Computer Architecture 2020 Lab 2

Processor Lab

Table of Contents

Overview 2

Provided Infrastructure 3

Tasks 3

1. Setup Environment 4

2. Implement all instructions in Lab 1 5

3. Handle Read-After-Write Hazards 8

Write-up 9

Methodology 9

Results 10

Question Answering 11

Submission 11

Grading policy 11

Functionality 12

Timing 12

Update: 20200405 20200326

Update description:

Go to piazza for what's new

Overview

Young emperor Qin Shi Huang waked up from a nightmare. In his dream, three Suns went into a straight line, leading his people to the devastation. The only solution was to calculate the orbits of those stars. At the same time, von Neumann and Wang Miao came to visit, with their powerful design called 5-stage CPU. According to their design, the emperor trained five troops, in charge of instruction fetching, decoding, executing, memory, and regfile writing-back. There was only one big problem in human-based processors: it was hard to synchronize the signal among the distant troops. In other word, these events can not be issued at the same time physically. Therefore, a global scheduler was established and operated by the emperor, to guide all events in the order of their timestamps. And each instruction was divided into five ordered events, correspondingly going through each stage. Although two events can not happen at the same physical time, their timestamps can be identical, resulting in logical paralleling.

The emperor was well trained and therefore he knew that testing is as important as developing, if not more important. Before solving the actual problem, the emperor wanted to first test this "machine" with an image processing MIPS assembly that he dreamed. The functionality (output) should be the same with Lab 1 and the timing statistics should be at least close enough in order to convince the emperor ...

-- adapted from *The Three-Body Problem: Remembrance of Earth's Past*.

In this lab, you are going to implement your own MIPS processor and emulate your final program from Lab 1. The processor should provide a similar but more accurate cycle information than the simple cycle counter used in Lab 1. Needless to say, this information is extremely valuable to the emperor and his people, so please be cautioned ...

Before starting this lab, you are supposed to be equipped with the following:

- The same environment as Lab 1 (the assembly lab)
- Basic C++ programming language
- Basic MIPS assembly

After this lab, you will obtain the following:

- Understanding 5-stage pipelining CPU model
- Better understanding about emulating, especially SPIM, a MIPS emulator

During this lab, you are suggested to refer to the following materials:

- MIPS32 Green Card
- Linux cheat sheet

http://cheatsheetworld.com/programming/unix-linux-cheat-sheet/

- Syscalls defined by SPIM

  http://students.cs.tamu.edu/tanzir/csce350/reference/syscalls.html

Provided Infrastructure

Inside this repository, we provide the following files in the tree form:

```
├── yao-archlab-s20
   ├── lab2
       ├── lab2-handout.pdf --- This document
       ├── run --- Our workspace folder
   ├── spim-timingmodel/CPU
       ├── lab2_def.h --- Defining cycles spent on each stage
       ├── timingmodel.h --- Basic definition of Timing Event/Component
       ├── timing_core.h --- Definition of CPU Top design
       ├── timing_{fetcher, decoder, executor, lsu, register, rob}.h/cp
p --- Basic implementation of specific processor component
       ├── alu.h --- Basic implementation of Arithmetic Logic Unit
       ├── next_pc_gen.h/cpp --- Basic implementation of PC generation 
Unit
       ├── scheduler.h/cpp --- The global scheduler for timing events
       ├── spim-utils.cpp --- Modified SPIM file. It launches the user 
program by lab2_run_program().
       └── run.cpp --- Modified SPIM file. It executes each instruction
 by run_spim().
```

Tasks

Figure 1 illustrates the 5 stages in the MIPS processor that you are going to implement.

