CS61C Summer 2015 Project 3-2: CPU

TA: Rebecca Herman

Due Monday, August 3rd, 2015 @ 11:59 PM

IMPORTANT INFO - PLEASE READ

Even though this project is broken into two parts, the second part is substantially more involved than the first. Do not procrastinate! Feel free to keep improving your ALU and Regfile for part 2.

- **MAKE SURE TO CHECK YOUR CIRCUITS WITH THE GIVEN HARNESSES TO SEE IF THEY FIT! YOU WILL FAIL ALL OUR TESTS IF THEY DO NOT.**
  (This also means that you should not be moving around given **inputs** and **outputs** in the circuits).
- Sample tests for a completed ALU and Regfile have been included in the proj3-1StartKit. Given the current directory structure, you can run the bash script (`short-test.sh`) with your `*.circ` files in the same directory and it will run the tests. We recommend running the sample tests locally, but they only work with **python 2.7**. These tests are NOT comprehensive, you will need to do further testing on your own.
- You are allowed to use any of Logisim's built-in blocks for all parts of this project.
- **Save often.** Logism can be buggy and the last thing you want is to lose some of your hard work. There are students every semester who have had to start over large chunks of their projects due to this.
- Approach this project like you would any coding assignment: construct it piece by piece and test each component early and often!
- **Tidyness and readability will be a large factor in grading your circuit if there are any issues, so please make it as neat as possible! If we can't comprehend your circuit, you will probably receive no partial credit.**

This project is very long, **but don't fret!** Many things are spelled out in incredible detail, so if you just take things one-by-one, it won't be as bad as it looks.

Updates | Overview | Deliverables | ISA | Logisim | Testing | Submission

Updates and Clarifications

- Your memory file should be BIG endian, not little endian. You're welcome ;)
- On jr: Unfortunately the test we provided is assuming that the value you've stored in the register is already word-addressed. This would actually make it impossible to properly run code that you've compiled yourself using your project 2 if it contains a jr instruction. But since we've had too many updates already, here's what you should do for jr: treat the value as word-addressed (no need to divide by four), but still take care of the offset yourself!

Updates | Overview | Deliverables | ISA | Logisim | Testing | Submission

Overview: A Reminder

In this project you will be using Logisim to implement a simple 32-bit two-cycle processor. Throughout the implementation of this project, we'll be making design choices that make it compatible with machine code outputs from MARS and your Project 2! However, there are some key differences between the processor we studied in class and the processor you will be designing for this project, so you will have to be careful to make sure that MIPS as it is usually written will operate properly on your processor.

In Part II of the project, you will complete your pipelined processor! Please read this document *CAREFULLY* as there are key differences between the processor we studied in class and the processor you will be designing for this project.

Updates | Overview | Deliverables | ISA | Logisim | Testing | Submission

Deliverables: Project 3-2

3) Obtaining the Files - Part 2 [show]

4) Processor [show]

5) Data Memory [show]

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 committed (written back). This is a combination of the remaining stages of a normal MIPS pipeline.

Your processor controller will be responsible for the following:

1. Providing controllers for Regfile, ALU, and other CPU logic
2. Killing the instructions following a jumps and a branches (if the branch is taken) by muxing a the instruction with a nop ("no operation")
3. Stalling the pipeline (not killing it!) while mulad and divsubu are calculating. **Note: the stall in the pipeline should begin as the new values of A and B are loaded into their registers in the ALU, and should stay until the results are safely stored in Lo and Hi!** We will not be particular about the exact number of stalls, so long as you implemented mulad and divsubu using addition and subtraction (there must be at least B stalls for mulad, and A/B stalls for divsubu), and execute the instructions which follow correctly.

