SCE212 Project 3: Simulating Pipelined Execution

Due 11:59pm, January 17th, 2018

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 post your question on Ajou Black Board.

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

We will fork the SCE212/Project3 repo to your team namespace. Then you will clone your team's repo into your local machines to work on the project.

(1) Go to the following page: http://beehive.ajou.ac.kr:10080 and register your ID.
(2) Your team needs to create a group "SCE212_team0X".
(3) Find the SCE212/project3: http://beehive.ajou.ac.kr:10080/SCE212/Project3. I have created a repository.
(4) You can just fork the project in your group page which has been created at (2).
(5) Cloning the team repository to your local machine

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

5. Simulator Options and Output

5.1 Options

```
$ ./sce212sim [-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 simulator

We will be providing a reference simulator (without the code, of course) so that you may compare the execution of your simulator to the reference. You will be able to dump debug messages, pipeline outputs of both the reference and your simulators to check the execution every cycle.

The 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 answer. We'll check it out and push a fixed reference, and fixed sample outputs.

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.

```
$ ./sce212sim -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.