CS2630: Computer Organization
Project 2, part 2
Single-cycle MIPS processor with I/O

Due April 26, 2017, 11:59pm
50 points

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
- Incorporate a memory-mapped IO device into the processor and create an interesting application

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 in part 1, we provide the top-level skeleton file and test circuits, and you will provide the implementation and additional tests.

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 ./sanity_test.py p2sc | tee ../TEST_LOG
Error loading circuit file: func_test.circ
Testing files...
$s0 Value       $s1 Value       $s2 Value       $ra Value       $sp Value       Time Step
Fetch Addr      Instruction
xxxxxxxx        xxxxxxxx        xxxxxxxx        xxxxxxxx        xxxxxxxx        00000000
xxxxxxxx        xxxxxxxx
00000000        00000000        00000000        00000000        00000000        00000000
00000000        20100001
        FAILED test: CPU starter test (Did not match expected output)
$s0 Value       $s1 Value       $s2 Value       $sp Value

00000000000000000000000000000000
        FAILED test: func test (Test circuit halted too early)
Passed 0/2 tests
```

In each segment, the first line is your implementation's output and the second line is the expected output. The x's indicate that those bits are disconnected.

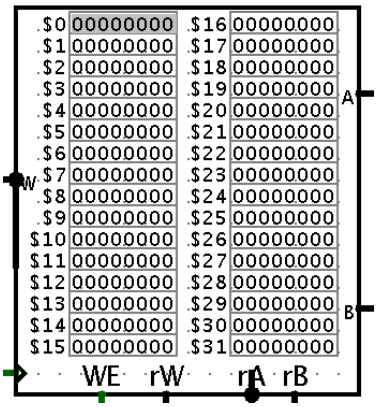
3. Copy your regfile.circ and alu.circ solutions from Project 2-1 into the new directory.

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/51389/files/folder/project2-resources

    You must put regfile.circ in the base directory of your project (as it was in project 2-1) and the cs3410.jar file must go in the 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/51389/files/folder/project2-resources

You must put alu.circ in the base directory of your project (as it was in project 2-1) and the ALU.jar file must go in 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 ./sanity_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
```

6. 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 2-stage pipelined 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
	bgezal

The specification of the instructions is exactly the one in the "MIPS reference card" or "Human-friendly MIPS reference card" on the Resources page http://homepage.cs.uiowa.edu/~bdmyers/cs2630_fa16//resources/ except for the details following.

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).

bgezal – Branch if greater than or equal to zero and link – this instruction is not listed in the MIPS reference sheet, but MARS supports it as a core instruction.

bgezal $t1, label *If $t1 is greater than or equal to zero, then set $ra, to PC+4 and branch to label.*

You must use MARS to reverse engineer the bit encoding of this instruction (hint: it is I-type). Assemble a program that includes the instruction and look at the 32-bit number.

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  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 about 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.

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 with an assembled program (and optionally your data memory).

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

The format of the tests/reference_output/CPU-starter_kit_test.out is:

$s0 Value (32-bit), $s1 Value (32-bit), $s2 Value (32-bit), $ra Value (32-bit), $sp Value (32-bit), Time Step (32-bit), Fetch Addr (32-bit), Instruction (32-bit)

Notice that each group of 4 bits is separated by a single space, and each group of 32-bits is separated by a TAB.

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 major 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.

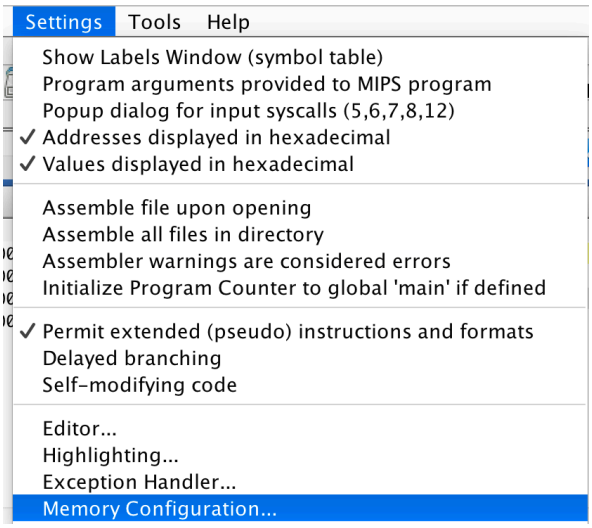
You specify the test Type by making the last argument to TestCase be "cpu", "cpu-lite", or "cpu-end" in your sanity_tests.py file.

