

CS2630: Computer Organization

Homework 5

Single-cycle ARM processor with I/O

1	Table of Contents	
2	<u>GOALS FOR THIS ASSIGNMENT</u>	<u>2</u>
3	<u>INTRODUCTION.....</u>	<u>2</u>
4	<u>TEAMS.....</u>	<u>3</u>
5	<u>READING THIS DOCUMENT.....</u>	<u>3</u>
6	<u>GETTING STARTED.....</u>	<u>4</u>
7	<u>THE PROCESSOR.....</u>	<u>5</u>
7.1	ONE OR MORE STUDENTS	6
7.2	TWO OR MORE STUDENTS.....	6
7.3	INSTRUCTION DETAILS.....	6
7.4	WHAT YOU MUST IMPLEMENT	7
7.5	TIPS ON BUILDING THE CONTROL UNIT	7
7.6	HOW YOU MUST TEST.....	7
8	<u>MAKING YOUR OWN TESTS</u>	<u>8</u>
8.1	CASE STUDY: ADDING A NEW TEST “MOV-SUB-REG”	8
8.1.1	WRITE AN ARM ASSEMBLY PROGRAM	8
8.1.2	ASSEMBLE THE ARM PROGRAM TO MACHINE CODE	8
8.1.3	CREATE A TEST CIRCUIT	9
8.1.4	SPECIFY EXPECTED OUTPUTS	10
8.1.5	RUN THE TEST AND DEBUG	10
8.2	GENERAL TESTING TIPS.....	10
9	<u>INPUT/OUTPUT (IO)</u>	<u>11</u>

9.1	WHAT YOU MUST DO	12
10	<u>REPORT: REFLECTING ON YOUR DEVELOPMENT PROCESS</u>	<u>13</u>
11	<u>REQUIREMENTS AND GRADING</u>	<u>13</u>
11.1	RUBRIC	13
11.2	EXAMPLE SCORES	13
11.3	ADDITIONAL REQUIREMENTS	14
11.4	SUBMISSION CHECKLIST.....	14
12	<u>RECOMMENDED APPROACH TO FINISHING THE PROJECT</u>	<u>15</u>
13	<u>HELP AND TIPS</u>	<u>16</u>
13.1	GENERAL TIPS	16
13.2	THE TEST FAILED, NOW WHAT?	16
13.3	TEAMWORK TIPS	18
13.4	WHERE TO GET HELP	18
14	<u>ACADEMIC HONESTY</u>	<u>18</u>
15	<u>ACKNOWLEDGEMENTS.....</u>	<u>19</u>

2 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 ARM programs that adequately test the processor

3 Introduction

In HW4 you built a major component of an ARM processor, and in HW5 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.

4 Teams

You can work either individually, as a team of 2, or as a team of 3. The number of students will determine the requirements of the project.

- 1 student: the basic set of instructions (see section: The Processor)
- 2 students: all of the above plus the additional instructions (see section: The Processor)
- 3 students: all of the above plus IO (see section: Input/Output (IO))

Feel free to do more than the requirements for your team size; however, there is no extra credit.

5 Reading this document

There is a lot here. I recommend thinking of the project in three components: 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. Read: ARM Processor (ICON) 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. Read: ARM Processor (ICON) 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. Read: ARM Processor (ICON) if you haven't yet
 4. *Assembling and running new programs*
 5. *Recommended approach to finishing the project*
 6. *Tips*
 7. *Teamwork Tips*
 8. *Where to get help*

6 Getting started

1. Download the starter code from

<https://github.com/bmyerz/project2-arm-processor/archive/refs/heads/proj2-part2-sp22.zip>

or, if you are using git, you can instead clone the repository. Just make sure checkout the branch proj2-part2-sp22. That is, do the following:

```
git clone https://github.com/bmyerz/project2-arm-processor.git
git checkout proj2-part2-sp22
```

- ## 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

[illegible]

FAILED test: B forward test (Error in the test)
Passed 0/5 tests

The x's indicate that those bits are disconnected.

