

# CS2630: Computer Organization

## Project 2, part 1

### Register file and ALU for MIPS processor

Due Friday, Nov 11, 2016, 11:59pm

#### 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>

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

- i. Check to be sure you have installed:
  - i. make
  - ii. python 2.7

If you use the instructional machines, then make and python are already installed. You can access instructional machines by three options: lab machines, ssh to <hawkid>@linux.divms.uiowa.edu, or use <https://fastx.divms.uiowa.edu>. Ask for help early if you have trouble using one of those methods.

- ii. Open the command line and then go to the directory where the project files are:

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

```

cp alu.circ regfile.circ tests
cd tests && python ./sanity_test.py | tee ../TEST_LOG
Testing files...
Test #  OF      Eq      Result
00      x      x      xxxxxxxx
00      0      0      7659035d
      FAILED test: ALU add (with overflow) test (Did not match expected output)
Test #  OF      Eq      Result
00      x      x      xxxxxxxx
00      0      0      f7ab6fbb
      FAILED test: ALU arithmetic right shift test (Did not match expected output)
Test #  $s0 Value  $s1 Value  $s2 Value  $ra Value  $sp Value  Read Data 1  Read Data 2
00      xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
00      00000000  00000000  00000000  00000000  00000000  00000000  00000000
      FAILED test: RegFile read/write test (Did not match expected output)
Test #  $s0 Value  $s1 Value  $s2 Value  $ra Value  $sp Value  Read Data 1  Read Data 2
00      xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
00      00000000  00000000  00000000  00000000  00000000  00000000  00000000
      FAILED test: RegFile $zero test (Did not match expected output)
Passed 0/4 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.

## Arithmetic Logic Unit (ALU)

### 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	Operation name	Result	Signed Overflow
0	sll	$Y \ll X$	0
1	srl	$Y \gg X$	0
2	sra	$Y \ggg X$	0
5	add	$X+Y$	1 iff there is signed overflow
6	addu	$X+Y$	0
7	sub	$X-Y$	1 iff there is signed overflow
8	subu	$X-Y$	0
9	and	$X \text{ AND } Y$	0
10	or	$X \text{ OR } Y$	0
14	slt ( <i>signed</i> comparison)	$(X < Y) ? 1 : 0$	0
15	sltu ( <i>unsigned</i> comparison)	$(X < Y) ? 1 : 0$	0

Notes:

- slt must do *signed* comparison (e.g.,  $0xFFFFFFFF < 0x00000000$  evaluates to 1) and sltu must do *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. There is a primer on signed overflow at <http://www.allaboutcircuits.com/textbook/digital/chpt-2/binary-overflow/>.

### 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

```

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
Test # $s0 Value    $s1 Value    $s2 Value    $ra Value    $sp Value    Read Data 1    Read Data 2
00     00000000     00000000     00000000     00000000     00000000     xxxxxxxx      xxxxxxxx
00     00000000     00000000     00000000     00000000     00000000     00000000      00000000
    FAILED test: RegFile read/write test (Did not match expected output)
Test # $s0 Value    $s1 Value    $s2 Value    $ra Value    $sp Value    Read Data 1    Read Data 2
00     00000000     00000000     00000000     00000000     00000000     xxxxxxxx      xxxxxxxx
00     00000000     00000000     00000000     00000000     00000000     00000000      00000000
    FAILED test: RegFile $zero test (Did not match expected output)
Passed 2/4 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.

```

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
Test # OF      Eq      Result
00     0       0      37ab6fbb
00     0       0      f7ab6fbb
    FAILED test: ALU arithmetic right shift test (Did not match expected output)
Test # $s0 Value    $s1 Value    $s2 Value    $ra Value    $sp Value    Read Data 1    Read Data 2
00     00000000     00000000     00000000     00000000     00000000     xxxxxxxx      xxxxxxxx
00     00000000     00000000     00000000     00000000     00000000     00000000      00000000
    FAILED test: RegFile read/write test (Did not match expected output)
Test # $s0 Value    $s1 Value    $s2 Value    $ra Value    $sp Value    Read Data 1    Read Data 2
00     00000000     00000000     00000000     00000000     00000000     xxxxxxxx      xxxxxxxx
00     00000000     00000000     00000000     00000000     00000000     00000000      00000000
    FAILED test: RegFile $zero test (Did not match expected output)
Passed 1/4 tests

```

The above output indicates that the “ALU add (with overflow)” test passed but that the “ALU arithmetic right shift 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, our implementation gave the result


37ab6fbb (has leading 0's)

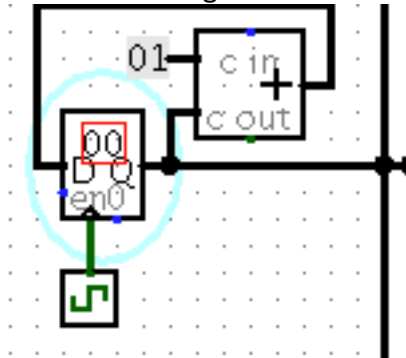
but the correct answer was

f7ab6fbb (has leading 1's)

In this case, we can infer that our ALU is computing shift right logical when it is supposed to be computing shift right arithmetic.

