CS2630: Computer Organization
Project 2, part 2
Pipelined MIPS processor

Due Dec 3, 2016, 11:59pm

Goals for this assignment

- Design and implement a substantial digital system
- Employ pipelining in a digital design
- Add new instructions to the datapath and control
- Use robust testing methodology in digital logic design
- Create MIPS programs that adequately test the processor

Introduction

In project 2-1 you built two major components of a MIPS processor, and in project 2-2 you will build the rest of a processor. As in part 1, we provide the top-level skeleton file and test circuits, and you will provide the implementation and additional tests.

Getting started

1. Download the starter code from
https://github.com/bmyerz/proj3-starter/archive/proj2-part2.zip

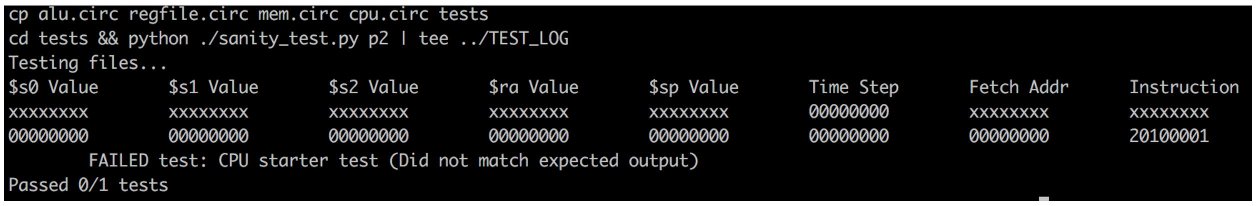
2. Try running the tests

Use the same method for running Linux commands that you used in part 1 of the project.

i. Run the tests

```
make p2
```

You should see output like



```
cp alu.circ regfile.circ mem.circ cpu.circ tests
cd tests && python ./sanity_test.py p2 | tee ../TEST_LOG
Testing files...
$s0 Value       $s1 Value       $s2 Value       $ra Value       $sp Value       Time Step       Fetch Addr      Instruction
xxxxxxxx        xxxxxxxx        xxxxxxxx        xxxxxxxx        xxxxxxxx        00000000        xxxxxxxx        xxxxxxxx
00000000        00000000        00000000        00000000        00000000        00000000        00000000        20100001
        FAILED test: CPU starter test (Did not match expected output)
Passed 0/1 tests
```

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

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

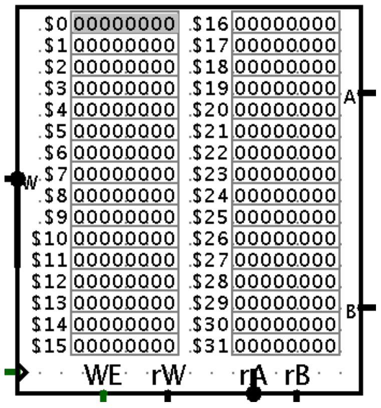
4. Alternatively, if you want to use the register file solution instead of your own regfile.circ, you may download

   regfile.circ: https://uiowa.instructure.com/files/2802038/download?download_frd=1

   To make that file work, you must also download the following jar file and put it in your tests/ folder.

   cs3410.jar: https://uiowa.instructure.com/files/2763043/download?download_frd=1

   The advantage to using the regfile.circ solution is that it uses a fancy register file that shows the values of the registers, which may make debugging a bit easier.



The processor

Your team's task is to design, implement, and test a 2-stage pipelined MIPS processor.

The processor must support a specific subset of instructions from the 32-bit MIPS instruction set architecture. That subset is

	Instruction
	sll
	srl
	sra
	add
	addu
	addiu
	addi
	jal
	jr
	j
	slt
	sltu
	sltiu
	slti
	and
	or
	andi
	ori
	lui
	lw
	sw
	beq
	bne
	bltzal

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

Instruction details

jal – Jump and Link – The reference sheet says to store PC+8 into `$ra`, but you must instead use PC+4. (Our architecture will not assume a jump delay slot; neither does MARS by default).

bltzal – Branch if less than zero and link – this instruction is not listed in the MIPS reference sheet, but MARS supports it as a core instruction.

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

You must use MARS to reverse engineer the bit encoding of this instruction (hint: it is I-type).

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.