3. Copy your alu.circ and alu-control.circ solution from HW4 into the new directory.

a. OR, to use the reference implementation of the ALU (optional)

You may choose to use **one or both** of alu.circ and alu-control.circ.

download ALU.jar from the HW5 ICON

download alu.circ and/or alu-control.circ from the HW5 ICON

You must put alu.circ and/or alu-control.circ in the **base directory** of your project (as it was in HW4).

You must have **two** copies of the ALU.jar file: one in the base directory and one in the tests/ directory.

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

```
make p1
```

```
cp alu.circ alu-control.circ regfile.circ cs3410.jar tests  
cd tests && python3 ./test.py p1 | tee ../TEST_LOG
```

Testing files...

```
PASSED test: ALU add (with overflow) test  
PASSED test: ALU arithmetic right shift test  
PASSED test: ALU Control Basic Data-processing Instructions Test  
PASSED test: ALU Control Shifts Test  
PASSED test: ALU Control CMP Test  
PASSED test: ALU Control B Test  
PASSED test: ALU Control ldr/str Test  
PASSED test: RegFile read/write test  
PASSED test: RegFile PC test  
PASSED test: RegFile debug outputs test
```

Passed 10/10 tests

6. When you open up cpu.circ in Logisim, you'll see that alu, mem, regfile, control, and alu-control are all available as subcircuits.

7 The processor

Your team's task is to design, implement, and test a single-cycle ARM processor. The processor must support a specific subset of instructions from the 32-bit ARM instruction set architecture. That subset is:

7.1 One or more students

Instruction	Versions required	
lsl	lsl rd, rm, shamt5	
lsr	lsr rd, rm, shamt5	
asr	asr rd, rm, shamt5	
add	add rd, rn, rm add rd, rn, #constant	
sub	add rd, rn, rm add rd, rn, #constant	
b		
and	and rd, rn, rm and rd, rn, #constant	
orr	orr rd, rn, rm orr rd, rn, #constant	
eor	eor rd, rn, rm eor rd, rn, #constant	
mov	mov rd, #constant	
mvn	mvn rd, rn	
ldr	ldr rd, [Rn, #constant]	positive constants only
str	str rd, [Rn, #constant]	positive constants only

7.2 Two or more students

Everything above, as well as:


- cmp, specifically the versions:
 - cmp Rn, Rm
 - cmp Rn, #constant
- conditional execution; only for these conditions
 - EQ
 - NE
 - GE
 - LT
 - GT
 - LE

7.3 Instruction details

The versions of each instruction you have to implement are shown above. For constants in data-processing instructions (all but ldr/str), you should eventually support using rotations (rot) of imm8 other than 0. But consider adding that functionality later on.

ldr and str only need to support Src2 being a constant, specifically a positive constant. And we are only supporting the index mode “offset”. Therefore, you don’t need to worry about I-bar, P, U, B, W; you only need to look at L.

7.4 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.

Note that the cpu.circ file has five output pins across the top. These provide hardwired access to five of the registers of the register file (R0, R1, R2, R14, R13). You should connect them directly to the five outputs at the top of the Register File we provided you.

7.5 Tips on building the control unit

Your control unit should go inside of **control.circ**.

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 probably want to use it on a subcircuit where you split the multi-bit wires into individual bits.
- Use "don't cares" to simplify the logic (the logic analyzer represents them as X's)
- Consider calculating control signals for each op separately (00=data processing, 10=branch, 01=load/store)
- For checking for a specific value on a multi-bit wire, an alternative to a truth table is a Decoder (under Plexers) or a Comparator (under Arithmetic).

7.6 How you must test

1. Run the tests with the command `make p2sc`.
2. See Section 13.2 for debugging your CPU when a test fails.
3. To ensure you pass the autograder, you **must test your CPU beyond the given tests**. See Section 8.

8 Making your own tests

In summary, making a test requires the following steps:

1. Write an ARM assembly program
2. Assemble the ARM program to machine code
3. Create a test circuit
4. Specify expected outputs
5. Run the test and debug

8.1 Case study: adding a new test “mov-sub-reg”

We’ll go through the entire process for a new test.

