

Homework 4

ALU and ALUControl for ARM processor

Table of Contents

1	GOALS FOR THIS ASSIGNMENT	2
2	INTRODUCTION.....	2
3	GETTING STARTED.....	2
4	ARITHMETIC/LOGICAL UNIT (ALU)	4
4.1	DESCRIPTION.....	4
4.2	WHAT YOU MUST IMPLEMENT	6
4.3	HOW YOU MUST TEST	6
4.3.1	DEBUGGING CASE STUDY.....	6
4.3.2	WRITING YOUR OWN TESTS FOR THE ALU	8
5	ALU CONTROL UNIT.....	10
5.1	WHAT YOU MUST IMPLEMENT	11
5.2	HOW YOU MUST TEST	11
6	ADDITIONAL REQUIREMENTS	11
7	REPORT.....	12
8	SUBMISSION CHECKLIST	12
9	TIPS	12
10	WHERE TO GET HELP	13
11	ACADEMIC HONESTY.....	14

1 Goals for this assignment

- Apply knowledge of combinational logic to build important components of the ARM processor
- Build digital circuits according to a specification
- Use robust testing methodology in digital logic design

2 Introduction

In project 2 (HW4 and HW5), you will use Logisim to build a processor that can execute actual assembled ARM programs. You will complete the project in two parts. In part 1 (HW4), you will build the ALU and the ALU Control. In part 2 (HW5), you will build the rest of the processor.

Beyond building something, an important aspect of this project is learning how to use the systematic testing and debugging process described in this document. This skill will be even more critical in Part 2, so practice it now.

3 Getting started

1. Download the starter code from
<https://github.com/bmyerz/project2-alu-fa21/archive/refs/heads/main.zip>
or if using git then you can clone <https://github.com/bmyerz/project2-alu-fa21.git>

The starter code contains the following files

- Makefile: contains commands for running the tests
- alu.circ: the skeleton file where you should implement your ALU
- alu-control: the skeleton file where you should implement your ALU Control
- tests/
 - alu-add.circ: tests alu.circ's add operation
 - alu-asr.circ: tests alu.circ's arithmetic shift right operation
 - alu-control-b-test.circ: tests alu-control.circ for b instruction
 - alu-control-basic-data-processing-test.circ: tests alu-control.circ for basic data-processing instructions
 - alu-control-cmp-test.circ: tests alu-control.circ for cmp instruction
 - alu-control-ldr-str-test.circ: tests alu-control.circ for ldr and str instructions
 - alu-control-shifts-test.circ: tests alu-control.circ for shift instructions
 - test.py: python script for testing your circuits; this is also where the expected outputs of the tests go
 - decode_out.py: python script to format Logisim output

- logisim.jar: a copy of Logisim used by the test code.

2. Try running the tests.

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 3)

Note that the remote Linux lab computers (i.e., divms accessed through FastX or ssh) already have these installed for you.

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

```
cd project2-alu-fa21-master (if you downloaded zip)
```

```
cd project2-alu-fa21 (if you git cloned)
```

```
ls
```

You should see output like

```
Makefile          alu.circ
alu-control.circ  tests
```

- iii. Run the tests

```
make p1
```

You should see output like

```
cp alu.circ alu-control.circ tests
cd tests && python3 ./test.py | tee ../TEST_LOG
Testing files...
Error in formatting of Logisim output:
  non-integer in ['00000000', 'xxxx', 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx']
  FAILED test: ALU add (with overflow) test (Error in the test)
Error in formatting of Logisim output:
  non-integer in ['00000000', 'xxxx', 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx']
  FAILED test: ALU arithmetic right shift test (Error in the test)
Error in formatting of Logisim output:
  non-integer in ['00000000', 'xxxx']
  FAILED test: ALU Control Basic Data-processing Instructions Test (Error in the test)
```

```

Error in formatting of Logisim output:
    non-integer in ['00000000', 'xxxx']
    FAILED test: ALU Control Shifts Test (Error in the test)
Error in formatting of Logisim output:
    non-integer in ['00000000', 'xxxx']
    FAILED test: ALU Control CMP Test (Error in the test)
Error in formatting of Logisim output:
    non-integer in ['00000000', 'xxxx']
    FAILED test: ALU Control B Test (Error in the test)
Error in formatting of Logisim output:
    non-integer in ['00000000', 'xxxx']
    FAILED test: ALU Control ldr/str Test (Error in the test)
Passed 0/7 tests

```

"Error in formatting of Logisim output" refers to the X's in the outputs (blue wires), which indicate that there is no circuit providing those outputs with either a 1 or 0.

4 Arithmetic/Logical Unit (ALU)

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

4.1 Description

The inputs and outputs of the ALU are as follows.

