs3410.jar from:

<https://uiowa.instructure.com/courses/87896/files/folder/project2>

You must put regfile.circ in the base directory of your project (as it was in project 2-1) and a copy of the cs3410.jar file must go in **both** the base directory and tests/ directory.

The advantage to using the replacement regfile.circ is that it uses a fancy register file that shows the values of the registers, which may make debugging a bit easier.



b. To use the ALU replacement

download alu.circ and ALU.jar from:

<https://uiowa.instructure.com/courses/87896/files/folder/project2>

You must put alu.circ in the base directory of your project (as it was in project 2-1) and a copy of the ALU.jar file must go in **both** the base directory and the tests/ directory.

5. You can check if you copied the ALU and register file (whether using yours or ours) properly into your project 2 folder by running the part 1 tests and seeing that they pass.

make p1

```

cp alu.circ regfile.circ tests
cd tests && python ./test.py p1 | tee ../TEST_LOG
Testing files...
    PASSED test: ALU add (with overflow) test
    PASSED test: ALU arithmetic right shift test
    PASSED test: RegFile read/write test
    PASSED test: RegFile $zero test
Passed 4/4 tests

```

- When you start editing the cpu.circ file, you'll want to include the ALU and register file components by choosing Project > Load Library > Logisim Library... and choosing the alu.circ or regfile.circ in the base directory of your project folder.

The processor

Your team's task is to design, implement, and test a single-cycle MIPS processor.

The processor must support a specific subset of instructions from the 32-bit MIPS instruction set architecture. That subset is

Instruction
sll
srl
sra
add
addu
addiu
addi
jal
jr
j
slt
sltu
sltiu
slti
and
or
andi
ori
lui
lw
sw
beq

	bne
	clz

The specification of the instructions is exactly the one in the “MIPS reference card” or “Human-friendly MIPS reference card” on (find in Modules on ICON) except for the following differences.

Instruction details

jal – Jump and Link – The reference sheet might say to store PC+8 into \$ra, but you must instead use PC+4. (Our architecture will not assume a jump delay slot; neither does MARS by default).


clz instruction

clz – Count leading zeroes – this instruction is not listed in the MIPS reference sheet, but MARS supports it as a core instruction.

The clz is a MIPS instruction that was not covered in the class, but you can understand its behavior and bit encoding by looking at the help in MARS. HINT: you can implement a clz circuit with a combination of some components available in the Arithmetic library.

You must use MARS to reverse engineer the bit encoding of this instruction. Try assembling a program with clz, with different values of the argument registers. It is actually not R-type, I-type, or J-type but a fourth type of instruction that is similar to R-type.

What you must implement

You must modify `cpu.circ` to implement the CPU. Do not modify or move the inputs and outputs. You may use sub-circuits in your implementation as long as  `main` remains the top level circuit of `cpu.circ`. You may use any *built-in* Logisim components.

For your Data Memory, you can use `mem.circ`. That module can read or write one memory location on every cycle. When `Write_En=1`, the memory will write data `Write_Data` to the location given by `Address` on the next rising edge of the clock, and when `Write_En=0` the `Read_Data` port will have the value at the location given by `Address`.

Tips on building the control unit

Building a control unit can be very complex and error prone due to the large number of input and output bits, so you should try to reduce complexity where possible. Specifically,

- Rely on a logic analyzer, such as Logisim's logic analyzer tool (found at Project | Analyze circuit). It will allow you to input a function as a truth table and automatically generate the circuit. Note that the logic analyzer requires 1-bit inputs, so you'll have to split multi-bit wires into individual bits.
- Use "don't cares" to simplify the logic (the logic analyzer represents them as X's)
- Consider separating the logic that computes the basic 1-bit control signals from the logic that computes the ALU's switch input.
- Keep in mind that all control signals (except for ALU's Switch input) are calculated from solely the opcode. A 6-bit truth table using the logic analyzer is reasonable to construct.
- Keep in mind that the ALU's Switch input is calculated from both opcode and funct fields. Don't try to build a single truth table of 12 inputs, it is far too large and has too many redundancies.

