Computer Organization Project #2

TA in charge: Wonyoung Lee
E-mail: wy_lee@kaist.ac.kr
Office Hour: 14:30 ~ 16:00, Tue. / Thu. (N1, 403)

I. Goal of this assignment

✓ Comprehensive understanding on pipeline architecture, hazard and branch prediction.

II. Work as a Team

✓ Project #2 is a team project. Up to three of students can be a team.

✓ You can do it alone or with a student; but there is no additional points to compensate for the lack of members.

III. Submission and Grading

✓ You should submit:

1. The intermediate and result files **(10 points)**:
   After simulation, you can get two intermediate files (*default.trc* and *bpred_not_taken.trc*)and two result files(*default.result* and *bpred_not_taken.result*)

2. Reports **(90 points)**
   The detail about the report will be explained in Section VII.B in the document. You can write your report in either English or Korean. The report file name should be [Project2_StudentID1_Name1_StudentID2_Name2_StudentID3_name3.pdf].

   (e.g. *Project2_20160000_WonyoungLee_20150000_JohnSmith_20140000_Anonymous.pdf*)

✓ Compress files into a .zip file and upload the compressed file on KLMS. The compressed file name should be [Proejct2_ StudentID1_Name1_~~~.zip].
(e.g. *Project2_20160000_WonyoungLee_20150000_JohnSmith_20140000_Anonymous.zip*)

IV. Due Date

✓ Nov. 9th Fri., 23:59:00

✓ Late submission due: Nov. 10th Sat., 23:59:00 (50% point deduction)

✓ After late submission due date, you cannot get any point.

✓ Do not try submit it at the very last moment. Please submit it with enough time to spare.

V. Installation

- ✓ You will use a different virtual machine image from Project #1.
- ✓ You can download it through KLMS Project #2 page that you download this document.
- ✓ The virtual machine image can be used in both VirtualBox and VMware. Download and setup the experimental environment with the image file as you did in Project #1.
- ✓ If you don't want to use virtual machine and want to do this Project #2 in your own Linux environment or in the virtual environment that you used in Project #1, download and install SimpleScalar-3.0 in following web site: http://www.simplescalar.com/tools.html

VI. Background

A. Differences between MIPS and Experimental Environment

i. Alpha ISA

The provided experimental environment is not exactly the same as MIPS architecture that we have learned in our class. It is called Alpha ISA. Alpha assembly is similar to MIPS', but it is little different, such as "add" in MIPS is "addq" in Alpha. You can find the definition is Alpha assembly instructions in the following web page: http://www.cs.hut.fi/~cessu/compilers/alpha-intro.html

ii. Pipeline Architecture

As we learned in the class, MIPS has 5-stage pipeline:

1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back to register file

However, in this experimental environment, the result of an command execution (will be explained soon) will show also 5-stage pipeline, and it is little different from MIPS' pipeline.

1. IF: Instruction fetch
2. DA: Instruction decode and register file read, Awaiting issue
3. EX: Execution or address calculation, **Data memory access**
4. WB: Write back to register files, Awaiting retirement
5. CT: Instruction retiring

You do not have to understand the concept of *issue* and *retirement* at this moment; it is advanced features for modern out-of-order processors. Therefore, this pipeline can be thought of as 4-stage pipeline that EX and MEM stages are merged into a single EX stage.

B. An Example of Execution Result

i. Execution Commands

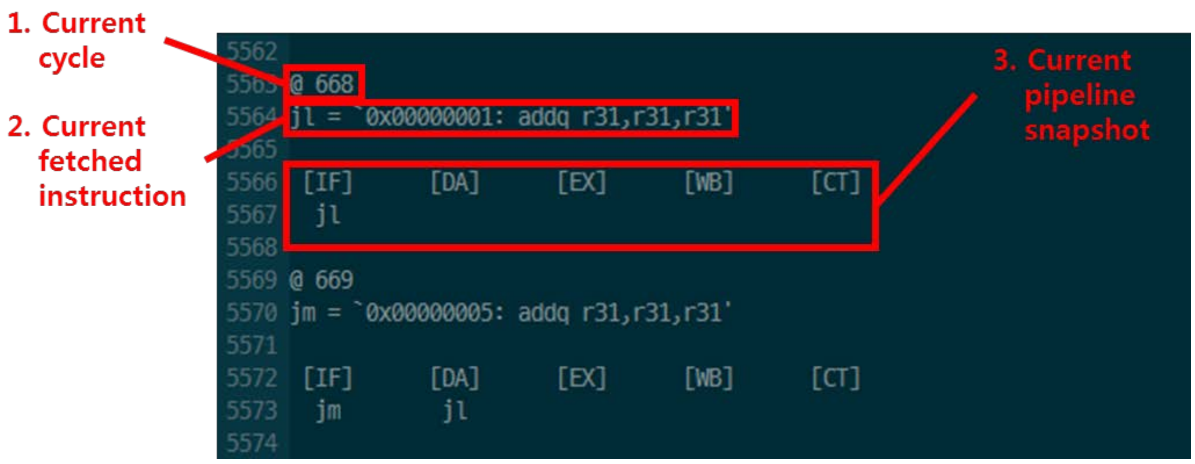
The students will execute the provided commands to generate the pipeline snapshots file. Below lines are an example of the commands

```
./sim-outorder -ptrace FOO.trc ./tests-alpha/bin/test-math
./pipeview.pl FOO.trc >> FOO.result
```