8.1.1 Write an ARM assembly program

Create a new text file in your tests/ directory named mov-sub-reg.s. It will be like mov-add-reg.s but with the sub instruction.

```
mov R0, #1
mov R1, #2
sub R2, R0, R1
```

8.1.2 Assemble the ARM program to machine code

8.1.2.1 Option A: Fully automatic assembly

This option requires an initial setup. But once you have it, assembly will require no manual work.

Setup

- [Download the ARM compiler](#), picking the appropriate platform. If you are on the Linux VMs (i.e., fastX) then you want x86_64 Linux. We’ll assume the file is in ~/Downloads and the version of the file was gcc-arm-none-eabi-10.3-2021.10.tar.bz2.
- On the command line
 - cd ~/Downloads
 - tar xvf gcc-arm-none-eabi-10.3-2021.10.tar.bz2
- Add the folder’s bin/ to your PATH environment variable. This will make the compiler commands available to your command line.
 - export PATH=\$PATH:\$HOME/Downloads/gcc-arm-none-eabi-10.3-2021.10/bin
 - Either run that “export” command every time you open a new terminal
 - Or, put that “export” command in your ~/.bashrc file

Assembling a file

- On the command line, go to your hw5 directory

- ./arm-assem.sh tests/mov-sub-reg.s
- You'll see the file mov-sub-reg.hex appear.

8.1.2.2 Option B: Partially automatic or Manual assembly

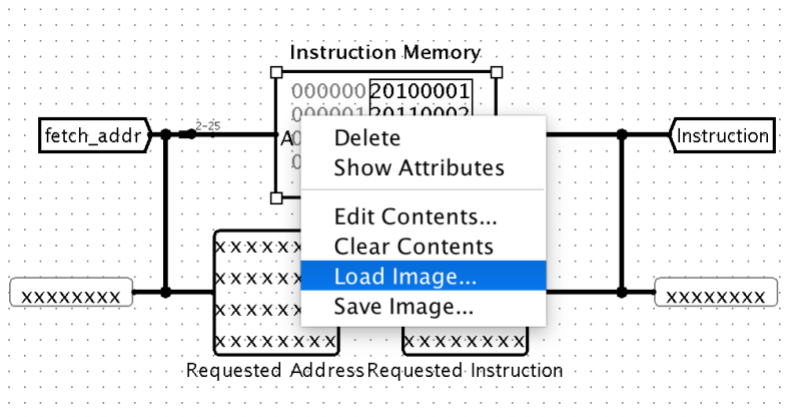
This option is to use ARMSim# like you did in Lab 7 on machine code. The slow part is that you'll have to copy the machine code out of ARMSim# one line at a time. Alternatively, use your skills and the textbook to translate your program by hand to machine code.

Once you've assembled the program in ARMSim#. Copy the machine code into a **plain text file** called mov-sub-reg.hex. Specifically, it should look like:

```
v2.0 raw
e3a00001
e3a01002
e0402001
```

8.1.3 Create a test circuit

1. Copy tests/cpu_test_template.circ to a file called tests/mov-sub-reg.circ. Open it in Logisim to edit.
2. Load the instruction memory by right-clicking the Instruction Memory ROM and choosing Load Image...



3. Choose the file mov-sub-reg.hex. You should see the ROM now has the machine code in it instead of 0's.
4. In general, you should also make sure the constant going into the halt comparator is **larger than** the number of clock periods your test program requires. It does not need to be exact.
5. Save and close the circuit.

Note: at this point you can actually test your CPU in Logisim manually by ticking the clock and looking at the register values. For automating the test, continue to the next step.