How you must test

1. Run the tests with the command `make p2sc`.
2. To ensure you pass the autograder, you **must test your CPU beyond the given tests**.

Adding new tests is similar to Project 2-1, except:

- the sample test harness to copy is `tests/CPU-starter_kit_test.circ` instead of `alu-harness.circ` and `regfile-harness.circ`.
- instead of loading the test inputs into RAMs, you will load the instruction memory (it is a ROM) with an assembled program.

See the section “Assembling and running new programs” for a step-by-step guide.

See the `get_test_format` function in `tests/decode_out.py` for the format of the output of the "cpu" tests. The first list is the headers and the second list is the bit widths.

```
['$s0 Value', '$s1 Value', '$s2 Value', '$ra Value', '$sp Value', 'Time Step', 'Fetch Addr', 'Instruction'], [32,32,32,32,32,8,32,32]
```

Testing tips

- Since there is some effort to adding a new test, try to balance keeping the tests simple while including multiple instructions
- Make sure to check different cases, such as branch, not branch, branch forward, branch backward
- Having to give the expected values of the 5 registers, fetch Address, and instruction bits on every single clock cycle can be overkill for more complex tests. To help you, we've included different types of tests that check only some of the outputs.

Type	Checks outputs	Recommendation
cpu	'\$s0 Value', '\$s1 Value', '\$s2 Value', '\$ra Value', '\$sp Value', 'Time Step', 'Fetch Addr', 'Instruction'	use for short/simple tests, where you want to check everything on every cycle
cpu-lite	'\$s0 Value', '\$s1 Value', '\$s2 Value', '\$ra Value', '\$sp Value', 'Time Step'	use for tests where you don't want to have to check fetch address and instruction
cpu-end	'\$s0 Value', '\$s1 Value', '\$s2 Value', '\$sp Value'	use for tests where you only want to check the state of some registers when they change.

Note: To understand why cpu-end tests do not check outputs every cycle, rather only when the registers change, it is helpful to know that Logisim only prints a new line of output when one of the output values change. For cpu and cpu-lite, the inclusion of "Time Step" ensures that a line gets printed every cycle.

You specify the test Type by making the last argument to TestCase be "cpu", "cpu-lite", or "cpu-end" in your tests.py file.

We have provided an example "cpu-end" test called func_test. To use it, you need follow the directions described in the next section. The directions describe a different test, but you will learn the steps you need to create a test harness called func_test.circ and loading its instruction memory with the binary code generated by assembling func_test.mars.s.

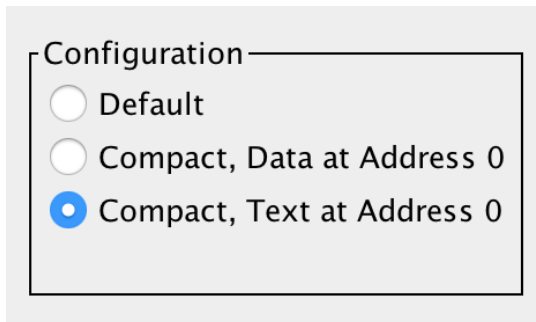
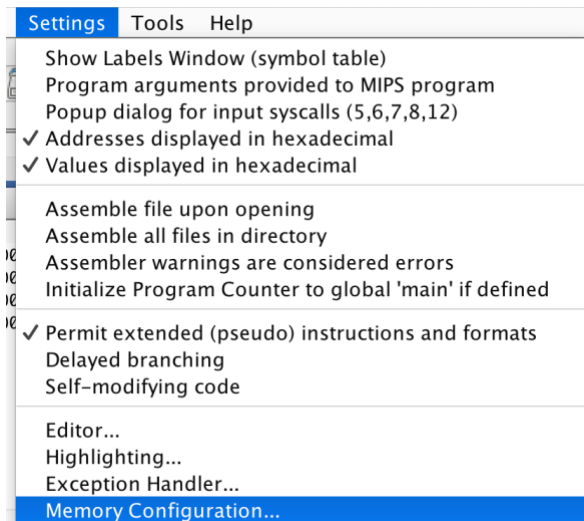
Do not try to pass func_test immediately! Think of func_test as one of the LAST tests you should pass after you've written and passed easier tests.