How you must test

1. Run the tests with the command `make p2`.
2. To ensure you pass the autograder, you **must test your CPU beyond the given tests**. Adding new tests is similar to Project 2-1, except:
    - the sample test harness to copy is tests/CPU-starter_kit_test.circ instead of alu-harness.circ and regfile-harness.circ.
    - instead of loading the test inputs into RAMs, you will load the instruction memory with an assembled program (and optionally your data memory).

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

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

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

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

Pipelining

Your processor will have a **2-stage** pipeline:

1. **Instruction Fetch:** An instruction is fetched from the instruction memory.
2. **Execute:** The instruction is decoded, executed, and results written back. This stage is a combination of stages 2-5 of the 5-stage MIPS pipeline.

This design **has no data hazards**, since all accesses to all sources of data happens only in a single pipeline stage. However, **there are still control hazards** to deal with. Our ISA does not expose branch delay slots to software; that is, the instruction immediately after a branch or jump is not executed if the branch is taken.

Your processor should predict “branch not taken” so that stalls are not introduced by default. By the time you have figured out if you will take a branch or jump in the execute stage, you have already accessed the instruction memory and pulled out (possibly) the wrong instruction. You will therefore need to *kill* instructions that are being fetched if the instruction in Execute is a jump or a taken branch. Instruction kills for should be accomplished by sending a NOOP into the instruction stream rather than the fetched instruction. Notice that 0x00000000 is a NOOP instruction (R-type instruction that writes to `$zero`). You should only kill if a branch is taken (do not kill otherwise) and kill on every type of jump (since it is always taken).

Some things to consider when changing your single-cycle implementation into the 2-stage pipeline implementation:

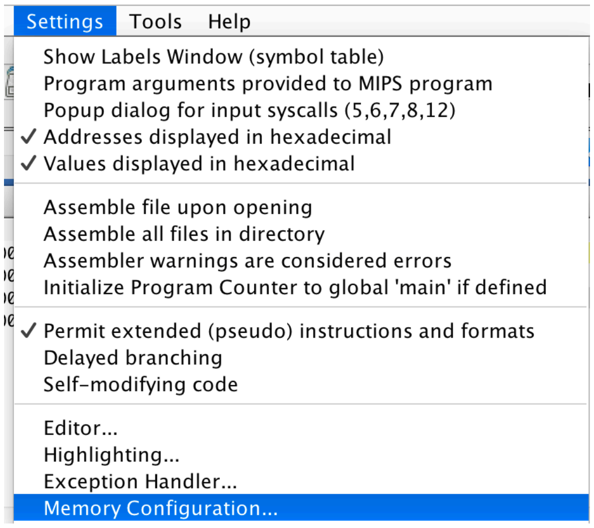
- Will the IF and EX stages have the same or different PC values?
- Do you need to store the PC between the pipelining stages?
- Where do you insert the NOOP into the instruction stream, before the register between stage 1 and 2 or after?
- What address should be requested next while the Execute stage executes a NOOP?

You might also notice a bootstrapping problem here: during the first cycle, the register sitting between the stages won't contain an instruction loaded from memory. How do we deal with this situation? We don’t have to! It happens that Logisim automatically sets registers to zero on reset; therefore, the pipeline register will then contain a NOOP. We will allow you to depend on this behavior of Logisim. Remember to go to Simulate --> Reset Simulation to reset your processor.

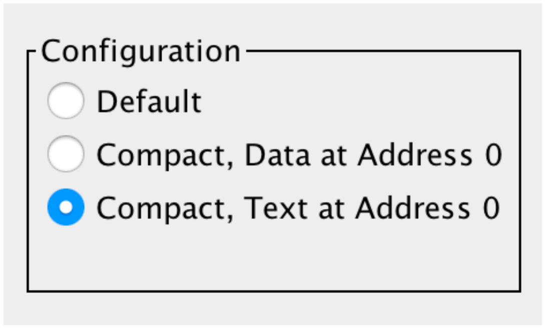
Assembling and running new programs

The project kit comes with a copy of Mars (mars.jar) so that you can assemble MIPS programs in the format required for the instruction memory. What follows is the workflow that we recommend for writing MIPS programs and running them on your processor.