8.1.4 Specify expected outputs

1. Open tests/test.py in a text editor.
2. Add the following element to the p2sc_tests list

```
("MOV SUB register test",
    TestCase(os.path.join(file_locations, 'mov-sub-reg.circ'),
        [[0, 0, 0, 0x0, 0x0, 0, 0x0, 0xe3a00001],
         [1, 0, 0, 0x0, 0x0, 1, 0x4, 0xe3a01002],
         [1, 2, 0, 0x0, 0x0, 2, 0x8, 0xe0402001],
         [1, 2, -1, 0x0, 0x0, 3, 0xC, 0x00000000]]), "cpu"),
```

What does this code mean?

- "MOV SUB register test" is just a name for the test
- 'mov-sub-reg.circ' needs to be the name of your test circuit
- Each element of the list is the expected values of all output pins of the test circuit on every clock period
 - The order is 'R0 Value', 'R1 Value', 'R2 Value', 'LR (R14) Value', 'SP (R13) Value', 'Time Step', 'Fetch Addr', 'Instruction'
- "cpu" means the kind of test
- When do I use a given number base? It **doesn't matter** what base you choose!!! Python recognizes decimal or binary (0b prefix) or hex (0x prefix). The bit widths are set elsewhere (in decode_out.py) such that they match with the test circuit's outputs.

8.1.5 Run the test and debug

From the base directory of the project
make p2sc

See Section 13.2 for debugging your CPU when a test fails.

8.2 General 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	Template
------	----------------	----------------	----------

cpu	'R0 Value', 'R1 Value', 'R2 Value', 'LR (R14) Value', 'SP (R13) Value', 'Time Step', 'Fetch Addr', 'Instruction'	use for short/simple tests, where you want to check everything on every cycle	cpu_test_template.circ
cpu-lite	'R0 Value', 'R1 Value', 'R2 Value', 'LR (R14) Value', 'SP (R13) Value', 'Time Step'	use for tests where you don't want to have to check fetch address and instruction, but you still want to check the register values on every cycle	cpu_lite_test_template.circ
cpu-end	'R0 Value', 'R1 Value', 'R2 Value', 'SP (R13) Value'	use for tests where you only want to check the state of some registers when they change .	cpu_end_test_template.circ

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 changes. 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.

9 Input/output (IO)

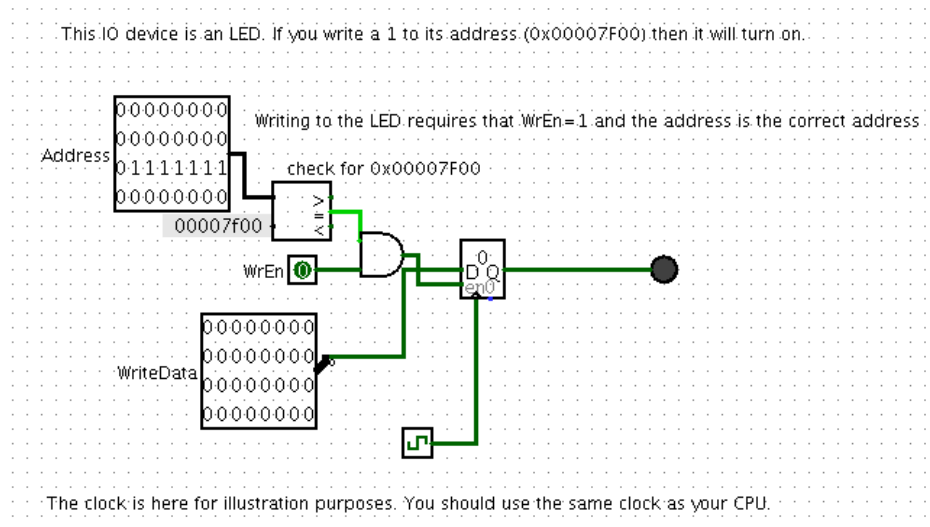
For groups of 3 only.

You now have a processor that executes real ARM 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).

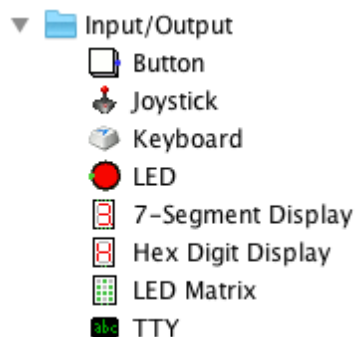
The way MMIO works is that some range of addresses is reserved for controlling IO devices rather than accessing memory.

