CS311 Project 3: Building a Simple MIPS Simulator

Due 11:59pm, November 16th, 2017
TA: Chang Hyun Park

1. Overview

This third project is to build a 5-stage pipelined simulator of a subset of the MIPS instruction set. The simulator loads a MIPS binary into a simulated memory, and executes the instructions. Instruction execution will change the states of registers and memory. The simulator will need to detect and behave accordingly for any data or control hazards. Please also read the README.md file provided in the repository.

If you have any questions related to the project, please ask them on the Q&A board. The assigned TA will answer them whenever possible.

2. Required Pipeline Implementation

In this project you will need to implement the 5 stage MIPS pipeline that you have learned in class into your MIPS simulator (from Project 2).

2.1 Pipeline Stages

We will use a simple 5-stage pipeline for this project:

1. IF: fetch a new instruction from memory.
2. ID: decode the fetched instruction and read the register file
3. EX: execute an ALU operation
    a. Execute arithmetic and logical operations
    b. Calculate the addresses for loads and stores
4. MEM: access memory for load and store operations
5. WB: write back the result to the register

Between the adjacent pipeline stages, pipeline registers (or pipeline latches) must be modeled. All communication between stages must be conducted using the pipeline registers.

2.2 Register File

The register file is used in both the ID stage and the WB stage, however the WB writes the register file at the first half of a cycle and ID reads on the second half of a cycle resulting in no structural hazards in the register file.

2.3 Memory Model

We assume an ideal dual-ported memory, which allows both a read at IF, and a read/write at MEM simultaneously within the same cycle. There is no structural hazard for the memory accesses from IF and MEM. You can assume the memory locations where instructions are stored are never updated. Therefore, you do not need to consider the case where a store at MEM updates the same address fetched at IF at the same cycle.

2.4 Forwarding

The pipelined architecture must support data forwarding from MEM/WB-to-EX, EX/MEM-to-EX. With the forwarding support, data hazard only occurs for the data dependency from a load to the

succeeding instruction which uses the value in the EX stage. MEM/WB-to-MEM is also supported.

2.5 Control Hazard

For unconditional jumps, you will always add a one-cycle stall to the pipeline (J, JAL, JR). For JAL, assume there is a single delay slot after JAL instruction. Therefore, when calling the JAL instruction save the PC+8 value into the R31 (as with project 2). In this project, the binaries and assembly files have been modified to have a nop instruction (in the form of add $0, $0, $0) right after every JAL instructions.
For the sake of simplicity, don't bother executing the delay slot. When the jump decision is made from the ID stage, just flush the IF stage (to skip executing the delay slot).

For conditional branches (BEQ, BNE), your simulator must support the static branch predictor which always predicts branches **not taken**. The actual evaluation of the branch will be executed by the ALU and thus the branch result will be ready at the end of the EX stage. However, **the actual flushing will take place at the beginning of the MEM stage.**. A correct branch (not taken) will not incur any stalls; a miss prediction will cause 3 cycle stall (flushing of the EX, IF, and ID of the newer instructions). Also, if the branch decision has dependencies on prior executions, stall the execution at the appropriate stage.

2.6 Stopping the Pipeline

The simulator must stop after a give number of instructions finishes the WB stage. At cycle 1, the first instruction is fetched from the memory. If the last instruction is in the WB stage at cycle N, the final CYCLE count is N.

3. Pipeline Register States

You need to add pipeline register states between stages. The followings are possible register contents, but you need to add more states.

IF_ID.Instr : 32-bit instruction
IF_ID.NPC : 32-bit next PC (PC+4)
ID_EX.NPC : 32-bit next PC
ID_EX.REG1 : REG1 value
ID_EX.REG2 : REG2 value
ID_EX.IMM : Immediate value
EX_MEM.ALU_OUT: ALU output
EX_MEM.BR_TARGET: Branch target address
MEM_WB.ALU_OUT: ALU output
MEM_WB.MEM_OUT: memory output

You must carefully design what fields are necessary for each pipeline register, and explain them in "README" file in your submission. You must explain what each field means.