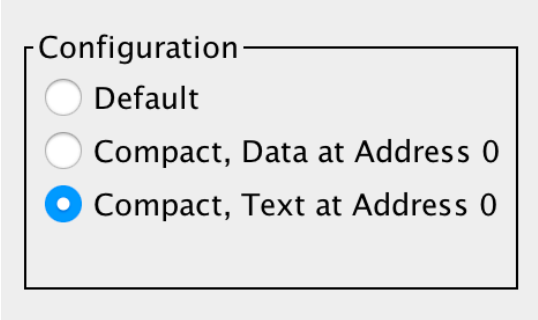
We have provided an example "cpu-end" test called func_test. To use it, you need follow the directions described in the next section. The directions describe a different test, but you will learn the steps you need to create a test harness called func_test.circ and loading its instruction memory with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

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 and Project1).
   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 and Project1).
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
```

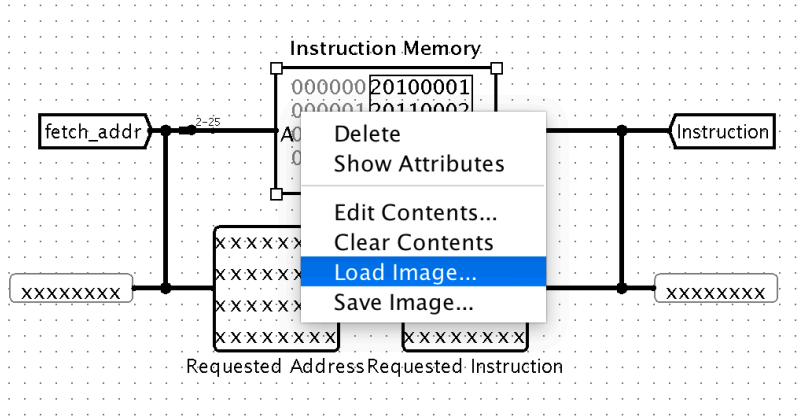
iv. List the files in the folder again to check that there was output.

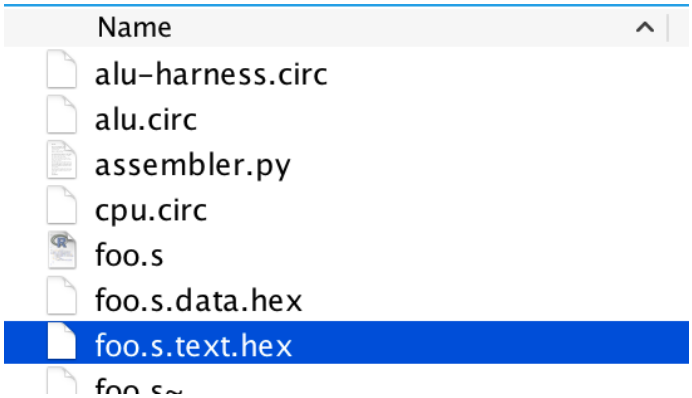
```
Makefile
TEST_LOG
alu-harness.circ
alu.circ
cpu.circ
example_IO_controller.circ
foo.s
foo.s.data.hex
foo.s.text.hex
mars-assem.sh
mars.jar
mem.circ
regfile-harness.circ
regfile.circ
run.circ
tests
text-out.hex
```

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.

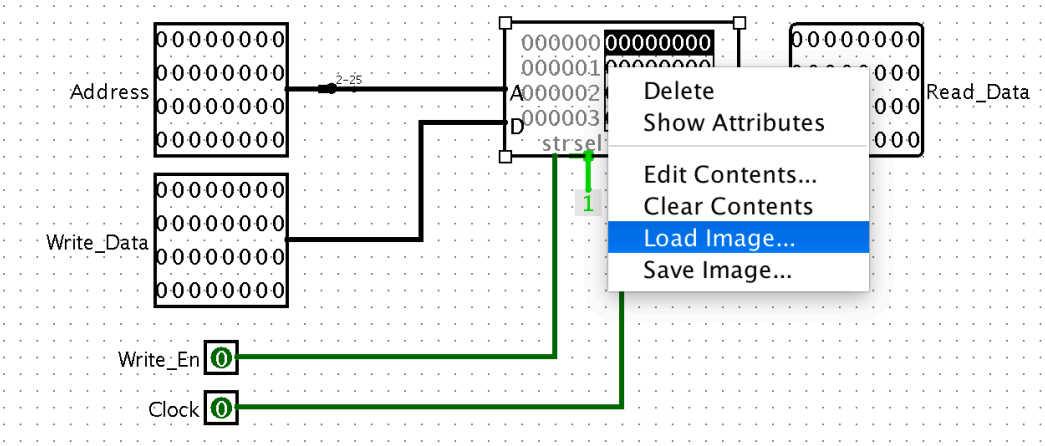
5. Now you can load the program into your processor in Logisim. **Make a copy of CPU-starter_kit_test.circ** and then open that new file in Logisim.

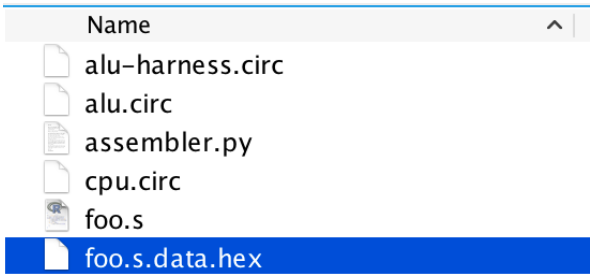
i. Load the instruction memory





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.





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 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.

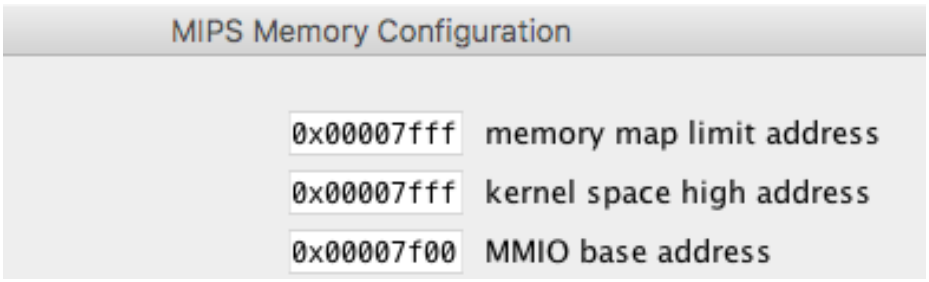
6. Now you can simulate your CPU by ticking the clock. Notice that once the instruction memory gets to instructions 0x00000000, your processor should just be executing NOOPs until you stop the simulation.

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.

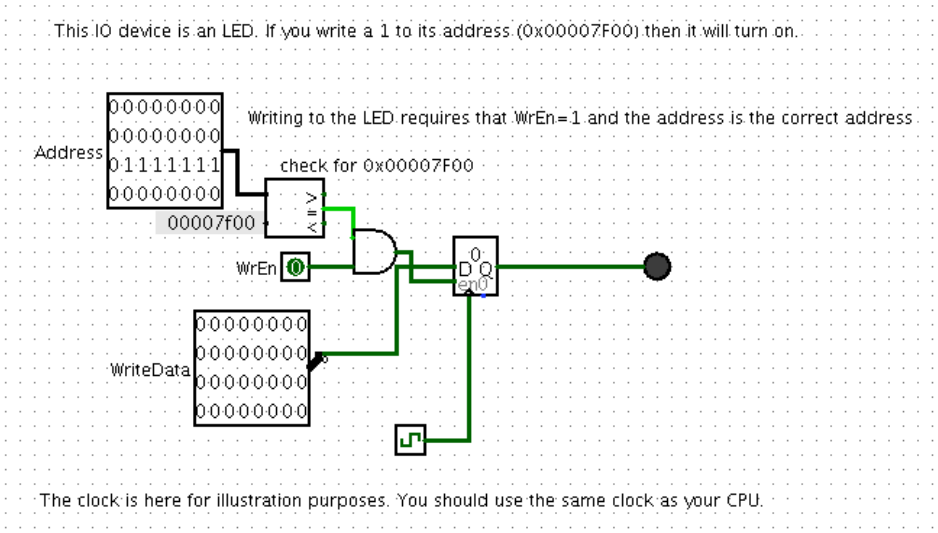