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.

9.1 What you must do

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

- In `iodevices.circ`, build an IO controller for the device that you want to use and attach it to the appropriate signals in your `cpu`. The `iodevices.circ` **should not contain** its own clock; rather, it should have an input pin for the clock. **Your `iodevices` subcircuit should be placed in the top level of `cpu.circ` OR in a clearly labeled subcircuit within `cpu.circ`.** The device must be more than the single LED example (i.e., *at least* two LEDs but other devices are encouraged, too). **Write as short of an ARM test program as possible** that will ldr from the device (if input) or str to the

device (if output). Call the test program `iotest.s` and the test harness `iotest.circ`.

- a. Note that if you choose to use an input device (e.g., button, joystick) then you'll want to capture the input into a register. Your program can use `ldr` to read that register with MMIO.
- ii. Optionally, you may include a more interesting program that shows off your IO device. Include a test harness, as well as a brief description inside of your report.
- 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)

10 Report: reflecting on your development process

Complete the prompts in `hw5-report.docx`

11 Requirements and grading

11.1 Rubric

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

- 76 points – **Demonstrate your processor works correctly.** The processor passes the autograder tests (we have hidden tests that cover all required functionality). Passing one given test will get you ~53 points. Then you will earn further points for every additional test you pass.
- 29 points – **Demonstrate that you tested your processor.** You submitted test files to show you considered each instruction and corner cases. **You will earn 14.5 of these points for having just 1 new test** and the remainder for covering the rest of the instructions/cases.
 - For teams of 3: 65% of the 29 points are for the tests and 35% of the 29 points are for the I/O requirements
 - For each test, you must include the `.s` file, changes to `test.py`, and the `.circ` file!
- 45 points – **Report responses.** These will be graded on completeness, quality, thoughtfulness, clarity, and specificity. The ICON rubric has more detail.
 - **A complete report is required to earn credit on any of the project elements.**

11.2 Example scores

To make it really clear what your priorities should be, here are some examples of scores.

- 96.17% (A): approximately $\frac{3}{4}$ of tests passing (70.25/76), you wrote tests for all cases (29/29), you had a high-quality report (45/45)
- 92.3% (A-): approximately $\frac{1}{2}$ tests passing (64.5/76), you wrote tests for all cases (29/29), you had a high-quality report (45/45)

- 88.5% (B+): approximately ½ tests passing (58.75/76), you wrote tests for all cases (29/29), you had a high-quality report (45/45)
- 84.6% (B): **one test passing** (53/76) and **you wrote tests for all cases** (29/29), you had a high-quality report (45/45)
- 75% (C): **one test passing** (53/76) and **you wrote one test** (14.5/29), you had a high-quality report (45/45)
- 49.3% (F): no tests pass (0/76), wrote all the tests (29/29), had a high-quality report (45/45)
- 0% (F): all tests pass (76/76), wrote all the tests (29/29), wrote **no report** (report is required to receive any credit on the project)
- 0% (F): there's certainly lots of stuff in cpu.circ but **no tests pass** (0/76) and you wrote zero tests (0/29) and no report (0/45)

11.3 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.

You must make your circuits as legible as possible. Learn to make use of *tunnels* when they will save on messy wiring. (see [Documentation on Tunnels](#))

11.4 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 given tests you intended to pass
- ✓ Your circuits pass additional automated tests that you have written
- ✓ You made a zip file hw5.zip that contains these files in the following directory structure:
 1. cpu.circ (your completed CPU)
 2. control.circ (your completed control unit)
 3. iodevices.circ (your completed IO device(s), if appropriate)
 4. tests/
 - any additional files you've added for your testing, which means
 - your test harness files (i.e., the copies of cpu_test_template/cpu_lite_test_template/cpu_end_test_template), renamed appropriately, and with instruction memory loaded appropriately
 - the ARM assembly files that you used for your tests
 - tests.py (because you will add code to this file for your tests)
 - **Exclude**
 - Any auto-generated files such as the .hex files
- ✓ **Be responsible. Double-check and triple-check your zip file that it contains the correct versions of your files. Near the end of the semester we have no time for exceptions.**