1. Edit your MIPS program in MARS (as you did in HW2 and Project1).
   You should set the following “Memory Configuration…”, to tell MARS to assemble the addresses the same way our command line assembler does (.text starts at address 0x00000000 and .data starts at address 0x00002000)



Show Labels Window (symbol table)
Program arguments provided to MIPS program
Popup dialog for input syscalls (5,6,7,8,12)
✓ Addresses displayed in hexadecimal
✓ Values displayed in hexadecimal
Assemble file upon opening
Assemble all files in directory
Assembler warnings are considered errors
Initialize Program Counter to global 'main' if defined
✓ Permit extended (pseudo) instructions and formats
Delayed branching
Self-modifying code
Editor...
Highlighting...
Exception Handler...
Memory Configuration...



Configuration
Default
Compact, Data at Address 0
[✓ Compact, Text at Address 0]

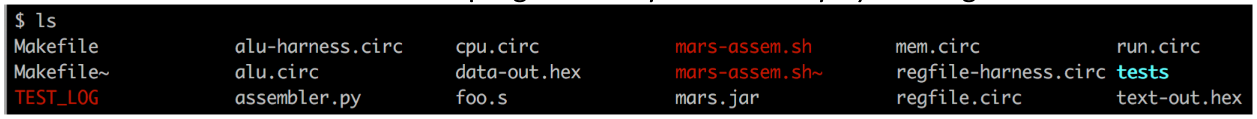
2. Test and debug your program in MARS (as you did in HW2 and Project1).
3. Save your MIPS program to a file. We'll assume the name "foo.s" for these directions, but you should name the file appropriately.
4. When you are ready to run your program on the MIPS processor, you will use the assembler provided with the project kit.

Make sure you know the file path of your MIPS file. It's easiest if you just save it to the project2-part2 folder.

i. Change directories to path of your project2-part2 folder

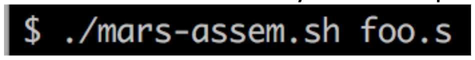
cd /path/to/project2-part2 (/path/to should be the actual file path)

ii. Double check that the MIPS program is in your directory by running `ls`.



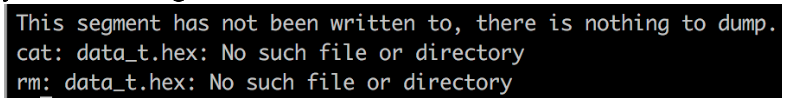
```
$ ls
Makefile          alu-harness.circ  cpu.circ          mars-assem.sh     mem.circ              run.circ
Makefile~         alu.circ          data-out.hex      mars-assem.sh~    regfile-harness.circ  tests
TEST_LOG          assembler.py      foo.s             mars.jar          regfile.circ          text-out.hex
```

iii. Run the assembler on your MIPS program



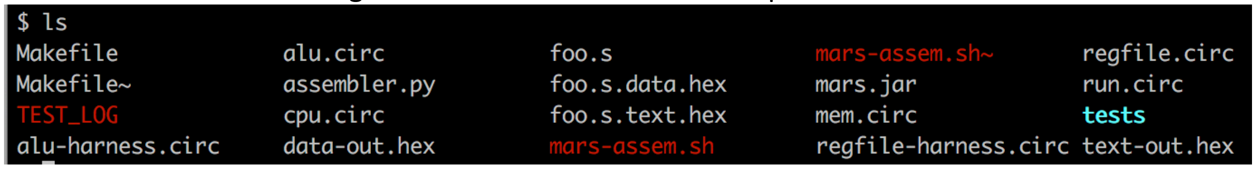
```
$ ./mars-assem.sh foo.s
```

If your assembly file didn't have a .data section you might see a message, but it is just a warning.



```
This segment has not been written to, there is nothing to dump.
cat: data_t.hex: No such file or directory
rm: data_t.hex: No such file or directory
```