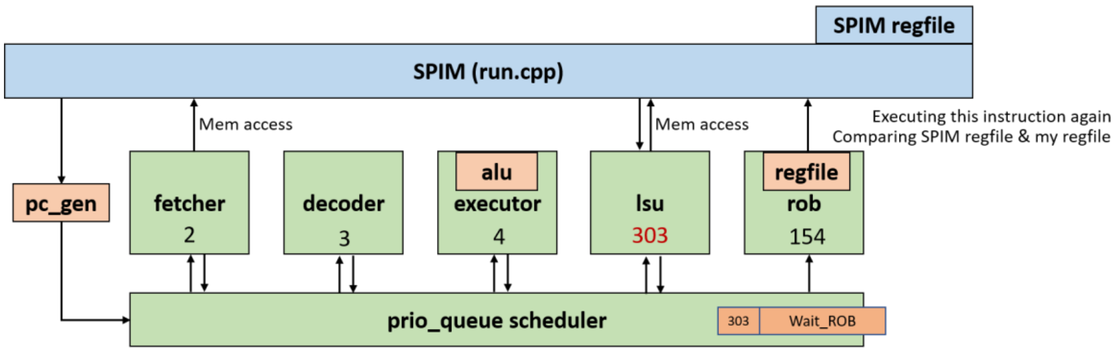

Figure 1: Workflow of SPIM-T (rev07 pp.5)

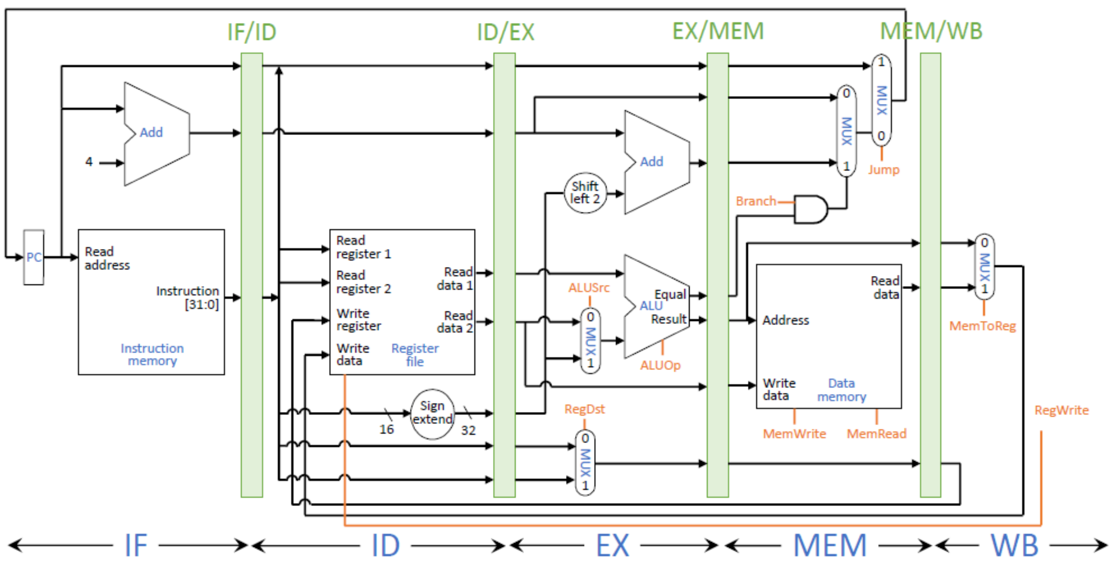

Figure 2: A possible 5-stage pipelining processor design (lec06 pp.14)

We have provided a processor framework which can only execute `addiu`, `jal`, `lw` and `syscall` instructions (refer to Figure 2). And `sll` is not correctly implemented. You should:

- Finish all related instruction types in your Lab 1 program (either the C version or the MIPS version is acceptable).
- Handle Read-After-Write data dependency and one-slot delayed branch correctly.
- Write a design document.

1. Setup Environment

First make sure you have the identical directory structure as in the section Provided Infrastructure.

```
$ export LAB2=$HOME/yao-archlab-s20/lab2
$ cd ~/yao-archlab-s20/spim-timingmodel/
$ git pull origin # catch up the newest code change
$ git checkout -b lab2 lab2-lecstyle-v1.0 # switch to latest lab 2 framework
$ cd spim
$ make clean
$ make && make install
$ cd ../../lab2
```

```
$ ln -sf $LAB2/../spim-timingmodel/spim/spim ./run/ # create a shortcut
for emulator
```

You should copy your final assembly program (*.S* file) from Lab 1 to lab2/run/ directory.

You can still validate your environment setup by `./spim file simple_add.S` in `run/` directory. It should return the sum of two input numbers. You can refer to the handout of Lab 1 for its usage.