1. Tip if you collaborating using GitHub or Gitlab, you can download a zip file of your repo when you are done.
- ✓ **As a team:** One submission by any team member for the team. You are responsible for the contents all being in there on time. Upload a zip file to ICON "Homework 5: AN ARM Processor".
 1. Make sure you complete the "HW5 team signup" on ICON before submitting your project.

12 Recommended approach to finishing the project

This project involves lots of implementation and testing (both circuits and ARM 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 ARM processor design, complexity was added incrementally). **During grading,**

1. **a CPU that passes many tests but is missing some instructions**

will be given far more credit than

2. **a CPU that passes 0 or a few tests but attempts to support all instructions**

Therefore, here is a general order we suggest

1. just enough to pass mov-i.s
2. just enough to pass mov-add-reg.s
3. just enough to pass mov-add-i.s
4. follow the directions to make the new test mov-sub-i.s; make changes to pass that one
5. write a complete first draft of your report
6. create and pass tests for other arithmetic/logical instructions
7. create and pass tests for arithmetic/logical instructions having a rot other than 0
8. just enough to pass branch-forward.s
9. create a pass tests for branch backwards
10. revise your report
11. (if team of 2+) implement just enough of conditional instructions to pass cond-add.s
12. create and pass tests for ldr/str
13. (if team of 2+) additional tests and support for conditional instructions
14. create and pass tests for remaining instructions and corner cases
15. (if team of 3) implement the IO device and program

13 Help and Tips

13.1 General tips

- If you open a file in Logisim, and Logisim prompts you saying it cannot find *.jar or *.circ then you should quit out of Logisim without continuing.
 - To fix the jar issue, make sure ALU.jar (if you are using our alu.circ and/or alu-control.circ) and cs3410.jar are in both the base directory and the tests/ directory.
 - To fix the circ issue, run the sanitize.sh command by typing “./sanitize.sh [FILENAME]” on the command line, from the base directory. For example, if when you open cpu.circ you get the prompt, then run “./sanitize.sh cpu.circ”.
- Be aware that running the tests will copy alu.circ, alu-control.circ, regfile.circ, cpu.circ, control.circ, mem.circ, and iodevices.circ into the tests/ directory. You **should not** modify those copies (tests/alu.circ, tests/alu-control.circ, tests/regfile.circ, tests/cpu.circ, test/control.circ, tests/mem.circ, tests/iodevices.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 all of the autograder tests, you will receive a failing 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).

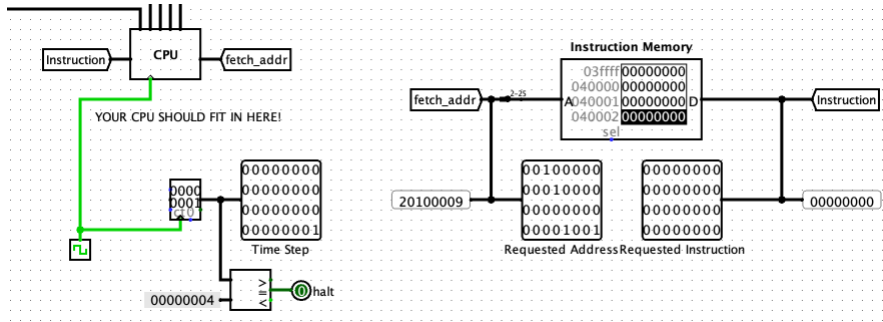
13.2 The test failed, now what?

We recommend a specific debugging process. It is similar to testing software: use a unit test to identify the failing test then use a debugger with a “breakpoint” to determine where the cause is.

