CSE26101 Project 3: Simulating Pipelined Execution

Due 11:59 PM, Nov 21st

1. Overview

This project aims to help you understand pipelined executions. You will build a 5-stage pipelined simulator of a subset of the MIPS instruction set. The simulator loads an MIPS binary into a simulated memory and executes the instructions. Instruction execution will change the states of registers and memory. Your simulator should support pipelined execution and handle any data or control hazards.

2. Simulation Details

For a given input MIPS binary (the output binary file from the assembler built-in Project 1), the simulator must be able to mimic the behaviors of the 5-stage MIPS pipelined execution.

2.1 Stages in the Pipeline

1. **IF**: fetch a new instruction from memory.
2. **ID**: decode the fetched instruction and read the operands from 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.

2.2 Register File

You need to add pipeline register states between stages. The following 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 the comments.

2.3 Register File

The register file is used in both **ID** and **WB** stages. You should schedule the read and write operations such that there are no structural hazards in the register file.

2.4 Memory Model

We assume an ideal dual-ported memory, which allows read@**IF** and read/write@**MEM** stage simultaneously within the same cycle. We further assume the memory location of the instructions stored does not change, so that stores@**MEM** will never update the same address fetched @**IF** of the same cycle.

2.5 Forwarding

Your simulator should support data forwarding for

1) **MEM**/**WB**-to-**EX**
2) **EX**/**MEM**-to-**EX**
3) **MEM**/**WB**-to-**MEM**

This way, data hazards occur only for the data dependency from a load to the succeeding instruction which uses the value in the **EX** stage.

2.6 Hazard Control

Unconditional jumps (`J, JAL, JR`): You need to add a one-cycle stall to the pipeline. For JAL instructions, assume there is a single delay slot after JAL and save PC+8 (instead of PC+4) into R31. We have modified the binary and assembly files used for testing to have an NOP instruction (add $0, $0, $0) after each JAL instruction. For the sake of simplicity, you do not have to execute NOP. You can simply flush the IF stage when the jump decision is made @ID stage, to skip executing the delay slot.

Conditional branches (`BEQ, BNE`): You should implement a static branch predictor that always predicts the branch is not taken. The actual branch decision depends on the ALU output and will be ready at the end of the EX stage. You should flush the pipeline at the beginning of the MEM stage in case of a mis-prediction. In this case, mis-predictions cause a 3-cycle stall (EX, IF, and ID of the subsequent instructions).
If the branch decision has dependencies on prior executions, stall the execution at the appropriate stage.

2.7 Stopping the Pipeline

The simulator must stop after a given number of instructions. You should stop the execution after the WB stage of the last instruction.

2.8 Supported Instructions

Not all instructions in PS2 are supported in PS3. Specially, the following instructions should be implemented:

ADDIU	ADDU	AND	ANDI	BEQ	BNE	J	JR	LUI
LW	SLTIU	OR	ORI	SLTIU	SLTU	SLL	SRL	SW
SUBU	JAL							

The following were required in PS2, but no longer required in PS3:

ADD	ADDI	SLT	SLTI	SUB

3. Testing and Submission via PA Submit System (PASS)

We will use ***PA Submit System***, developed by the Systems Software Lab of Ajou University, to manage the programming assignments. Please refer to the BlackBoard announcement for the PASS user manual.

3.1 Downloading the Skeleton Code

The skeleton code is available on BlackBoard under "Assignment" → "PA3: Building a Simple MIPS Simulator". Please download and read the skeleton code carefully before you begin working on PA3, you can start with the README file. Some macros and helper functions are given, please do not modify them.

If you do not want to use the skeleton code, you are allowed to write code from scratch. However, you are supposed to follow the input and output file format exactly, as the grading script works on the provided sample input and sample output files provided to you.

3.2 Submission

You should zip the two files: *run.py* and *parse.py* as *ps3.zip*, and submit the zipped file on PASS , under "PA3: Simulating Pipelined Execution". **Please do not modify the file name or include unnecessary files in your submission.**

Due to the server issues and the complexity of execution outputs of PA3, we have modified the submission limit. You can submit up to **20** times for PA3; We encourage you to use PASS only to verify your implementation, and not to use it for debugging purposes.
Instead, we will provide more complicated local test cases and a reference simulator for you to test your own testcases, please see section 3.3 for details.
You can choose which submission will be used for grading.

You should remove or comment out unnecessary codes before you submit to PASS, especially if it prints or generates log files. The automated grading program may fail to evaluate your assignment accurately.

3.3 Testing Locally

You can test your implementation locally with the following command:

```
$ python main.py [-m addr1:addr2] [-d] [-n num_instr] [-p] inputBinary
```

-m: Dump the memory 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; The default is 100 instructions
-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/empty.

Your simulator should print the output with standard output. The output file should show memory contents of specific memory regions, PCs of each pipeline stage at every cycle and register values.

The functions for printing the memory and register values are provided in the *util.py* file.

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, and pipeline outputs of both the reference and your simulators to check the execution of every cycle.

You can generate outputs for your own testcases by:

```
$ ./reference_simulator [-m addr1:addr2] [-d] [-p] [-n num_instr] inputBinary
```

The reference *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.

4. Grading Criteria

Your assignment will be graded with 10 testcases. Each testcase is worth 1 point, no partial points will be given.

For late submissions, your slip days will be automatically subtracted; post-slip-day submissions will receive a 20% penalty from your score for each day late.

5. Updates/Announcements

We will post a notice on BlackBoard and Pizza if there are any updates to the assignment, please check BlackBoard and Piazza regularly.

Note that you are allowed to import and use external packages for this assignment. If you think it is necessary to use an external package, you must first get permission from the TAs by posting in Piazza. Your post must explain clearly why the package you intend to use is necessary. **Using an external package without permission will be regarded as plagiarism.**

6. Misc

Be careful about plagiarism! Last semester, we found a couple of plagiarism cases through an automated tool. If you are caught in “deep collaboration” with other students, we will split the score equally with your collaborators.
Please do not upload your code when posting in Piazza as it is considered as cheating. If you think your question contains hints to other students, please make sure you post it as a “private post”.

If you have any requests or questions (technical difficulties, late submission due to inevitable circumstances, etc.), please ask the TAs on Piazza.

We generally encourage the use of Piazza for discussions. However, for urgent issues, you can send an email to the TAs (minseok1335@unist.ac.kr(Head TA) / dyryu@unist.ac.kr / xinyuema@unist.ac.kr / garvel@unist.ac.kr).

7. FAQs

Q1. How to run the simulator in windows?

A: Please use WSL or VirtualBox to create a linux environment and run the simulator on top of it.

Q2. Can I add additional control signals or registers?

A: You are free to add additional properties in the files to submit. Just make sure that you implement them in the files that you should submit (**run.py**, **parse.py**), not in the files that you should not submit.