To find out more about the test that failed, we can open up `tests/alu-sra.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
00	0	0	37ab6fbb
00	0	0	f7ab6fbb

The Test # is 00, so we would want to set the register to 00 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 many different cases.

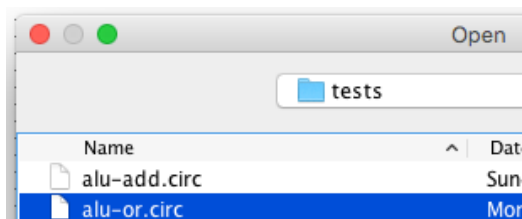
You test your circuit however you like. We recommend extending the existing testing infrastructure. We give an example below.

To add new tests, do the following

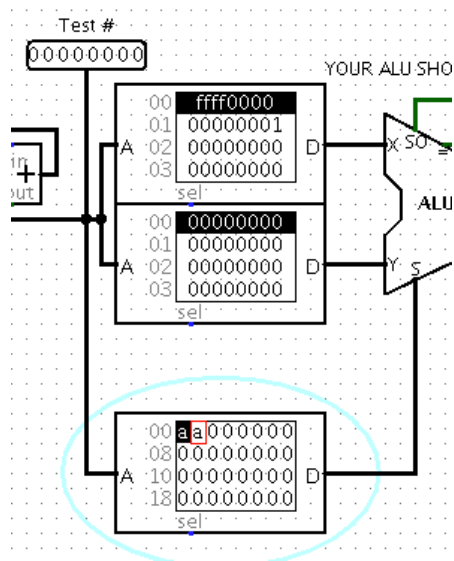
- i. Make a copy of the test harness and put it in `tests/`. You should pick a descriptive name. Here, we picked `tests/alu-or.circ` to indicate that we are testing OR.

```
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 RAMs.
- v. Create a new file in tests/reference\_output/ that has one line per expected output for the test inputs you created in the previous step. You can use the other .out files as examples.

Here, we created a file tests/reference\_output/alu-or.circ with the following contents.

```
0000 0000 0 0 1111 1111 1111 1111 1111 1111 1111 1111
0000 0001 0 0 0000 0000 0000 0000 0000 0000 0000 0001
0000 0010 0 1 0000 0000 0000 0000 0000 0000 0000 0000
```

- vi. Add your test to tests/sanity\_tests.py

```
p1_tests = [
    ("ALU add (with overflow) test",
     TestCase(os.path.join(file_locations, 'alu-add.circ'),
              os.path.join(file_locations, 'reference_output/alu-add.out')), "alu"),
    ("ALU arithmetic right shift test",
     TestCase(os.path.join(file_locations, 'alu-sra.circ'),
              os.path.join(file_locations, 'reference_output/alu-sra.out')), "alu"),
    ("ALU or",
     TestCase(os.path.join(file_locations, 'alu-or.circ'),
              os.path.join(file_locations, 'reference_output/alu-or.out')), "alu"),
    ("RegFile read/write test",
```

- vii. 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
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 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

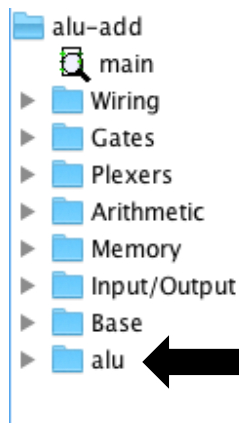
1. You must sufficiently document your circuits using labels. For sub-circuits, label all inputs and outputs. Label important wires descriptively and label regions of your circuit with what they do.
2. You must make your circuits as legible as possible. Learn to make use of *tunnels* when they will save 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 file `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. *Does not* need to include the files we provided you *unless* you changed them (e.g., you will change `sanity_tests.py` if you use our testing methodology, so you can include that file in your submission).
- 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.
- **Individually:** submit an entry to ICON "Project2-1 Evaluation". In this online text submission, you should briefly describe (a) your own contribution to the project and (b) each of your teammate's contributions.

## 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.
- Notice that when you open one of the test circuits (e.g. `tests/regfile-read_write.circ`, `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

- 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.
- **Slip days:** to use a slip day, all team members must have at least one available to spend. As usual, you may use no more than 2 slip days on this assignment.

### 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 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/](http://www-inst.eecs.berkeley.edu/~cs61c/sp16/projs/03_1/)