In the first line, it runs SimpleScalar and generates the intermediate "FOO.trc" file that includes pipeline snapshots. However, it is hard to read the intermediate file immediately, so the second line changes it to a human-readable file named "FOO.result".

ii. Execution Results

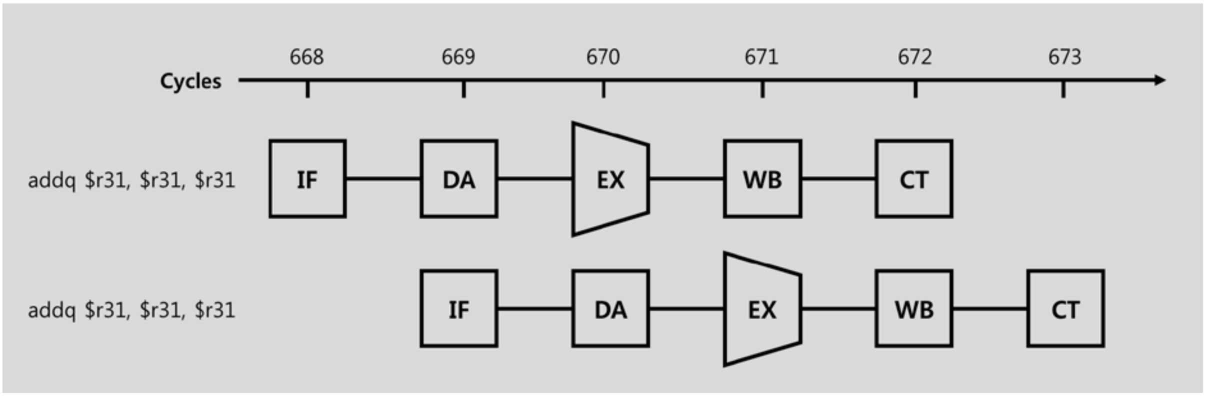
The execution of the above command generates "FOO.result" file, and the below captured image shows a part of the file contents:



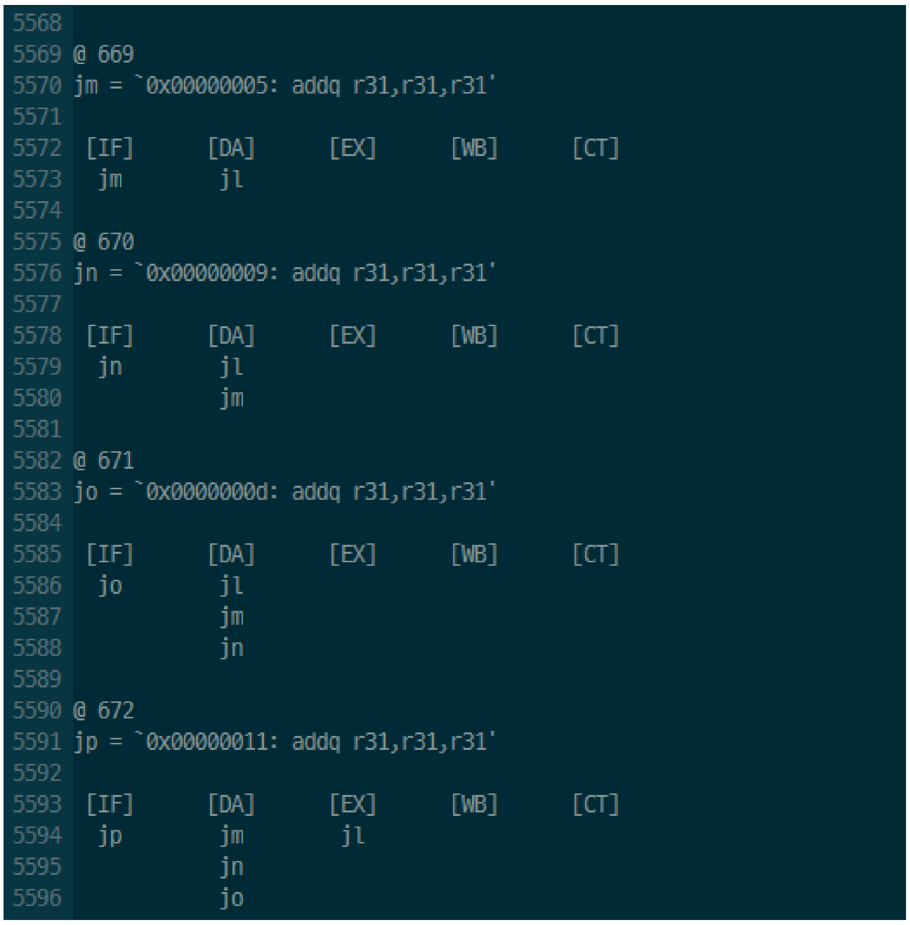
```

@ 668
jl = `0x00000001: addq r31,r31,r31'

 [IF]      [DA]      [EX]      [WB]      [CT]
  jl

@ 669
jm = `0x00000005: addq r31,r31,r31'

 [IF]      [DA]      [EX]      [WB]      [CT]
  jm        jl