Input

Name	Bit width	Description
A	32	First operand
B	32	Second operand
ALUControl	4	the operation the ALU should compute to obtain the value of output Result

Output

Name	Bit width	Description
ALUFlags	4	{Negative, Zero, Carry, oVerflow}
Result	32	Result of the operation

Here are the individual bits in the ALUFlags

Name	Bit width	Description
oVerflow	1	1 iff operation is add or sub and there was signed overflow
Carry	1	1 iff operation is add or sub and there was carry out
Zero	1	1 iff Result is equal to 0
Negative	1	1 iff Result is negative

Here is the specification for the Result.

ALUControl	ALU operation name (<i>NOT</i> an ARM instruction)	Result
0	and	A AND B
1	or	A OR B
2	xor	A XOR B
3	not	NOT A
4	lsl	B << A
5	lsr	B >> A (logical)
6	asr	B >> A (arithmetic)
7	add	A+B
8	sub	A-B


Notes:

- **REMINDER: ALU OPERATIONS ARE NOT THE SAME THING AS ARM INSTRUCTIONS, even though they are named similarly.**
- **REMINDER: ALU OPERATIONS ARE NOT THE SAME THING AS ARM INSTRUCTIONS, even though they are named similarly.**
- **DID WE MENTION? ALU OPERATIONS ARE NOT THE SAME THING AS ARM INSTRUCTIONS, even though they are named similarly.**
 - These are operations of the ALU. We've named some of them the same as ARM instructions (e.g., lsl, lsr). However, they can be used by many different instructions. For example, the ALU operation add is used by add, ldr, str, and other instructions.
 - Understanding the difference between ALU operations and ARM instructions now will save you confusion during the next project. If you don't see the difference, revisit the chapter on the ALU in the textbook. In short, the ALU is only a small piece of the processor. All it does is perform an arithmetic or logical operation on two inputs, whereas an ARM instruction does much more, such as reading and writing registers, updating the PC, reading and writing memory, etc.
- oVerflow and carry must only be 1 in the cases given in the table
 - Signed overflow is not the same as carry out. Signed overflow occurs when add or subtract produces the *wrong* answer because you have too few bits.
 - Suggestions:
 - do the reading on ALU and the Knowledge Check
 - work out examples on paper with smaller (e.g. 5-bit) two's complement numbers

- For the shift amount, given by A in lsl, lsr, and asr, your ALU should only consider the **least significant 5 bits of A**. The alu-asr.circ test includes cases where this is relevant.

4.2 What you must implement

You must modify alu.circ to implement the ALU. **Do not modify, move, add, or delete** any inputs and outputs. Doing so will break the tests. You may use [subcircuits](#) in your

implementation as long as  **main** remains the top-level circuit of alu.circ. You may use any *built-in* Logisim components.

4.3 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
    PASSED test: ALU arithmetic right shift test

Passed 2/2 tests
```

4.3.1 Debugging case study

This section shows you the procedure you should use to debug your circuit.

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

```
Testing files...
Format is student then expected
Test #  ALUFlags  Result
0       0         7659035d
0       0         7659035d
1       8         87a08d79
1       9         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 ALUFlags


8 (that is 1000, or N=1, Z=0, C=0, V=0)

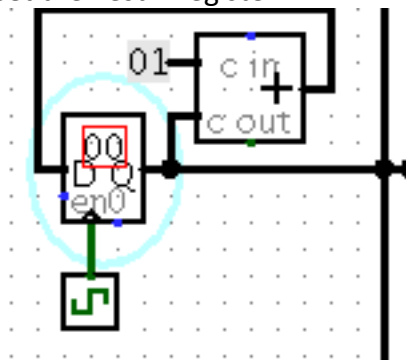
but the correct answer was

9 (that is 1001, or N=1, Z=0, C=0, V=1)

So it looks like overflow happened, but our ALU didn’t detect it.

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

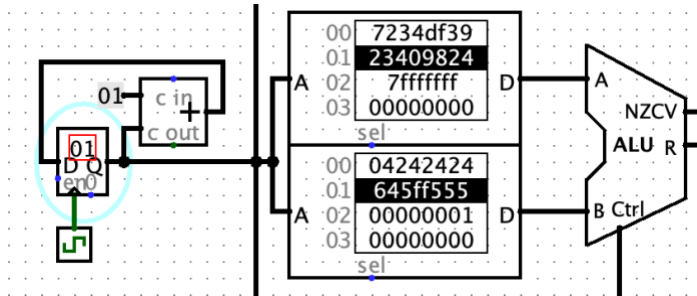
Logisim. The three memories contain the test inputs. You can use the poke tool  to set the Test # register



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

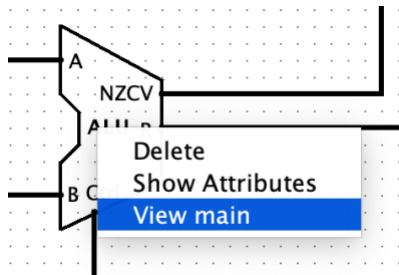
Test #	ALUFlags	Result
...		
1	8	87a08d79
1	9	87a08d79

The Test # is 1, so we would want to set the Test # register to 1 to see which inputs for A, B, and ALUControl were being tested.