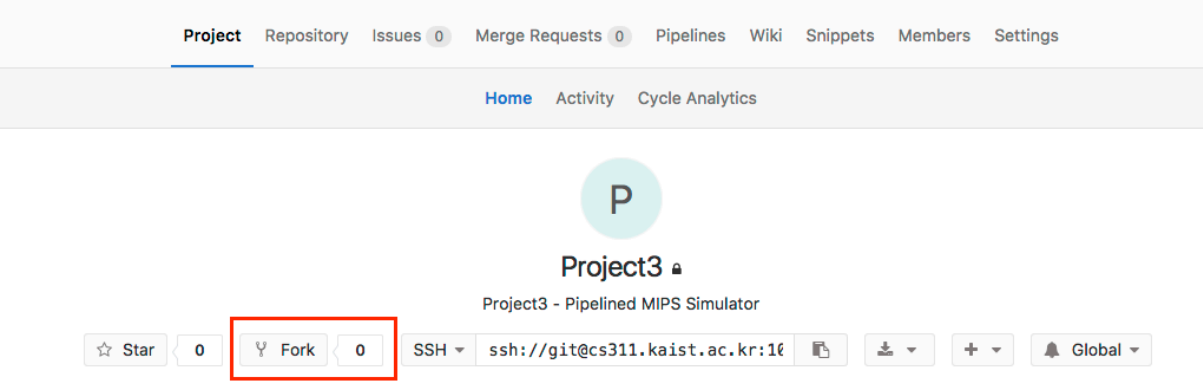
4. Forking and Cloning your Repository

As with Project 1, we will fork the TA's Project2 repo to your team namespace. Then you will clone your team's repo into your local machines to work on the project.

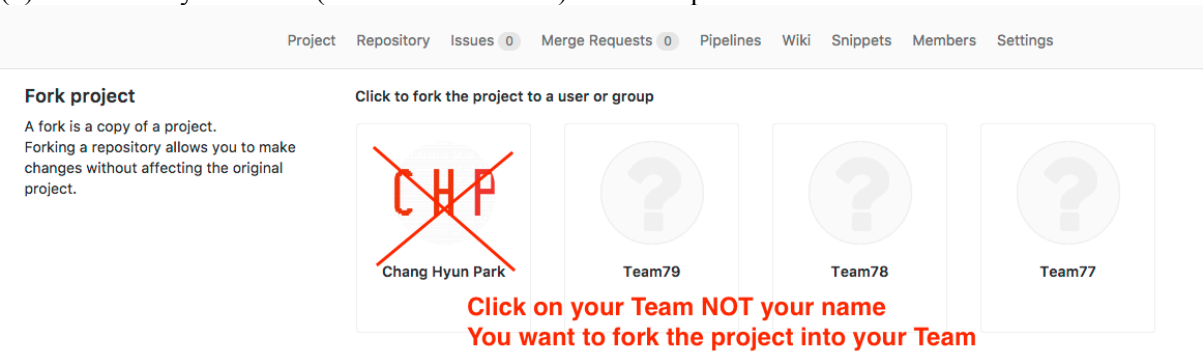
4.1 Forking the TA's Repo

(1) Go to the following page: https://cs311.kaist.ac.kr/TAs/Project3. The page is the TA's repository.

(2) Click the fork button just like Project 1.



(3) Now select your Team (NOT YOUR USER) and the repo will be forked.



(4) Your Team repo will have the following URL: https://cs311.kaist.ac.kr/Team[your number]/Project3
NOTE: We will be running automated scripts to download your work and grade your projects. **Please do not change the name of your project paths**. (Keep the project name & path as Project3)

4.2 Cloning the team repository to your local machine

From the website of your team repo copy the SSH or HTTPS URL of the git repository
the SSH URL will look something like the following:
ssh://git@cs311.kaist.ac.kr:10022/Team[your number]/Project3.git

Change directory to the location you want to clone your project and clone!
$ git clone ssh://git@cs311.kaist.ac.kr:10022/Team[your number]/Project3.git

Be sure to read the README.md file for some useful information.

5. Simulator Options and Output

5.1 Options

$ ./cs311sim [-m addr1:addr2] [-d] [-n num_instr] inputBinary

- -m : Dump the memory content between addr1 to addr2
- -d : Print the register file content every cycle. Prints memory content every cycle if –m option is enabled.
- -n : number of instructions simulated
- -p : print the PCs of of the instructions in each pipeline stage at every cycle. Prints a blank if the stage of the pipeline is stalled/empty.
  ex) CYCLE 5:0x00400010|0x0040000c|0x00400008|0x00400004|0x00400000

5.2 Reference output

We will be providing a reference output (located in the sample_output directory) so that you may compare the execution of your simulator to the reference.

