

CS2630: Computer Organization

Project 2, part 1

Register file and ALU for MIPS processor

Goals for this assignment

- Apply knowledge of combinational logic and sequential logic to build two major components of the MIPS processor datapath
- Build digital circuits according to a specification
- Use robust testing methodology in digital logic design

Introduction

In project 2, you will use Logisim to build a processor that can execute actual assembled MIPS programs. You will complete the project in two parts. In part 1, you will build two important components: the ALU and register file. In part 2, you will build the rest of the processor.

Getting started

1. Download the starter code from

<https://github.com/bmyerz/proj3-starter/archive/master.zip>

or clone using git: <https://github.com/bmyerz/proj3-starter.git>

The starter code contains the following files

- Makefile: contains commands for running the tests
- alu-harness.circ: a circuit for testing your ALU
- alu.circ: the skeleton file where you should implement your ALU
- regfile-harness.circ: a circuit for testing your register file
- regfile.circ: the skeleton file where you should implement your ALU
- tests/
 - alu-add.circ: modified version of alu-harness.circ that does ALU tests related to add
 - alu-sra.circ: modified version of alu-harness.circ that does ALU tests related to shift right arithmetic
 - regfile-read_write.circ: modified version of regfile-harness.circ that does register file tests involving reading and writing

- regfile-zero.circ: modified version of regfile-zero.circ that does register file tests involving \$0 (or, \$zero)
- sanity_test.py: python script for testing your circuits
- decode_out.py: python script to format Logisim output
- logisim.jar: a copy of Logisim used by the test code.
- reference_out/
 - text files containing the expected output for each test

2. Try running the tests.

You should pick a method of using Linux command line from <https://uiowa.instructure.com/courses/87896/pages/linux-alternatives>.

Ask for help early if you have trouble using one of those methods.

Ask for help early if you have trouble using one of those methods.

Ask for help early if you have trouble using one of those methods.

- i. Check to be sure your command line of choice has installed:
 - i. make
 - ii. python (version 2.7)
- ii. Open the command line and then go to the directory where the project files are:

cd proj3-starter-master (if you downloaded zip)

cd proj3-starter (if you git cloned)

ls

You should see output like

```
Makefile          alu-harness.circ  alu.circ
regfile-harness.circ regfile.circ      tests
```

- iii. Run the tests

make p1

You should see output like

Testing files...

Error in formatting of Logisim output:

```

        non-integer in ['00000000', 'x', 'x',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx']
    FAILED test: ALU add (with overflow) test, with output
in python (Error in the test)
Error in formatting of Logisim output:
        non-integer in ['00000000', 'x', 'x',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx']
    FAILED test: ALU add (with overflow) test (Error in
the test)
Error in formatting of Logisim output:
        non-integer in ['00000000', 'x', 'x',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx']
    FAILED test: ALU arithmetic right shift test (Error in
the test)
Error in formatting of Logisim output:
        non-integer in ['00000000',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx']
    FAILED test: RegFile read/write test (Error in the
test)
Error in formatting of Logisim output:
        non-integer in ['00000000',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx']
    FAILED test: RegFile $zero test (Error in the test)
Passed 0/5 tests

```

The "Error in formatting of Logisim output" refers to the X's in the outputs.

Arithmetic Logic Unit (ALU)

The ALU is a component that performs one of a number of arithmetic operations on two 32-bit inputs.

Description

The inputs and outputs of the ALU are as follows.

Input

Name	Bit width	Description
X	32	First operand
Y	32	Second operand
Switch	4	the operation the ALU should compute (same purpose as ALU control from the textbook) to obtain the value of output Result

Output

Name	Bit width	Description
Signed overflow	1	1 iff operation is add or sub and there was signed overflow
Equal	1	1 iff $X == Y$
Result	32	Result of the operation

Here is the specification for your ALU.