Add your expected outputs as a list of lists in the Python file as described in Project 2-1.

Assembling and running new programs

The project kit comes with a copy of Mars (mars.jar) so that you can assemble MIPS programs in the format required for the instruction memory. What follows is the workflow that we recommend for writing MIPS programs and running them on your processor.

1. Edit your MIPS program in MARS (as you did in HW2,3).
You should set the following "Memory Configuration...", to tell MARS to assemble the addresses the same way our command line assembler does (.text starts at address 0x00000000 and .data starts at address 0x00002000)



2. Test and debug your program in MARS (as you did in HW2,3).
3. Save your MIPS program to a file. We'll assume the name "foo.s" for these directions, but you should name the file appropriately.
4. When you are ready to run your program on the MIPS processor, you will use the assembler provided with the project kit.

Make sure you know the file path of your MIPS file. It's easiest if you just save it to the proj2-part2 folder.

- i. Change directories to path of your proj2-part2 folder

`cd /path/to/proj2-part2` (/path/to should be the actual file path)

- ii. Double check that the MIPS program is in your directory by running `ls`.

`Makefile`

`TEST_LOG`

`alu-harness.circ`

`alu.circ`

`cpu.circ`

`example_IO_controller.circ`

foo.s

mars-assem.sh
mars.jar
mem.circ
regfile-harness.circ
regfile.circ
run.circ
tests
text-out.hex

- iii. Run the assembler on your MIPS program

```
./mars-assem.sh foo.s
```

If your assembly file didn't have a .data section you might see a message, but it is just a warning.

```
This segment has not been written to, there is nothing to dump.  
cat: data_t.hex: No such file or directory  
rm: data_t.hex: No such file or directory
```

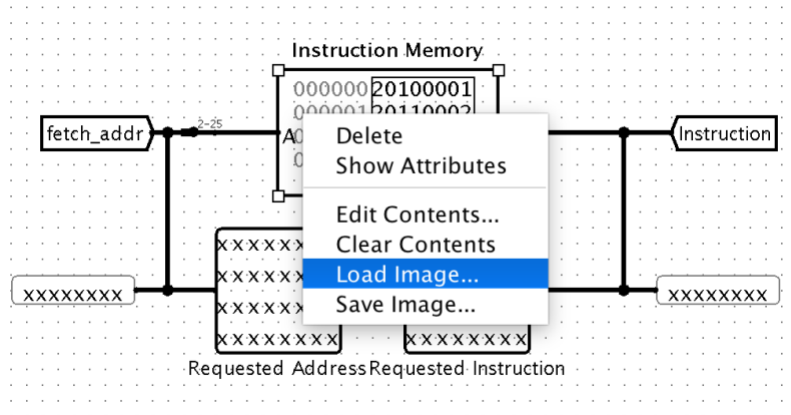
- iv. List the files in the folder again (by running `ls`) to check that there was output.

Makefile
TEST_LOG
alu-harness.circ
alu.circ
cpu.circ
example_IO_controller.circ
foo.s
foo.s.data.hex
foo.s.text.hex
mars-assem.sh
mars.jar
mem.circ
regfile-harness.circ
regfile.circ
run.circ
tests
text-out.hex