You should note that data hazards do NOT pose a problem for this design, 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. This means that the instruction immediately after a branch or jump is not necessarily executed if the branch is taken. This makes your task a bit more complex. By the time you have figured out that a branch or jump is 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 under execution is a jump or a taken branch. Instruction kills for this project MUST be accomplished by MUXing a `nop` into the instruction stream and sending the `nop` into the Execute stage instead of using the fetched instruction. Notice that 0x0000 is a `nop` instruction; please use this, as it will simplify grading and testing. You should only kill if a branch is taken (do not kill otherwise), but do kill on every type of jump.

Because all of the control and execution is handled in the Execute stage, **your processor should be more or less indistinguishable from a single-cycle implementation, barring the one-cycle startup latency and the branch/jump delays.** However, we will be enforcing the two-pipeline design. If you are unsure about pipelining, it is perfectly fine (maybe even recommended) to first implement a single-cycle processor. This will allow you to first verify that your instruction decoding, control signals, arithmetic operations, and memory accesses are all working properly. From a single-cycle processor you can then split off the Instruction Fetch stage with a few additions and a few logical tweaks. Some things to consider:

- Will the IF and EX stages have the same or different `PC` values?
- Do you need to store the `PC` between the pipelining stages?
- To MUX a `nop` into the instruction stream, do you place it *before* or *after* the instruction register?
- What address should be requested next while the EX stage executes a `nop`? Is this different than normal?

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

Instruction Set Architecture (ISA)

The instructions you will implement are listed below. In all of the instructions you recognize from MIPS, the instruction format, opcode, funct, and register numbers should be taken directly from your greensheet. Just as we discussed in class, our processor will pull out a 32-bit value from instruction memory and determine the meaning of that instruction by looking at the `opcode` (the top 6 bits, which are bits 31-26). If the instruction is an R-type (i.e. `opcode == 0`), then you must also look at the `funct` field.

Notice how we do not use all the instructions in MIPS. Your project only has to work on these specified instructions, most of which you should have seen. There are two new instructions as well, which are explained below the table! We have taken out `sb` to simplify your memory file.

INSTRUCTION	FORMAT
Add	`add $rd, $rs, $rt`
Add Unsigned	`addu $rd, $rs, $rt`
Sub	`sub $rd, $rs, $rt`
Sub Unsigned	`subu $rd, $rs, $rt`
And	`and $rd, $rs, $rt`

Or	`or $rd, $rs, $rt`
Set Less Than	`slt $rd, $rs, $rt`
Set Less Than Unsigned	`sltu $rd, $rs, $rt`
Jump Register	`jr $rs`
Shift Left Logical	`sll $rd, $rt, shamt`
Shift Right Logical	`srl $rd, $rt, shamt`
Shift Right Arithmetic	`sra $rd, $rt, shamt`
Add Immediate Unsigned	`addiu $rt, $rs, immediate`
And Immediate	`andi $rt, $rs, immediate`
Or Immediate	`ori $rt, $rs, immediate`
Load Upper Immediate	`lui $rt, immediate`
Load Byte	`lb $rt, offset($rs)`
Load Byte Unsigned	`lbu $rt, offset($rs)`
Load Word	`lw $rt, offset($rs)`
Store Word	`sw $rt, offset($rs)`
Branch on Equal	`beq $rs, $rt, label`
Branch on Not Equal	`bne $rs, $rt, label`
Jump	`j label`
Jump and Link	`jal label`
Move from Lo	`mflo $rd`
Move from Hi	`mfhi $rd`
Multiply by Addition	`mulad $rs, $rt`
Divide by Subtraction Unsigned	`divsubu $rs, $rt`

Mulad and Divsubu Specifications

Mulad: Multiply by Addition. You should have implemented this in lab 7! More details:

- It is an R-type instruction with a funct code of 62.
- It multiplies the values in `$rs` and `$rt` by repeatedly adding `$rs`, `$rt` # of times.
- For example: 3*4 = 3 + 3 + 3 + 3, we added +3 repeatedly four times.
- The upper 32 bits of the product will be ignored, and the lower 32 bits will be stored in Lo. **It should NOT write to Hi.**
- It will take approximately `R[$rt] + 1` clock cycles to complete this instruction (we will not be strict on the exact number, as long as it's more than `R[$rt]` and withing a few clock cycles of the recommended value).
- Should capture values of A, B and opcode so that long computation can continue despite the pipeline.
- Outputs a stall signal from the clock cycle that grabs the data until the appropriate instruction is done executing.