Switch	ALU operation name	Result	Signed Overflow
0	sll	$Y \ll X$	0
1	srl	$Y \gg X$ (logical)	0
2	sra	$Y \gg X$ (arithmetic)	0
3	add	$X + Y$	1 iff there is signed overflow
4	addu	$X + Y$	0
5	sub	$X - Y$	1 iff there is signed overflow
6	subu	$X - Y$	0
7	and	$X \text{ AND } Y$	0
8	or	$X \text{ OR } Y$	0
9	xor	$X \text{ XOR } Y$	0
10	slt (<i>signed</i> comparison)	$(X < Y) ? 1 : 0$	0
11	sltu (<i>unsigned</i> comparison)	$(X < Y) ? 1 : 0$	0


Notes:

- REMINDER: ALU OPERATIONS ARE NOT THE SAME THING AS MIPS INSTRUCTIONS!

- These are operations of the ALU. We've named some of them the same as MIPS instructions (e.g., sll, srl). However, they can be used by many different instructions. For example, ALU operation add is used by add, addi, and other instructions.
- Understanding the difference between ALU operations and MIPS instructions now will save you confusion during Project 2-2.
- slt does *signed* comparison (e.g., 0xFFFFFFFF < 0x00000000 evaluates to 1) and sltu does *unsigned* comparison (e.g., 0xFFFFFFFF < 0x00000000 evaluates to 0).
- The output Equal must always be the value of X==Y
- Signed Overflow must only ever be 1 in the cases above.
 - Signed overflow is not simply carry out. Signed overflow occurs when add or subtract produces the *wrong* answer because you have too few bits.
 - We suggest that you first work out examples on paper with smaller (e.g. 5-bit) two's complement numbers.
- For the shift amount, given by X in sll, srl, sra, your ALU should only consider the least significant 5 bits.

What you must implement

You must modify alu.circ to implement the ALU. 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 alu.circ. You may use any *built-in* Logisim components.

How you must test

To test your ALU with the provided test cases, open up the command line.

1. go to the base directory with your project files

```
cd ~/path/to/unzipped/project/files    (use the actual path!)
```

2. run the tests

```
make p1
```

If the ALU tests pass then you should see something like

Testing files...

```
PASSED test: ALU add (with overflow) test, with output in python
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 5/5 tests

Note that in the above example, we haven't implemented the register file yet, so those tests are failing.

If an ALU test *fails*, you will see more debugging information about that test case.

Testing files...

Format is student then expected

Test #	OF	Eq	Result
0	0	0	7659035d
0	0	0	7659035d
1	1	0	87a09724
1	1	0	87a08d79

FAILED test: ALU add (with overflow) test (Did not match expected output)

PASSED test: ALU arithmetic right shift test


The above output indicates that the "ALU arithmetic right shift test" test passed but that the "ALU add (with overflow) test" failed. The three lines above the "FAILED test: ALU arithmetic..." show the output of your circuit followed by the expected output. In the example, in cycle 1, our implementation gave the result

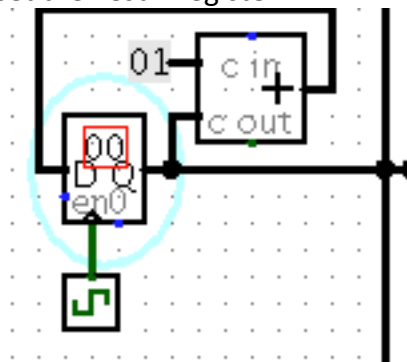
87a09724

but the correct answer was

87a08d79

To find out more about the test that failed, we can open up `tests/alu-add.circ` in

Logisim. The three memories contain the test inputs. You can use the poke tool  to set the Test # register



to the Test # given in the error message. Our test's error message said

Test #	OF	Eq	Result
...			
1	1	0	87a09724
1	1	0	87a08d79

The Test # is 1, so we would want to set the Test # register to 1 to see which inputs for X, Y, and Switch were being tested.

3. You **must** test your circuit beyond the two example test cases, alu-add and alu-sra. Our autograder will test all the ALU operations on many kinds of inputs.

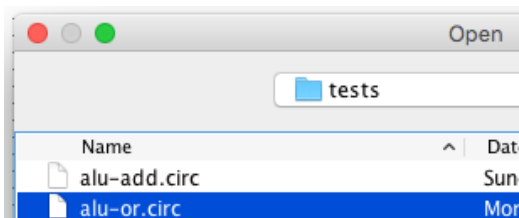
You may test your circuit however you like. We recommend extending the existing testing infrastructure. We give an example below.

To add new tests, do the following

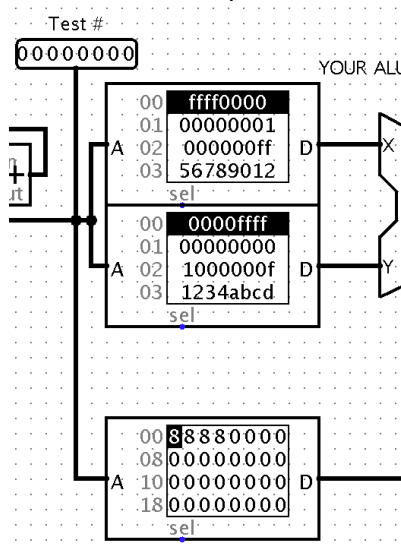
- i. Make a copy of the test harness and put it in tests/. You should pick a descriptive name. Here, we picked tests/alu-or.circ to indicate that we are testing OR.