In this case it was $A=0x23409824$, $B=0x645ff555$, $ALUControl=7$ (that is, add). Indeed, this addition should lead to an overflow, but our ALU is outputting that overflow=0.

From here, to see why our ALU is outputting the wrong answer, right click the ALU and choose view Main.



This will take you into your `alu.circ`. The test inputs will be coming in so that you can investigate what is happening on all the wires. **Trace wires backwards from output towards the input.** That is, start at the ALUFlags output, and look for where overflow is coming from. Keep tracing backwards until you find an unexpected value on a wire; that will help you find the cause.

- Tip: To see the value on a wire either:
 - Use the poke tool (hand with pointer finger)
 - Or place a Probe, found under Wiring folder

4.3.2 Writing your own tests for the ALU

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.

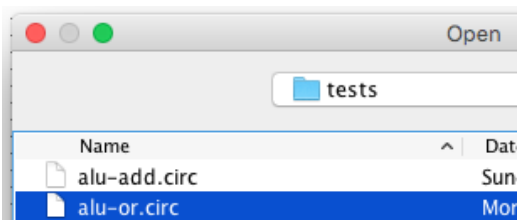
We give an example below of how to add your own test.

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 tests/alu-add.circ tests/alu-or.circ
```

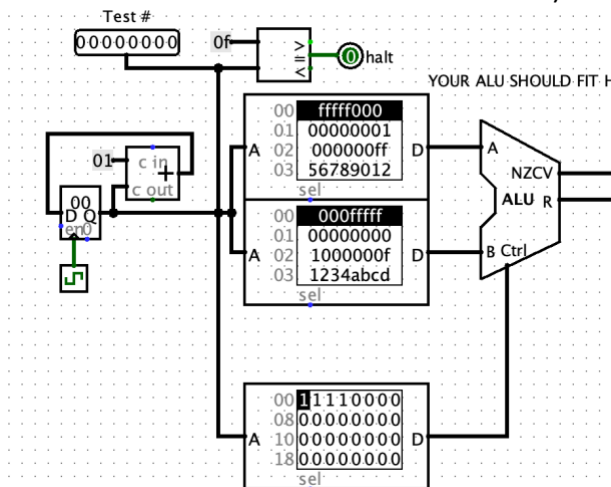
- ii. Open the new test circuit in Logisim.



IMPORTANT: Logisim should just open the file successfully with no further action from you. However, if Logisim asks you to find the dependency alu.circ, then you should exit the prompt. **DO NOT** use the file explorer to find the file for Logisim. Instead, first make sure that there is a copy of alu.circ in your tests/ folder. If there isn't then run the "make p1" command from the base directory (this command copies alu.circ for you). Then try opening alu-or.circ again.

If you already have a copy of alu.circ in the tests/ folder and still get the message about finding alu.circ, then this indicates that your alu-or.circ file needs to be fixed. Although you can fix it manually in a text editor, we recommend just re-downloading a fresh copy of one of the sample tests and copying again.

- iii. Add some test inputs (note that the bottom ROM values should not be 8: they should match the ALUControl code for OR).



- 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'),
              [[0,8,0xffffffff],
               [1,0,0x00000001],
               [2,0,0x100000ff],
```

```
[3,0,0x567CBBDF],  
[4,4,0x00000000]]), "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 (no prefix), binary (0b), hex (0x)). So pick the number base you prefer to use.

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

5 ALU Control Unit

In your eventual implementation of the ARM processor, you'll need to determine which ALUControl to perform based on the instruction that is currently running. That instruction is identified by its Op, as well as its Funct for Data-processing instructions, and sh for shift instructions.

In this assignment, you will build that part of the processor that translates from Op, Funct, and sh into the ALUControl.


The instructions whose Op/Funct/sh you need to support right now are in the following table, along with the ALUControl each instruction relies on.

Instruction (see ARM reference sheet for Op, Funct, and sh)	ALUControl (see table in 4.1 for 4-bit encoding)	The rationale for why the instruction uses this ALUControl, when we implement the ARM Processor in the next project
Data-processing instructions: Op=00 ₂ and ALUControl is then determined by cmd (the middle 4 bits of Funct).		
add	add (i.e., 7)	the ALU will be used to add two values
sub	sub (i.e., 8)	the ALU will be used to subtract two values
and	and	the ALU will be used to AND two values
orr	or	the ALU will be used to OR two values
eor	xor	the ALU will be used to XOR two values
mvn	not	the ALU will be used to complement a value
cmp	sub	this instruction subtracts the two operands to determine the flags
Data-processing instructions (shifts): Op=00 ₂ , cmd=1101 ₂ , and ALUControl is then determined by sh		
lsl	lsl	the ALU will be used to lsl one value by another
lsr	lsr	the ALU will be used to lsr one value by another
asr	asr	the ALU will be used to asr one value by another

Branch instructions: Op=10 ₂		
b	add	to do the (PC+8) + (sign extended imm24)
Memory: Op=01 ₂		
ldr	add	the ALU will be used to compute the memory address
str	add	the ALU will be used to compute the memory address

5.1 What you must implement

Build the logic for the ALU Control Unit in alu-control.circ. It takes a 2-bit Op, 6-bit Funct, and 2-bit sh, and has one 4-bit output, the ALUControl indicating what ALU operation is needed.