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

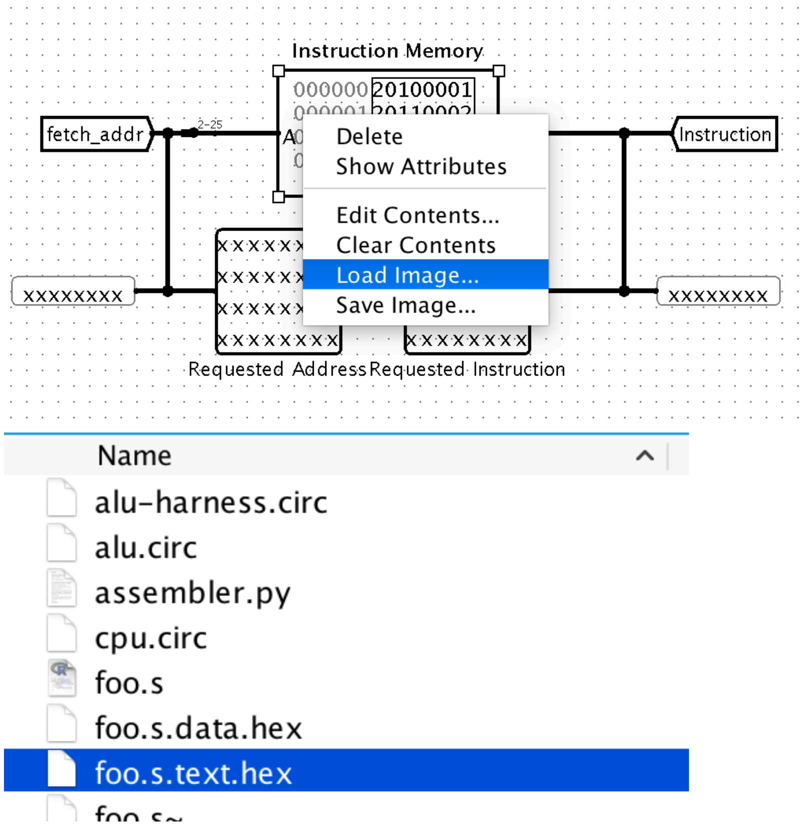


```
$ ls
Makefile          alu.circ          foo.s             mars-assem.sh~       regfile.circ
Makefile~         assembler.py      foo.s.data.hex    mars.jar             run.circ
TEST_LOG          cpu.circ          foo.s.text.hex    mem.circ             tests
alu-harness.circ  data-out.hex      mars-assem.sh     regfile-harness.circ text-out.hex
```

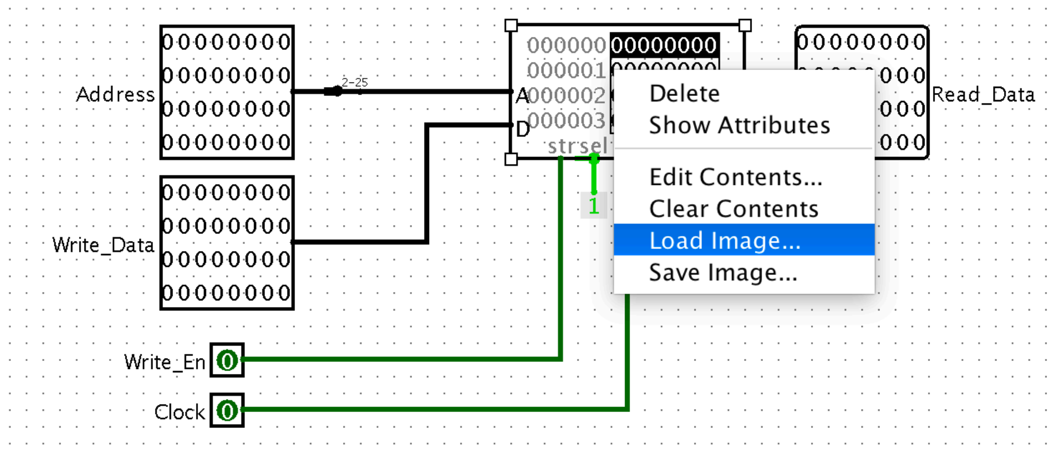
You should see a .text.hex file, which contains the text segment. If you had a .data section, you should also see a .data.hex file, which contains data memory contents **up to and including** the .data segment.