```
cp alu-harness.circ tests/alu-or.circ
```

- ii. Open the new test circuit in Logisim.



- iii. Add some test inputs



- iv. Save your file. You must save to keep the entries in the ROMs.
- v. Add your test to the list in tests/tests.py

```
("ALU or",
    TestCase(os.path.join(file_locations,'alu-or.circ'),
        [[0,0,0,0xffffffff],
         [1,0,0,0x00000001],
         [2,0,0,0x100000ff],
         [3,0,0,0x567CBBDF],
         [4,0,1,0x00000000]]), "alu"),
```

About the list-of-lists. Each inner list represents the test case # and the expected outputs of each output pin. These correspond to the columns you see in the output of `make p1`. You can specify the integers in any format Python supports (e.g., decimal, binary (0b), hex (0x)).

- vi. Run the tests with the `make` command, as before. You should now see messages about the new test.

Register File

Description

The register file presents a memory-like interface for reading and writing the 32 registers of the MIPS ISA.

The inputs and outputs are as follows.

Inputs

Name	Bit width	Description
Read Register 1	5	The number of the register whose contents is read out to Read Data 1. <i>E.g., 01000 would be register \$8, or \$t0</i>
Read Register 2	5	The number of the register whose contents is read out to Read Data 2
Write Register	5	The number for the register to be written at the rising edge of Clock
Write Data	32	The data to write to the register specified by Write Register
Write Enable	1	If 1, then a register will be written at the rising edge of Clock. If 0, no register will be written at the rising edge of Clock.
Clock	1	The clock signal

Outputs


Name	Bit width	Description
Read Data 1	32	The data read from the register specified by Read Register 1
Read Data 2	32	The data read from the register specified by Read Register 2
\$s0 Value	32	(Debugging output) the value stored in register \$s0)

\$s1 Value	32	(Debugging output) the value stored in register \$s1)
\$s2 Value	32	(Debugging output) the value stored in register \$s2)
\$ra Value	32	(Debugging output) the value stored in register \$ra)
\$sp Value	32	(Debugging output) the value stored in register \$sp)

The last 6 outputs are called "debugging output" because they are not typical register file ports but do provide convenient access to the values of those 6 registers for testing/debugging purposes. These debugging outputs will be required by the autograder for cycle-by-cycle testing of the completed processor in part 2 of the project. These outputs must always have the value of the given register.

Remember that register \$0 (aka, \$zero) is not writeable. If the inputs say to write to \$0, ***no change occurs*** to the register file on that cycle.

What you must implement

You should edit the file regfile.circ. Do not move or modify the inputs and outputs. You may use sub-circuits in your implementation as long as  **main** remains the top level circuit of regfile.circ. You may use any *built-in* Logisim components.

How you must test

1. Run the tests as described in the "How you must test" section of the ALU. The difference is that you'll want to pay attention to the tests labeled "Regfile..."
2. To ensure you pass the autograder, **you must test your register file beyond the given tests**. See the "How you must test" section of the ALU, under step 3 for an example of adding a test. The difference is that for each new test file you create, you should copy regfile-harness.circ to a new file in tests/.

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

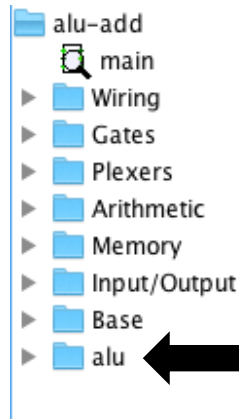
- Your circuits don't have any errors (Red or orange wires)
- `make p1` runs the tests without crashing
- Your circuits pass the original tests