```

1. **Current cycle**: the cycle number of the specific pipeline snapshot.
2. **Current fetched instruction**: the instruction fetched at the cycle. Each instruction has its own ID (e.g. `jl`) to distinguish each other in pipeline status.
3. **Current pipeline snapshot**: it shows the pipeline snapshot of the cycle. In this example, an "addq" instruction is fetched at 668 cycle, and another "addq" instruction is fetched at the next cycle (at 669), so there are two instructions simultaneously.

The above captured image can be transformed as below familiar figure (without consideration of hazards):



Let us check more cycles from the above example. The below captured image shows the pipeline snapshot from 669 cycle to 673 cycle.

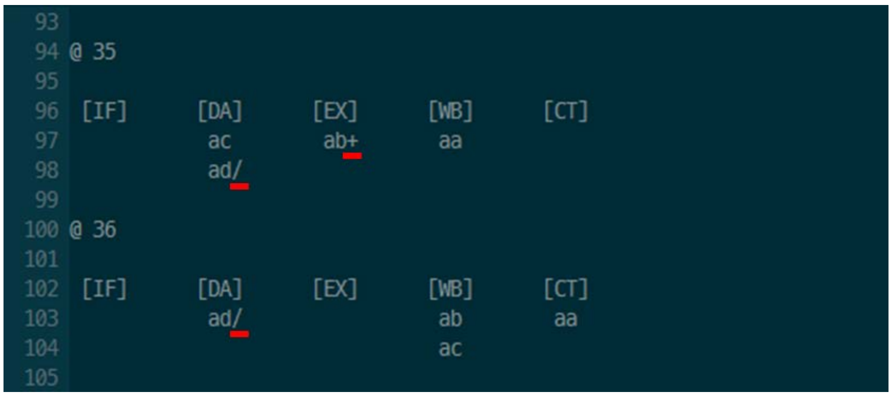


```

@ 669
jm = `0x00000005: addq r31,r31,r31'

 [IF]      [DA]      [EX]      [WB]      [CT]
  jm        jl

@ 670
jn = `0x00000009: addq r31,r31,r31'

 [IF]      [DA]      [EX]      [WB]      [CT]
  jn        jl
            jm

@ 671
jo = `0x0000000d: addq r31,r31,r31'

 [IF]      [DA]      [EX]      [WB]      [CT]
  jo        jl
            jm
            jn

@ 672
jp = `0x00000011: addq r31,r31,r31'

 [IF]      [DA]      [EX]      [WB]      [CT]
  jp        jm        jl
            jn
            jo
```

In this _sequence of pipeline snapshots_, the instruction `jl` is stalled due to some unknown reason at 670 cycle. Consequently, other following instructions are also stalled because there are dependence on register `r31`. This situation is called **data hazard** as we learned in the class.