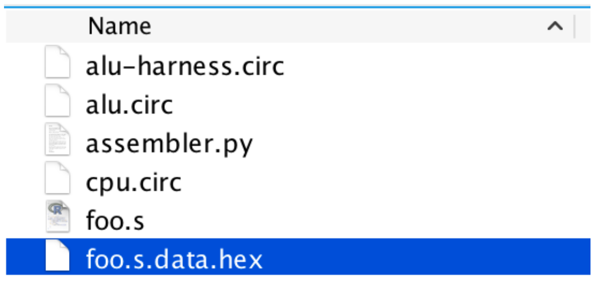
5. Now you can load the program into your processor in Logisim. **Make a copy of CPU-starter_kit_test.circ** and then open that new file in Logisim.

i. Load the instruction memory



ii. Load the data memory (OPTIONAL; only need to do this step if your MIPS program had a .data section). The data memory implementation is provided to you in mem.circ, which you should use as a “Logisim library…” in your CPU.





You can double-check that your data was loaded into the expected address in memory by right clicking the RAM > Edit Contents… > and scrolling down to the row for 002000 to see the data.

iii. Make sure to Save the circuit file containing the instruction memory so that you don’t have to load the program again (just have another copy of CPU-starter_kit_test.circ for each test program).

Note that you *will* have to load your *data memory* each time you change programs or “Reset Simulation”. Logisim applies the reset to RAMs but not ROMs.

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

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 p2` runs the tests without crashing
- ✓ Your circuits pass the original tests
- ✓ Your circuits pass additional automated tests that you have written
- ✓ You made a zip file `proj2-2.zip` that contains these files in the following directory structure:
  - cpu.circ (your completed CPU)
  - tests/
    - any additional files you've added for your testing.
    - Includes
      - your test harness files (i.e., the copies of CPU-starter_kit_test.circ, renamed appropriately, and with instruction memory loaded appropriately)
      - sanity_tests.py (because you will change this if you use our testing methodology)
      - .out files you've added to tests/reference_out
    - Excludes
      - the files we provided you *unless* you changed them
- ✓ Double-check your zip file
- ✓ **As a team**: Upload `proj2-2.zip` to ICON Project 2-2. One submission by any team member for the team. You are responsible for the contents all being in there on time.
- ✓ **Individually**: submit an entry to ICON "Project2-2 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.

Recommended approach to finishing the project

This project involves lots of implementation and testing (both circuits and MIPS code). **We highly recommended** that you get to a basic working processor quickly, which passes some simple tests with support for limited number of instructions and then add complexity from there. (Notice that in the textbook and lectures on MIPS processor design, complexity was added incrementally). **During grading,**

a) a CPU that passes several tests but is missing instructions or pipelining

will be given more credit than

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

Regarding pipelining, start by implementing a complete single-cycle version. Once it passes all your tests for all the instructions, then you can modify your design to the 2-stage pipeline.

Tips

- Do not waste your time writing the instruction memory hex files manually. You should be writing MIPS source programs in MARS. Leave the assembling to the assembler (See "Assembling and running new programs").
- Be aware that running the tests will copy alu.circ, regfile.circ, cpu.circ, and mem.circ into the tests/ directory. You **should not** modify those copies (tests/alu.circ, tests/regfile.circ, tests/cpu.circ, tests/mem.circ) because you risk getting mixed up and losing work.
- Do not leave testing until the last minute. You should be testing as you finish more functionality.
- Do not rely on just the provided tests. You must add more. The autograder will test your circuits extensively. **If you fail most of the autograder tests, you will receive a poor grade**.
- Do not rely solely on manually testing your circuits (i.e. poking inputs in the Logisim GUI and looking at the output). Manual testing is both time-consuming and error-prone. You should either extend the automated tests (as described in the testing sections of this document) or come up with your own automated testing approach.

Teamwork tips

- Although we do not require you and your team to use a version control system (e.g., git or svn), we highly recommend doing so to keep track of your changes. If you use version control just be aware that merging .circ files will corrupt them (unlike plain text files), so avoid working on the same file concurrently to avoid merge conflicts all together. Ask the staff for help if you get stuck!
- Logisim circuits are hard to collaborate on unless you break them up into pieces. Therefore, you should break up the work among the group members. Some ways to break up the work are: different members work on different sub-circuits, designs and truth tables, and test cases/MIPS programs.
- **Slip days:** to use a slip day, 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 3.2 http://www-inst.eecs.berkeley.edu/~cs61c/sp16/
- "Pipelining" specification adapted from the above source
- Helper library register file from Cornell CS3410, Spring 2015 http://www.cs.cornell.edu/courses/cs3410/2015sp