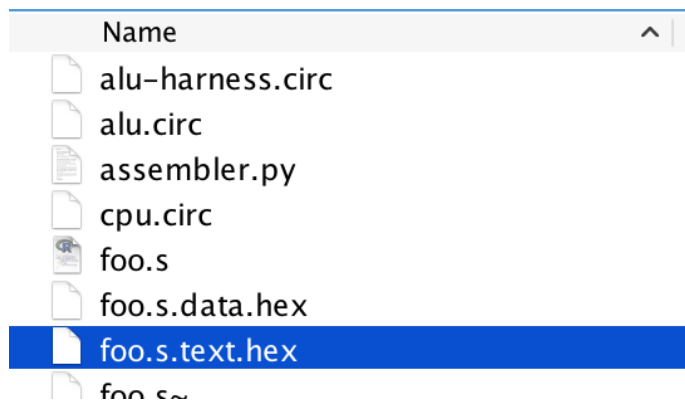
You should see a .text.hex file, which contains the text segment. If you had a .data section, you should also see a .data.hex file, which contains data memory contents **up to and including** the .data segment.

5. Now you can load the program into your processor in Logisim. **Make a copy of CPU-starter_kit_test.circ** and then open that new file in Logisim.

- i. Load the instruction memory by right-clicking the Instruction Memory ROM and choosing Load Image...

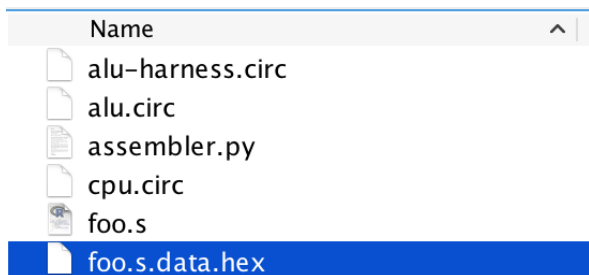
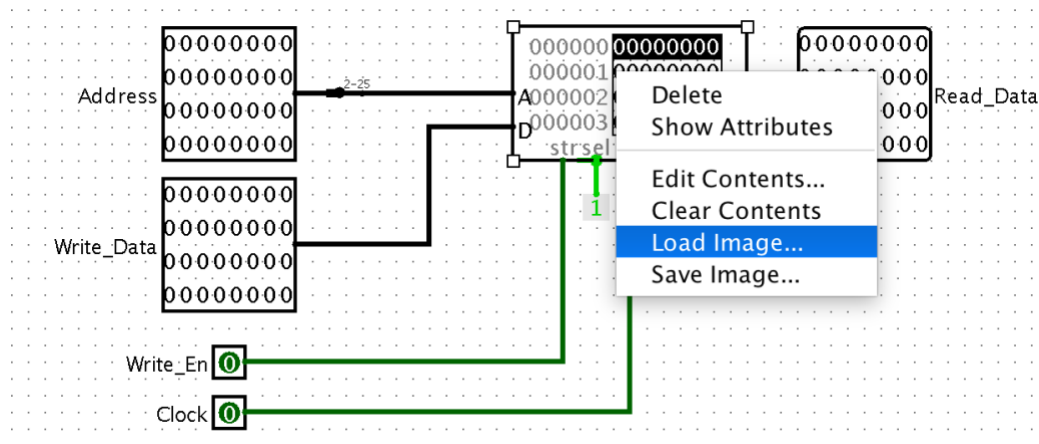


Navigate to the foo.s.text.hex file



- ii. Load the data memory (**OPTIONAL; only need to do this step if your MIPS program has a .data section**). The data memory implementation is provided to you in mem.circ, which you should use as a “Logisim library...” in your CPU.

right-click the RAM | Load image | navigate to foo.s.data.hex



You can double-check that your data was loaded into the expected address in memory by right clicking the RAM > Edit Contents... > and scrolling down to the row for 002000 to see the data.

- iii. Make sure to Save the circuit file containing the instruction memory so that you don't have to load the program again (just like you had many test harnesses in Part 1, you will just have another copy of CPU-starter_kit_test.circ for each test program).

