Project 3. MIPS Pipelined Simulator

Due 23:59, May 9th

TA: Sangyeop Lee, Minsu Kim

1. Introduction

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 read this document and README.md file provided in the repository carefully before you start.**

For any questions related to the project, please ask on the Piazza board or via email (cs311_ta@casys.kaist.ac.kr).

Additionally, we will run 2 office hours for this project:

- **1 week before the deadline: Friday, May 3rd, 2024, 8pm ~ 10pm**
- **2 days before the deadline: Tuesday, May 7th, 2024, 7pm ~ 9pm**

@ E3-1 4th Floor, Room 4429

2. Required Pipeline Implementation

In this project, you will need to implement the 5 stage MIPS pipeline that you have learned in class. The baseline MIPS simulator (sample solution of project 2) will be given as a backbone.

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.

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.

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.

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.

Control Hazard
Similar to project 2, we are going to ignore the JAL delay slot for this project. Therefore, when calling the JAL instruction, save the PC+4 value into the R31 (as with project2).

For conditional branches (BEQ, BNE), your simulator must support the **1-bit dynamic branch predictor** which is universally used regardless of PC. The 1-bit dynamic branch predictor is initialized to predict the next branch as **not taken**, but **it changes the branch prediction if prediction fails**. The 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 flushing will take place at the beginning of the MEM stage.** A correct branch prediction to **not taken** will not incur any stalls, but a correct prediction to **taken** will incur a single cycle stall; a miss prediction will cause 3 cycle stalls (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.

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.

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.md file in your submission. You must explain what each field means.

For more information about simulator requirements, please see project 2 document.

Running Simulation
$ ./cs311sim [-m addr1:addr2] [-d] [-n num_instr] [-p] <input file>

Simulation Options

-m : Print the memory content from `addr1` to `addr2`
-d : Print the register file content for each instruction execution. Print the memory content together if –m option is specified.
-n : number of instructions simulated
-p : Print the PCs of the instructions in each pipeline stage at every cycle. Prints a blank if the stage of the pipeline is stalled or empty.

3. GitLab Repository

Basically, you need to repeat the same procedure of forking and cloning the repository explained in the document for Project 1 and 2. The only difference is that you should start from the TA's repository of **Project 3** (https://cs311.kaist.ac.kr/root/project3).
Please check prior notifications available in Piazza board, for more information about getting started with GitLab.

4. Grading Policy

Submissions will be graded based on the 10 test cases. **6 of them will be randomly chosen from the 'sample_input' directory** and **4 of the hidden cases will not be opened**. Hidden examples have similar complexity with respect to given examples.

To get a full score, your simulator should print the exact same output as the reference solution. You can evaluate your code within given examples by comparing your output with files in 'sample_output' directory. **You may check the correctness of your code by executing** `make test` **at your working directory of this project**.

If there are any differences (including whitespaces) it will print the differences as below (here, there are two different lines corresponding to register R4 and R31). If there is no difference for an example, it will print "Test seems correct".

```
Testing example01
        Test seems correct

Testing example02
        Test seems correct

Testing example03
        Test seems correct

Testing example04
        Test seems correct

Testing example05
        Test seems correct

Testing fact
--- sample_output/fact  2019-09-26 21:29:59.290040281 +0900
+++ -   2019-09-26 21:30:04.128396077 +0900
@@ -8,7 +8,7 @@
 R1: 0x00000000
 R2: 0x00000001
 R3: 0x00000000
-R4: 0xfff1ffec
+R4: 0xffffffec
 R5: 0x00000000
 R6: 0x00000000
 R7: 0x00000000
@@ -35,5 +35,5 @@
 R28: 0x00000000
 R29: 0x00000000
 R30: 0x00000000
-R31: 0x0040201c
+R31: 0x0040001c

        Results not identical, check the diff output
```

Please make sure your outputs for given examples are **identical to the files in the 'sample_output' directory, without any redundant prints. Every single character of the output must be identical** to the given sample output. Otherwise, you will receive **0 score** for the example.

Note that **simply hard-coding outputs for given examples would lead you to 0 score** for this project.

5. Submission (Important!!)

Make sure your code works well on our class Linux server.
It is highly recommended to work on the class server since your project will be graded on the same environment as those servers.

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.

If there is no `'submit'` tag, your work will not be graded. Please do not forget to submit your work with the tag.

Please make sure you push your work before the deadline. If you do not "push" your work, we won't be able to see your work so that your work will be marked as unsubmitted.

For more information about submission procedure, please refer to the specification of 'Submission' in project 1 document.

6. Late Policy & Plagiarism Penalty (Important!!)

Submissions that are **one day late** (Friday, May 10th, 0:00~23:59) will lose **30%** of your score. We will **not accept** any 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: Not only for main routine functions, but also helper functions.**

TAs will compare your source code with open-source codes and other students' code. If you are caught, you will receive a serious penalty for plagiarism as announced in the first lecture of this course.

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 TAs.

7. Tips

Modifying your repository via http may cause some problems.

You may use C++ instead of C if you want. Just make sure your `` `make test` `` command works properly.

Please, do not modify contents (workflows and functionalities) of files other than `util.h`, `run.h`, `run.c`, and `Makefile`.

Please start as early as possible.

Please read this document, `` `README.md` ``, and given skeleton code carefully before you start.