iii. Event Indicator

In the pipeline snapshot of result file, there is a mark right next the instruction ID sometimes:



```

@ 35

 [IF]     [DA]     [EX]     [WB]     [CT]
           ac       ab+      aa
           ad/

@ 36

 [IF]     [DA]     [EX]     [WB]     [CT]
           ad/               ab       aa
                             ac

```

These marks are *event indicator*, and it shows a relevant event with the instruction at that moment. There are five kind of event indicator:

1. *: Cache miss
2. !: TLB miss
3. **/: Expected to be branch mis-prediction**
4. **\: Branch mis-prediction detected**
5. +: Address generation execution

“/” mark appears on [DA] stages, and it represents that the instruction will be mis-predicted on its branch operation. And soon, in the [WB] stage, “/” mark is changed to “\” mark. You don’t have to understand why this simulator uses two separate marks at this moment; the thing you should understand is the “/” and “\” marks represent branch mis-prediction. The branch instruction without these marks is correctly predicted one.

VII. What You Should Do For Project #2

A. Run command

In the virtual machine image, there are prepared command lines in "running.sh" script file. You can run the simulation simply by executing the script file.

```
cs311@cs311:~$ cd simplesim-3.0/
cs311@cs311:~/simplesim-3.0$ ./running.sh
```

OR, you can run below commands directly. These will generate the exactly same result.

```
# Default branch predictor
./sim-outorder -ptrace default.trc 10000:11000 \
  -fetch:ifqsize 1 -decode:width 1 -issue:width 1 -commit:width 1 \
  ./tests-alpha/bin/test-math &&
./pipeview.pl default.trc >> default.result

# With always-not-taken branch predictor
./sim-outorder -ptrace bpred_not_taken.trc 10000:11000 \
  -fetch:ifqsize 1 -decode:width 1 -issue:width 1 -commit:width 1 \
  -bpred nottaken \
  ./tests-alpha/bin/test-math &&
./pipeview.pl bpred_not_taken.trc >> bpred_not_taken.result
```

Above two sets of code lines generate two result files: "*default.result*" and "*bpred_not_taken.result*". "*default.result*" is generated by using default branch predictor, and "*bpred_not_taken.result*" is generated by using a branch predictor that predicts that branch instructions will always not be taken.

B. Analyze the result file

Based on the generated result file, the students should write report as directed below.

i. Pipeline basic (with only "*default.result*" file, 20 points)

1. Capture a sequence of pipeline snapshots in cases of Load/Store, R-type and Control instructions (Total 3 captures of the sequence of snapshots). Include the captured image in your report.
2. Calculate how many cycles are required to perform each instruction in case of the above captured sequence of pipeline snapshots.

ii. Hazard (with only "*default.result*" file, 30 points)

1. Capture a sequence of pipeline snapshots in case of a hazard situation. Include the captured image in your report. Specify the type of the hazard (i.e. Data, Structure or Control hazard).
2. Calculate how many cycles are wasted due to the hazard in the captured sequence. Discuss how to prevent such cycle wasting in case of the captured sequence. Include the discussion in your report.
3. Discuss following questions and include the result in your report: In the case of this simulator, does it seem like it has any kind of hazard-prevention method (e.g. forwarding, bypassing, etc.)? If it does, how can you figure it out, and what kind of methods are used. If it does not, why you cannot figure it out?

iii. Branch Prediction (with both "*default.result*" and "*bpred_not_taken.result*" files, 40 points)

1. With the *default.result* file, Capture the sequence of pipeline snapshots in cases of branch mis-prediction and branch correct-prediction (Total 2 captures of the sequence of snapshots). Include the captured image in your report.
2. With the *default.result* file, this result file is generated by the default dynamic branch predictor, and we learned 1-bit and 2-bit branch prediction in our class. Can you deduce how many bits are used for this predictor (i.e. 1-bit, 2-bit, or even 3-bit predictor) only with this result file contents? If you can, provide the capture of pipeline snapshots with appropriate explanation to support your claim. If you cannot, explain why it is impossible.
3. With both "*default.result*" and "*bpred_not_taken.result*" files, Count every **beq** instructions in both file. How many **beq** instructions are mis-predicted and correct-predicted? Calculate mis-prediction rate based on the count results, and choose a better branch prediction policy (default or always-not-taken). Explain why the policy is better.

VIII. Cheating

- ✓ If there are any cheatings in your submission, you will get 0 point.
- ✓ *Followings will be regarded as cheating*:
  - A. Copying other students' simulation results or reports.
  - B. Modifying other students' results and using them as if they were your own.
  - C. Using other sources without any references excluding your own simulation results.
  - D. All other sorts of inappropriate behaviors.