Divsubu: Divide by Subtraction Unsigned. It is quite similar to Mulad:

- It is an R-type instruction with a funct code of 63.
- It divides the unsigned value in `$rs` by the unsigned value in `$rt` by repeatedly subtracting `$rt` from `$rs` until `R[$rs] < R[$rt]` while keeping track of the number of times it does so.
- For example: 7/3 -> 7 - 3 - 3 -> 2 r1.
- It stores the quotient in Lo, and the remainder in Hi.
- It will take `R[$rs]/R[$rt] + 1` clock cycles to complete this instruction (we will not be strict on the exact number, as long as it's close to `R[rs]/R[rt]`).
- Should capture values of A, B and opcode so that long computation can continue despite the pipeline.
- Outputs a stall signal from the clock cycle that grabs the data until the appropriate instruction is done executing.

Jumping

- The target address in a jump instruction in normal MIPS is pseudodirect addressing, because it can only hold 26 of the 32 bits in an address. However addresses in our memory file will only be 24 bits long (again, due to limitations of logisim). This means that we can't actually access many of the addresses specified in any given MIPS program. Because our memory is word-addressed, we will not need to concatenate any zeros to the bottom of our addresses. But this still leaves us with 30 bit address. To reduce to 24 bit addresses, we will ignore the four bits that you would usually take from `PC+4` (It would be `PC+1` in our architecture, since we are word-addressed), and we will also have to ignore the first two bits in the immediate of the jump instruction. You may assume that the programs we are given will not need to store so much data that this would become a problem.

- Remember that MARS will represent absolute addresses of the `.text` section starting from a base address of `0x00400000`, **byte-addressed**. However, your instruction memory starts this section at `0x000000`, **word-addressed**, so make sure you account for this offset while calculating your address for **jumps (j, jr, jal)**.
- Note that you should kill the next instruction after a `jump`, `jr`, or `jal` even if that is the instruction you are going to be jumping to.
- On a `jal` the address of the next instruction should be written into `$ra` just like in MIPS. **Don't forget to add the offset to the address of the next instruction!**

Branching

- The argument to the `beq` and `bne` instructions is a **signed** offset relative to the next instruction to be executed if we don't take the branch, which is similar to MIPS. Note that the address of this next instruction is `PC+1` rather than `PC+4` because our processor is word-addressed. Here, `currPC` means the address *of the branch instruction*. We can write `beq` as the following:

```
 if $rs == $rt
      nextPC = currPC+1 + offset
  else
      increment PC like normal
```

- Think! There's a reason we write "`increment PC like normal`" here instead of just "`currPC+1`".
- The `bne` instruction differs only by the conditional in the `if` statement: replace the `==` with `!=`.
- You should not create a bubble while executing a branch instruction: you should load the following instruction and kill it if the branch is not taken. Note that you should not kill the next instruction if the branch is not taken. If the branch is taken you should always kill the instruction.

Immediates

- Note that the `immediate` field is only 16 bits wide, so we must perform some kind of extension on it before passing it to the ALU.If an immediate is supposeed to be **unsigned**, be sure to **zero-extend** it. If an immediate is **signed**, be sure to **sign-extend** it. This should be the same specifications as on the MIPS green sheet.