- Your circuits pass additional automated tests that you have written
- You made a zip file¹ `proj2-1.zip` that contains these files in the following directory structure:
 - `alu.circ` (Your completed ALU)
 - `regfile.circ` (Your complete register file)
 - `tests/`
 - any additional files you've added for your testing. **That includes** the testing circ files and your modified `tests.py`.
- Double-check your zip file
- **As a team:** Upload `proj2-1.zip` to ICON Project 2-1. One submission by any team member for the team. You are responsible for the contents all being in there.
- **As a team each week**
 - On ICON submit a brief "Team Evaluation" describing what progress you've made and what each team member worked on. You should agree on what goes in this document and only submit one per team. Each is worth 1 homework point.

Tips

- Be aware that running the tests will copy `alu.circ` and `regfile.circ` into the `tests/` directory. You **should not** modify those copies (`tests/alu.circ` and `tests/regfile.circ`) because you risk getting mixed up and losing work. You should only modify the copies of `alu.circ` and `regfile.circ` that are in the base of your project files directory.
- 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.
- While you should run through your tests using our testing infrastructure, once you observe a failure, you should **debug** manually using Logisim's GUI. Open the test circuit (e.g., `alu-add.circ` if `alu-add` is the failing test) in Logisim, tick to the test case # you are failing, and then use the poke tool to trace backwards through the circuit until you find the source of the wrong value.
- Notice that when you open one of the test circuits (e.g. `tests/regfile-read_write.circ`,

¹ You may either create the zip file on your computer as you normally would, or if you are using Git and GitHub, you can push all your commits and then download your repository as a zip file.

tests/alu-add.circ, tests/alu-sra.circ, tests/regfile-zero.circ), Logisim will have an additional folder in the component browser.



This additional folder contains the circuit that is in tests/alu.circ or tests/regfile.circ, respectively. Note that from here, you cannot (and should not!) edit your ALU or register file implementation. Think of it as a read-only library, just like the other Logisim components.

Teamwork tips

- Feel free to split the work as appropriate for your team. If you are unsure, then a good starting point is ALU, Register file, and Testing.
- 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 logisim's .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.
- **Slip days:** the late policy on the syllabus applies. See the discussion of how slip days work in group projects.

Where to get help

- **I don't know what {alu.circ, regfile.circ} is supposed to do.**
 - Re-read the {alu, register file} section
 - Refer to readings and class notes
 - Refer to the test cases, the expected outputs are in tests/test.py and the inputs are in the ROMs of the .circ files in tests/
- First, refer to the readings and class 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. The register file is related to Inquiry activities and homeworks on sequential logic and memories.

- 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 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 circuits designed by someone outside of your team that you clearly cite where they came from. Not citing your sources is plagiarism.

Acknowledgements

- starter code forked from UC Berkeley CS61C
<https://github.com/cs61c-spring2016/proj3-starter>
- document based on UC Berkeley CS61C project 3.1
http://www-inst.eecs.berkeley.edu/~cs61c/sp16/projs/03_1/