Do not modify, move, add, or delete any inputs and outputs. Doing so will break the tests. You may use [subcircuits](#) in your implementation as long as  **main** remains the top-level circuit of alu-control.circ. You may use any *built-in* Logisim components.

Tips:

- this is just a lot of conditional logic! “If op, funct, sh is this, then ALUControl is that.” If you feel stuck getting started, it might help you to write the ALUControl algorithm as pseudocode first, using if-statements or switch-case statements. Then convert those statements to a circuit. See the lecture on turning if-statements into circuits (basically, multiplexers and comparators are helpful).
- Truth tables where the inputs are a subset of all the inputs to ALUControl (e.g., just op, or just the 4 cmd bits of funct) might also be helpful. You can convert your truth table to a circuit on paper, or [you can use Logisim’s combinational analysis to convert the truth table for you](#).

5.2 How you must test

See Section 4.3 for general information about tests. The tests for ALU Control use the same framework. For ALU Control, you can find test circuits in the circuits called alu-control-???-test.circ. Unlike with the ALU, these tests are fairly comprehensive, so you probably don’t need to add more.

6 Additional requirements

1. You must sufficiently document your circuits using labels. For subcircuits, 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.

7 Report

You will explain what you did by answering the following questions. Submit as a document.

1. Explain the process of developing your `alu.circ`. How did you get started? What is the sequence of steps you took to complete it? (3-5 sentences)
2. Explain the process of developing your `alu-control.circ`. How did you get started? What is the sequence of steps you took to complete it? (3-5 sentences)
3. Explain the difference between overflow and carry for two's complement adding and subtracting. (2-4 sentences)
4. Explain how your implementation of `alu-control.circ` works. What approaches did you use to implement all the different cases? (3-6 sentences)

8 Submission checklist

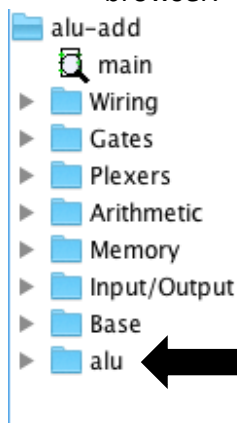
- Your circuits don't have any errors (Red, blue, or orange wires)
- `make p1` runs the tests without crashing
- Your circuits pass the original tests
- Your circuits pass additional automated tests that you have written
- You made a zip file¹ `hw4.zip` that contains these files in the following directory structure:
 - `alu.circ` (Your completed ALU)
 - `alu-control.circ` (Your completed ALU Control)
 - `tests/`
 - optionally, 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
- Upload `hw4.zip` and a report document to ICON Homework 4. You are responsible for the contents all being in there.

9 Tips

- Do not leave testing until the last minute. You should be testing as you finish more functionality. **If you fail all of the autograder tests, you will receive a poor grade.**
- Do not rely on just the provided tests for the ALU. You must add more. The autograder will test your circuits extensively.

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.

- The ALU Control unit tests provided are probably enough
- 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. See the 4.3.1 case study.
- Be aware that running the tests will copy alu.circ and alu-control.circ into the tests/ directory. You **should not** modify those copies (e.g., tests/alu.circ) because you risk getting mixed up and losing work. You should only modify the copy of alu.circ and alu-control.circ that is in the base of your project files directory.
 - Notice that when you open one of the test circuits (e.g. tests/alu-add.circ, tests/alu-asr.circ), Logisim will have an additional folder in the component browser.



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

10 Where to get help

- **I don't know what alu.circ is supposed to do.**
 - Re-read the alu section
 - Refer to readings, knowledge check, and lecture on the ALU
 - 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/
- **I don't know what alu-control.circ is supposed to do.**
 - Re-read the ALU Control section
 - Refer to the textbook Appendix B to see the Op, Funct, and sh of instructions
 - Refer to the test cases

11 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 and will be reported to the college.

12 Acknowledgements

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