Note that you *will* have to load your *data memory* each time you change programs or "Reset Simulation". Logisim applies the reset to RAMs but not ROMs. **Therefore, we suggest making most tests *not* have a .data section so that you don't have the inconvenience of loading the RAM.**

6. Now you can simulate your CPU by ticking the clock. Notice that once the instruction memory gets to instructions 0x00000000, your processor should just be executing NOOPs until you stop the simulation.

Input/output

You now have a processor that executes real MIPS programs! Now it is time to make it more interesting by including some IO devices. We will use a methodology for IO called Memory mapped IO (MMIO). You already have some experience with it from the drawing application in HW 3.

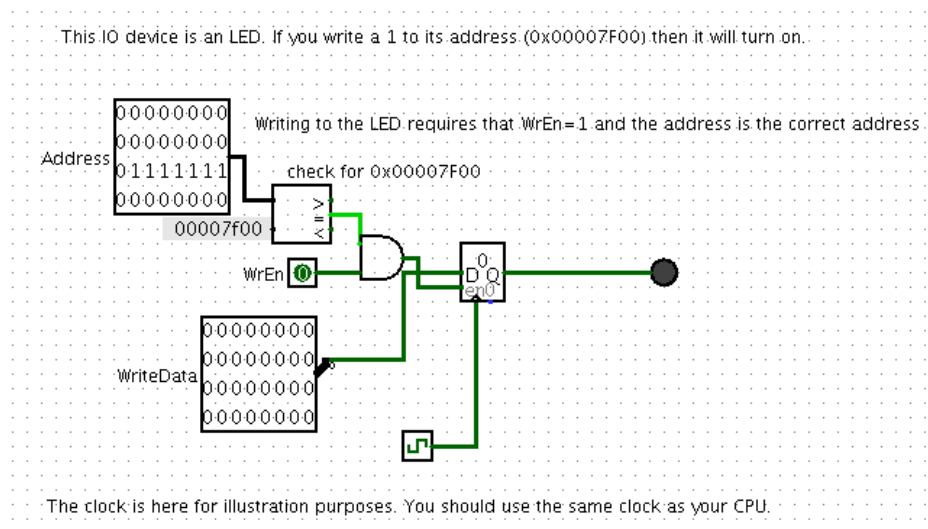
The way MMIO works is that some range of addresses is reserved for controlling IO devices rather than accessing memory.

If you look back at MARS's Memory Configuration (Settings > Memory Configuration), you'll see that addresses starting at 0x00007f00 are reserved for MMIO.

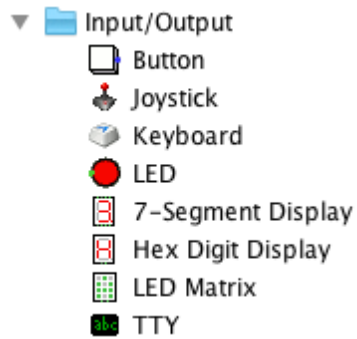
MIPS Memory Configuration	
0x00007fff	memory map limit address
0x00007fff	kernel space high address
0x00007f00	MMIO base address

An output device will take in the same Address/MemWrite/WriteData signals as the data memory, except it will *listen* for its range of addresses (0x00007F00 and above). When MemWrite=1 and the Address is in range, the output device will be written to.

We've provided an example module in `example_IO_controller.circ`. It is a single LED that can be turned on by writing a 1 to address 0x00007F00 and turned off by writing a 0 to address 0x00007F00.



There are more IO devices besides the LED available in Logisim's Input/Output folder.



You can experiment with how they work. Post to Piazza discussion if you have trouble understanding how to control one of them.

What you must do

Your task is to implement an interesting application that uses some input and/or output device.