The TA's answer *may* contain some incorrect executions. If you do find any upon your debugging and comparing, please do drop us an e-mail of the expected behavior, and the actual behavior shown by the TA's answer. We'll check it out and push fixed reference outputs. Afterwards, you may update the reference outputs using the instructions in Section 9.1.

6. Grading Policy

Grades will be given based on the examples provided for this project provided in the `sample_input` directory. Your simulator should print the exactly same output as the files in the `sample_output` directory.

We will be automating the grading procedure by seeing if there are any difference between the files in the `sample_output` directory and the result of your simulator executions.
Please make sure that your outputs are identical to the files in the sample_output directory.

You are encouraged to use the *`diff`* command to compare your outputs to the provided outputs.

```
$ ./cs311sim -p sample_input/example01.o > my_output
$ diff -Naur my_output sample_output/example01
```

If there are any differences (including whitespaces) the diff program will print the different lines. If there are no differences, nothing will be printed. Furthermore, we have provided a simple checking mechanism in the Makefile. Executing the following command will automate the checking procedure.

```
$ make test
```

There are 10 code segments to be graded and you will be granted 10% of total score for each correct binary code and **being "Correct" means that every digit and location is the same** to the given output of the example. If a digit is not the same, you will receive **0 score** for the example.

7. Submission (Important!!)

7.1 Make sure your code works well on your allocated Linux server.

In fact, it is highly recommended to work on your allocated server throughout this class.
Your project will be graded on the same environment as your allocated Linux server.

7.2 Summarize the contribution of each team member.

If you are working with your teammate, you need to summarize your contributions to each project. Add a 'contribution.txt' file in your repository (don't forget to commit it). If you use good commit messages, this can be done in a simple step. **"git shortlog"** summarizes commit titles by each user and will come in handy (especially if your commit titles have useful information).

```
$ git shortlog > contribution.txt
$ git add contribution.txt
$ git commit
```

If you want to add commit messages, please fill in the part after the option '–m' when committing.

```
$ git commit
```

A text editor will pop up with some information about the commit. Fill out your commit message at the top. The first line is the subject line of the commit. The second line should be blank, the third line and onwards will be the body of your commit message.

7.3 Add the 'submit' tag to your final commit and push your work to the gitlab server.

The following commands are the flow you should take to submit your work.

```
$ git tag submit
$ git push
$ git push --tags
```

If there is no "submit" tag, your work will not be graded so please remember to submit your work with the tag.

If you do not `push` your work, we will not have the visibility to your work. Please make sure you push your work before the deadline.

7.4 Updating Your Submit Tag

If you decide after tagging your commit and pushing, that you want to update your submission, you will need to remove the existing tag and retag & repush your submit tag.

```
$ git tag -d submit       # Deletes the existing tag
$ git push origin :submit # Removes the 'submit' tag on the server
```

Now you may re-tag your work and submit using the instruction in Section 7.3.

8. Late Policy

You will lose **50%** of your score on the **first day** (Nov 17th 0:00~23:59). We will **not accept** works that are submitted after then.

Be aware of plagiarism! Although it is encouraged to discuss with others and refer to extra materials, copying other students or opened code is strictly banned.

The TAs will compare your source code with open source codes and other team's code. If you are caught, you will receive a penalty for plagiarism.

If you have any requests or questions regarding administrative issues (such as late submission due to an unfortunate accident, GitLab is not working) please send an e-mail to the TAs(cs311_ta@calab.kaist.ac.kr).

9. Updates/Announcements

If there are any updates to the project, including additional tools/inputs/outputs, or changes, we will post a notice on the Notice board of KLMS, and will send you an e-mail using the KLMS system. **Frequently check your KLMS linked e-mail account or the KLMS notice board for updates.**

9.1 Merging any updates into your repository

During the course of the project, the TA may need to update some sample_input/sample_output files. If so the TA will notify you via KLMS notice board and mailing system from KLMS. If the TA instructs you to pull any new changes you can do the following.

Move to your local git directory. Commit any uncommitted changes.

```
$ git remote add upstream ssh://git@cs311.kaist.ac.kr:10022/TAs/Project3.git
$ git pull upstream master  # This may ask for you to write a merge commit
                            # Just save and exit
```

Now you should have caught up with the TA's change. Check with `` `git log` `` and check if you see the TA's commit at the very top. If the TA sends you another notification asking for merging any new changes, you can skip the first command and only execute the second `` `git pull upstream master` `` command.

Keep going with your work!