Updates | Overview | Deliverables | ISA | Logisim | Testing | Submission

---

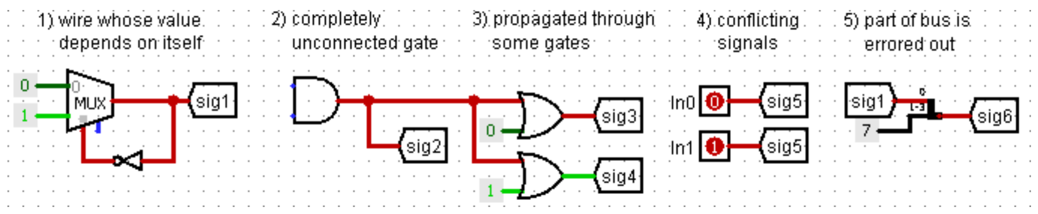
Logisim Notes

While you may use Logisim 2.7.1 for developing your `alu.circ`, `regfile.circ`, `mem.circ`, and `cpu.circ`, do note that you have to open `run.circ` with the MIPS-logisim file we provided.

If you are having trouble with Logisim, ***RESTART IT and RELOAD your circuit!*** Don't waste your time chasing a bug that is not your fault. However, if restarting doesn't solve the problem, it is more likely that the bug is a flaw in your project. Please post to Piazza about any crazy bugs that you find and we will investigate.

Things to Look Out For

- Do **NOT** gate the clock! This is very bad design practice when making real circuits, so we will discourage you from doing this by heavily penalizing your project if you gate your clock.
- **BE CAREFUL with copying and pasting from different Logisim windows.** Logisim has been known to have trouble with this in the past.
- When you import another file (Project --> Load Library --> Logisim Library...), it will appear as a folder in the left-hand viewing pane. The skeleton files should have already imported necessary files.
- Changing attributes *before* placing a component changes the default settings for that component. So if you are about to place many 32-bit pins, this might be desireable. If you only want to change that particular component, place it first before changing the attributes.
- When you change the inputs & outputs of a sub-circuit that you have already placed in `main`, Logisim will automatically add/remove the ports when you return to `main` and this sometimes shifts the block itself. If there were wires attached, Logisim will do its automatic moving of these as well, which can be extremely dumb in some cases. Before you change the inputs and outputs of a block, it can sometimes be easier to first disconnect all wires from it.
- Error signals (red wires) are obviously bad, but they tend to appear in complicated wiring jobs such as the one you will be implementing here. It's good to be aware of the common causes while debugging:


1) wire whose value depends on itself
2) completely unconnected gate
3) propagated through some gates
4) conflicting signals
5) part of bus is errored out
MUX
sig1
sig2
sig3
sig4
In0
In1
sig5
sig5
sig1
sig6
7

Logisim's Combinational Analysis Feature

Logisim offers some functionality for automating circuit implementation given a truth table, or vice versa. Though not disallowed (enforcing such a requirement is impractical), use of this feature is discouraged. Remember that you will not be allowed to have a laptop running Logisim on the final.

Updates | Overview | Deliverables | ISA | Logisim | Testing | Submission

---

Testing

Part 2

To test your CPU, we have provided a couple tests in the directory `test-files/assem/`. To run the tests for the CPU, run

```
make cpu
```

This will run the test and dump a log as `TEST_LOG`. To view this log, you can open it in any text editor or run

```
cat TEST_LOG
```

Since the log has long lines, it's a good idea to zoom out so you can view everything. The log compares the student implementation versus the reference; if there is a difference it will print out `***` in the first column. Note that the logfile is overwritten every time you run a test. As mentioned before in the ALU section, you can also test the capturing of the funct input in the ALU by running

```
make alu
```

Alternatively, to test your CPU, you can open `run.circ`. **Note that you have to open `run.circ` with the MIPS-logisim.jar file we provided.** Find the Instruction Memory RAM and right click --> Load Image... Select the assembled program (`.hex` file in `test-files/hex/` or see details on the Assembler below) to load it and then start clock ticks.

We are not requiring you to write and submit your own tests for points. However, the autograder tests are much more detailed than the tests we provide you, and additional testing on your own is recommended! **Remember:** Debugging Sucks. Testing Rocks.