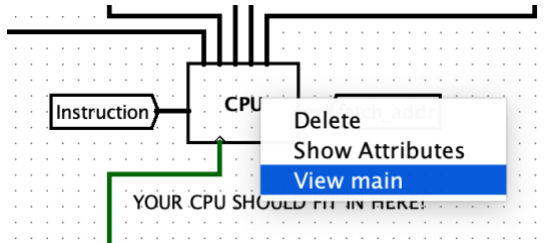
1. Determine the failing test. Do this by running make p2sc. Choose a single failing test to focus on. Within the output for that test, identify which Time Step is failing. For example, Time Step 1 is failing here.

```
cp alu.circ regfile.circ mem.circ cpu.circ control.circ iodevices.circ tests
cd tests && python3 ./test.py p2sc | tee ../TEST_LOG
Testing files...
Format is student then expected
R0 Value  R1 Value  R2 Value  LR (R14) Value  SP (R13) Value  Time Step  Fetch  AddrInstruction
0         0         0         0              0              20100001
0         0         0         0              0              20100001
0         0         0         0              1              20100009  0
1         0         0         0              1              4          20110002
FAILED test: starter test (1,2,3) (Did not match expected output)
```

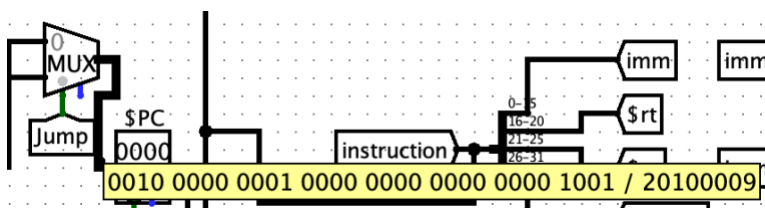

- When debugging, you always need to go top down. The top in our case, is the test harness. Open the circ file in tests/, for the failing test. In the running example, this is tests/starter_test.circ.
- Let's just first check that we can replicate the failing test. Tick the clock until the failing Time Step. In our case, it is 1, so we tick once. Indeed, the Fetch Addr is a huge number but it should have been 4.



- Since the CPU is sequential logic, values change at the rising edge of the clock. Therefore, the root cause of a failing test happened in the clock period BEFORE the failing Time Step. Reset the simulation and tick up to that Time Step. In this case, we failed on Time Step 1, so we want to be on Time Step 0.
- Now let's find the register whose value was wrong in Time Step 1, to see who wrote the wrong value to it in Time Step 0. In many cases, one of R0, R1, R2, R14, R13 will be wrong, which are registers in the Register File. But in our running example, the Fetch Addr was wrong so it is the PC register we want to focus on. To dive into the CPU, right click the CPU in the editor and click View Main.



- Find the register, and use the poke tool to see the values going into it. In our case of the PC, it looks like it is indeed getting the wrong value of 20100009.



- From here, continue tracing backwards to see where the erroneous value came from. In our case, we discovered we were giving the PC register the value of Instruction+4 instead of PC+4.
- You cannot edit the debugging mode version of your circuit!** Open your actual circuit file and make the changes there. In our case, it was cpu.circ that we needed to edit.

13.3 Teamwork tips

- Our recommendations are on assigning expertise to team members. However, you are producing one working product. The team is responsible for the project as a whole.
- Logisim circuits are hard to collaborate on unless you break them up into pieces. This is due to the .circ file format not being easily merged. **Therefore, we have provided a structure so that you can work in parallel:**
 - Test expert works on the test .circ files
 - Datapath expert works on the cpu.circ file
 - Control expert works on the control.circ file and alu-control.circ file
 - IO expert works on the iodevices.circ file
- 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! (Re-read the 2nd teamwork tip in this list)
- **Slip days:** refer to the syllabus

13.4 Where to get help

Go in this order:

1. Look for the answer in this document.
2. Refer to the readings and class materials. 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.

14 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 past and present submissions. Any academic dishonesty will result in a zero on this project and report of the incident to the College.

15 Acknowledgements

- original 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>