- i. Build an IO controller for the device that you want to use and attach it to the appropriate signals in your CPU. **Your IO device should be placed in the top-level of `cpu.circ`.** The device should be more than the single LED example (i.e., *at least* two LEDs but other devices are encouraged). **Write as short of a MIPS test program as possible** that will read or write (depending on if you picked an input or output) the device. Call the test program `iotest.s` and the test harness `iotest.circ`.
- ii. Write a second, more **interesting MIPS program**, that uses your IO device(s). Maybe it is a timer, an animation, a calculator, a screen and keyboard, a game. The sky is the limit here! Include a file `io_readme.txt` that briefly describes how your application works. Also, tell us which Tick Frequency the application works best at.
- iii. Pat yourself on the back. You've built a working and *useful* computer! Show off your work to others. (just don't share copies of your files)

Requirements and grading

Rubric

You can also see ICON for the rubric that we'll use to grade

- 25 points – the processor passes the autograder tests (we have hidden tests that cover all required functionality)
- 10 points - the tests you submit demonstrate coverage of the instructions and corner cases

- 5 points - clz instruction and a small test program. (clz_test.s) should show that clz works including various corner cases. Also include a test harness file (clz_test.circ) and its expected output in tests.py.
- 5 points - attach one or more IO devices and include the simplest possible program showing the device works with the CPU (iotest.s, iotest.circ).
- 2 points - difficulty of the IO device (we'll take the highest if you have multiple)
 - 0 points: single LED (output)
 - 1 point: 2+ LEDs (output)
 - 2 points: 7-segment (output), hex display (output)
 - 3 points (yes extra credit): button (because input takes more thought), LED matrix, joystick, several 7-segment/hex displays
 - 4 points (yes extra credit): keyboard, TTY
- 1 points - overall originality of the "interesting MIPS program". Note that this one can be graded independently of the difficulty of the IO device. For example, you might come up with a program where the blinking of a single LED means something interesting (morse code?!). Conversely, filling an LED matrix with all green pixels is not that interesting.

Additional requirements

1. You must sufficiently document your circuits using labels. For sub-circuits, label all inputs and outputs. Label important wires descriptively and label regions of your circuit with what they do.
2. You must make your circuits as legible as possible. Learn to make use of *tunnels* when they will save on messy wiring. (see <http://www.cburch.com/logisim/docs/2.6.0/en/libs/base/tunnel.html>)

Submission checklist

- ✓ Your circuits don't have any errors (Red (E) or orange (wrong width) wires). You ought to avoid blue (X) wires, too.
- ✓ make p2sc runs the tests without crashing
- ✓ Your circuits pass the original tests
- ✓ Your circuits pass additional automated tests that you have written
- ✓ You made a zip file proj2-2.zip that contains these files in the following directory structure:
 1. cpu.circ (your completed CPU)
 2. clz_test.s, clz_test.circ (your test of the clz instruction)
 3. tests/
 - any additional files you've added for your testing
 - Includes
 - your test harness files (i.e., the copies of CPU-starter_kit_test.circ, renamed appropriately, and with instruction memory loaded appropriately)
 - the MIPS assembly files that you used for your tests

- tests.py (because you will add to this file if you use our testing methodology)
- .out files you've added to tests/reference_out (although we recommend putting your expected outputs in test.py as a list of lists)
- 1 .s file and .circ file test harness for your "simple" IO program
- 1 .s file and .circ file test harness for your "interesting" IO program
- Excludes
 - generated files such as the .hex files
- ✓ **Be responsible. Double-check and triple-check your zip file (or github repo) that it contains the correct versions of your files. Near the end of the semester we have no time for exceptions.**
- ✓ **As a team:** One submission by any team member for the team. You are responsible for the contents all being in there on time. You have two options for submission
 1. Upload proj2-2.zip to ICON "Project 2-2: A MIPS Processor"
 2. Submit via github.uiowa.edu instead. To do so, put your files on a **private** repository on github.uiowa.edu, add (xinman, bdmyers, aszecsei) as collaborators, create a tag called "final_submission", and on ICON "Project 2-2: A MIPS Processor" use the text box to provide the link to your repository.
- ✓ **As a team:** keep submitting a weekly update approved by all team members to "Project 2: Evaluation X" assignments on ICON.