The way MMIO works is that some range of addresses is reserved for controlling IO devices rather than accessing memory.

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

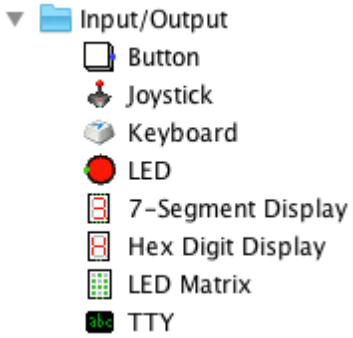


An output device will take in the same Address/MemWrite/WriteData signals as the data memory, except it will *listen* for its range of addresses (0x00007F00 and above). When MemWrite=1 and the Address is in range, the output device will be written to.

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 ICON discussion if you have trouble understanding how to control one of them, and the staff can provide an example.

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. 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.

ii. Write a 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.

iii. Pat yourself on the back. You've built a working and *useful* computer! Show off your work to others.

Grading

- 35 points – the processor passes the autograder tests; the tests you submit demonstrate good coverage of the instructions
- 7 points - attach an IO device. A small test program should show the device works with the CPU.
- 4 points – difficulty of the IO device
- 4 points – overall originality of the "interesting MIPS program"

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 or orange wires).
- ✓ `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. io_readme.txt (description of your application that uses IO)
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
        - sanity_tests.py (because you will add to this file if you use our testing methodology)
        - .out files you've added to tests/reference_out
    - Excludes
        - generated files such as the .hex files

✓ Double-check your zip file

✓ **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 srinivasreddy and bdmyers 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 "Team Evaluation ..." 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 several tests but is missing instructions or IO

will be given more credit than

b) a CPU that passes no or few tests but attempts to support all instructions and IO

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 more functionality.
- 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**.
- 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 either extend the automated tests (as described in the testing sections of this document) or come up with your own automated testing approach.

Teamwork tips

- It is your responsibility to keep in contact with your team and notify the staff as early as possible (the weekly status report is best but email if you are between updates) 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 the work among the group members. Some ways to break up the work are: different members work on different sub-circuits, designs and truth tables, and test cases/MIPS programs.
- **Slip days:** to use a slip day, at least one team member must have one to spend. All team members that have one will consume one for each day taken.

Where to get help

- First, refer to the readings and lecture notes. For example, the design of an ALU is given in the textbook, but you'll want to make sure you build yours to the specifications given in this project document.
- Second, get help from your teammates.
- Third, find other students to discuss issues at a high level. However, do not share solutions or circuit files outside of your team.
- Fourth, refer to the discussion board on ICON and ask questions there.
- Fifth, ask the staff in class, DYB, or office hours.

Academic honesty

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 referring to solutions to other versions of this project.

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