Assembler

We've provided a basic assembler to make writing your programs easier so you can use assembly instead of machine code. You should try writing a few by hand before using this, mainly because it's good practice and makes you feel cooler. This assembler.py supports all of the instructions for your processor.

The assembler is included in the start kit (one you pull from the repo with earlier instruction) or can be downloaded from the link above. The standard assembler is a work in progress, so please report bugs to Piazza!

The assembler takes files of the following form (this is `halt.s`, which is included in the start kit):

```
        #Comments are great!
            lui $t0, 0x3333          #3c083333
            ori $t0, $t0, 0x4444     #35084444
            lui $t1, 0x3333          #3c093333
            ori $t1, $t1, 0x4444     #35294444
      self: beq $t0, $t1, self       #1109ffff
```

Commas are optional but the '`$`' is not. '`#`' starts a comment. The assembler can be invoked with the following command:

```
   $ python assembler.py input.s [-o output.hex]
```

The output file is `input.hex` if not explicitly set - that is, the same name as the input file but with a `.hex` extension. Use the `-o` option to change the output file name arbitrarily.

As an alternative to the assembler.py, you can also use MARS command line utilities to assemble your file. This will also allow you to create `.hex` files for your memory, although it won't assemble the new instructions we added to your processor. You can look at this link for specifics, but a sample script has been written in `mars-assem.sh`.

In addition, you are welcome to use your project 2 assembler and linker to create these .hex file! Try it out and marvel at having created 3/4th of the CALL process. Although, be wary of bugs in your project 2.

Updates | Overview | Deliverables | ISA | Logisim | Testing | Submission

Submission: Proj3-2

There are **two** steps required to submit proj3-2. Failure to perform both steps will result in loss of credit:

1. First, you must submit using the standard unix submit program on the instructional servers. This assumes that you followed the earlier instructions and did all of your work inside of your `git` repository. To submit, follow these instructions after logging into your cs61c-XX class account:

   ```
   cd ~/proj2-XX-YY                             # Or where your shared git repo is
   submit proj3-2
   ```

   Once you type `submit proj3-2`, follow the prompts generated by the submission system. It will tell you when your submission has been successful and you can confirm this by looking at the output of `glookup -t`.

2. Additionally, you must submit proj3-2 to your **shared** GitHub repository:

   ```
   cd ~/proj3-XX-YY                             # Or where your shared git repo is
   git add -u
   git commit -m "project 3-2 submission"
   git tag "proj3-2-sub"                        # The tag MUST be "proj3-2-sub". Failure to do so will result in loss of credit.
   git push origin proj3-2-sub                  # This tells git to push the commit tagged proj3-2-sub
   ```

Resubmitting

If you need to re-submit, you can follow the same set of steps that you would if you were submitting for the first time, but you will need to use the `-f` flag to tag and push to GitHub:

```
# Do everything as above until you get to tagging
git tag -f "proj3-1-sub"
git push -f origin proj3-1-sub
```

Note that in general, force pushes should be used with caution. They will overwrite your remote repository with information from your local copy. As long as you have not damaged your local copy in any way, this will be fine.

Deliverables

```
cpu.circ
mem.circ
regfile.circ
alu.circ
```

We will be using our own versions of the `*-harness.circ` files, so you do not need to submit those. In addition, you should not depend on any changes you make to those files.

You must also submit any `.circ` files that you use in your solution (they are not copied into your `.circ` file when you import them, only referenced). Make sure you submit every `.circ` file that is part of your project! You might want to test your `cpu.circ` file on the lab machines before you submit it, to make sure you got everything.

Grading

This project will be graded in large part by an autograder. Readers will also glance at your circuits. If some of your tests fail the readers will look to see if there is a simple wiring problem. If they can find one, they will give you the new score from the autograder minus a deduction based on the severity of the wiring problem. For this reason, neatness is a small part of your grade - please try to make your circuits neat and readable.

Updates | Overview | Deliverables | ISA | Logisim | Testing | Submission