Recommended approach to finishing the project

This project involves lots of implementation and testing (both circuits and MIPS code). **We highly recommended** that you get to a basic working processor quickly, which passes some simple tests with support for limited number of instructions and then add complexity from there. (Notice that in the textbook and lectures on MIPS processor design, complexity was added incrementally). **During grading,**

a) a CPU that passes many tests but is missing instructions

will be given more credit than

b) a CPU that passes 0 or a few tests but attempts to support all instructions

Here is a general order we suggest

1. just enough to pass CPU-starter_kit_test.circ
2. create and pass tests for other arithmetic instructions
3. create and pass tests for lw/sw
 - a. at this point you have what you need for I/O so you might have someone start working on that
4. create and pass tests for branches and j instruction

5. create and pass tests for clz
6. create and pass tests for jr
7. create a func_test.circ with ROM loaded with machine code for func_test.mars.s and then pass func_test

Tips

- Do not waste your time writing the instruction memory hex files manually. You should be writing MIPS source programs in MARS. Leave the assembling to the assembler (See the section above called “Assembling and running new programs”).
- Be aware that running the tests will copy alu.circ, regfile.circ, cpu.circ, and mem.circ into the tests/ directory. You **should not** modify those copies (tests/alu.circ, tests/regfile.circ, tests/cpu.circ, tests/mem.circ) because you risk getting mixed up and losing work.
- Do not leave testing until the last minute. You should be testing as you finish each instruction.
- Do not rely on just the provided tests. You must add more. The autograder will test your circuits extensively. **If you fail most of the autograder tests, you will receive a poor grade** (See ICON rubric for scale).
- Do not rely solely on manually testing your circuits (i.e. poking inputs in the Logisim GUI and looking at the output). Manual testing is both time-consuming and error-prone. You should extend the automated tests (as described in the testing sections of this document).

Teamwork tips

- It is your responsibility to keep in contact with your team and notify the staff as early as possible if cooperation problems arise that your team cannot resolve on its own. Often, issues can be remedied if recognized early. The staff's role in problem solving will be to facilitate team discussions and not to criticize individual team members.
- Although we do not require you and your team to use a version control system (e.g., git or svn), we highly recommend doing so to keep track of your changes. If you use version control just be aware that merging .circ files will corrupt them (unlike plain text files), so avoid working on the same file concurrently to avoid merge conflicts all together. Ask the staff for help if you get stuck!
- Logisim circuits are hard to collaborate on unless you break them up into pieces. Therefore, you should break up expertise among the group members. One idea is:
 - datapath design/implementation + control design/implementation + test cases/programs
- Our recommendations are on assigning expertise to team members. You are producing one working product. The team is responsible for the project as a whole.
- **Slip days:** refer to the syllabus

Where to get help

Go in this order:

1. Look for the answer in this document.

2. Refer to the readings and class notes. For example, the design of a processor with a small number of instructions is given in the textbook (of course, you'll want to make sure you build yours to the specifications given in this project document).
3. Get help from your teammates.
4. Find other students to discuss issues at a high level. However, do not share programs or circuit files outside of your team.
5. Refer to the discussion board on Piazza and ask questions there.
6. Ask the staff in class, DYB, or office hours.

Academic honesty

Building your own programmable processor is a highlight of a CS (or related) degree, so do it yourself! We remind you that if you do choose to reuse sub-circuits designed by someone outside of your team that you **clearly cite where they came from**. **Not citing** your sources is plagiarism. You are **strictly prohibited** from looking at solutions to other versions of this project. The staff will be checking design .circ files against similar past projects and watermarks. Any academic dishonesty will result in a zero on this project and report of the incident to the College.

Acknowledgements

- starter code forked from UC Berkeley CS61C
<https://github.com/cs61c-spring2016/proj3-starter>
- document based on UC Berkeley CS61C project 3.2
<http://www-inst.eecs.berkeley.edu/~cs61c/sp16/>
- Helper library register file from Cornell CS3410, Spring 2015
<http://www.cs.cornell.edu/courses/cs3410/2015sp>