We provide a basic implementation of this five-stage processor. If you want to test it, try `$ ./spim -lab2 file simple_add.S`. You will see an error message which probably looks like this:

```
initial PC:400000
[0x00400000]    0x8fa40000  lw $4, 0($29)                   ; 183: lw
$a0 0($sp)                # argc
        reg id:4 write:0x1
        addr: 0x400000 ROB correct
...
[0x0040000c]    0x00041080  sll $2, $4, 2                   ; 186: sll
$v0 $a0 2
        Error reg2: 0x0 should be 0x4
spim: ../CPU/timing_register.h:38: int
TimingRegister::CheckCorrectness(): Assertion `false' failed.
Aborted (core dumped)
```

This is because the basic implementation does not contain all instructions, such as `sll`. The register file of our own processor does not match the correct register file by SPIM, which leads to the emulating abortion. You may notice that there is not a `sll` instruction in our `simple_add.S`. This is because SPIM automatically inserts a few instructions at the beginning of our program, most of which are related to argc/argv setup. If you want to only emulate your program, use the `-skipstart` flag: `./spim -lab2 -skipstart file simple_add.S`, you will find your program now aborted for `addu/move` unimplementation. But eventually you should emulate your program without `-skipstart`.

2. Implement all instructions in Lab 1

You are supposed to implement your own processor in `~/yao-archlab-s20/spim-timingmodel/CPU`. We have specified the code structure with the following initial files for you to start with:

- `timingmodel` has the definitions of all base classes like `TimingComponent` and `TimingEvent`. `TimingComponent` is the base class for pipeline stages. `TimingEvent` represents the execution details of an instruction.

- `timing_fetcher`, `timing_decoder`, `timing_executor`, `timing_lsu`, and `timing_rob` represent the five stages (IF, ID, EXE, MEM, WB) in our processor.
- `next_pc_gen` should return the address of the next instruction to be fetched.
- `timing_register` and `alu` are the register file and ALU of our processor.
- `scheduler` is the global scheduler of timing events.
- `timing_core` provides a top class which includes all the hardware components.

You can **only** modify any of the above files (`*.cpp` or `*.h`) in your implementation. The other files should remain unchanged, and during the evaluation we will overwrite them. So make sure that your implementation can work correctly.

Don't be scared by so many files. Modular design is always a good habit for understandability. You will find that most of the files are short and simple after you finish the lab.

The entry point of your program is at `spim-utils.cpp::lab2_run_program`. One way to get started is to follow the execution of an already-implemented instruction over the 5 stages from IF to WB. For example, `addiu` is an example illustrating how an I-type instruction gets executed. You can follow it to implement more I-type instructions like `ori` and `andi`. Use `gdb` for step debugging if you are in any doubt.

An alternative way is to implement instructions in the order of program execution. For example when you see errors due to unimplementated `sll`, you try to realize `sll`. This way is more challenging. You may not be able to write your first line of code until you have gone through the whole Lab 2 framework and understanded how it works.

After you implement your CPU design, you should go to `spim-timingmodel/spim/` and build your project again (`$ make`). Finally your processor should emulate the whole program of your Lab 1 submission (either the C version or the MIPS version is okay). You are supposed to see an output from your own processor without unimplementation error. However the output might not be correct, and some of your instructions might write back with the wrong values. This is because RAW hazards are not handled correctly yet, which is the next task of this lab.

Here are some suggestions:

1. All function-related code is written in `CPU/run.cpp` in SPIM but they do not record timing information. Feel free to borrow the codes (and macros) from SPIM and split the code into five stages. You can check and compare `addiu` implementation in SPIM and our own implementation.
2. Classify your instruction properly in the decoding stage. Generally there are three instruction categories in MIPS: R-type, I-type and J-type. But they can be divided into more subtypes. A proper classification can greatly simplify your

`switch...case...` logic in the rest stages. Feel free to change (or even refactor) `timing_decoder.h` to do your own classification.

3. Sometimes you can exploit `$zero` for reusing ALU to write prettier codes. You can check the implementation of `jal`. This is one of the many wonderful merits of MIPS ISA design.

4. One-slot delayed branch is required in this lab (why?). That means, every instruction immediately following branch and jump instructions will always get executed even when the branch is not taken. For simplicity, you can insert `nop` after each jump instruction in your Lab 1 code before implementing delay slot. But eventually your processor should be able to execute any instruction in the delay slot.

5. Implement instructions one by one. To improve developing efficiency, we conduct register file checking after each instruction commits to ensure correctness (why it works?). How can we achieve this? We force SPIM to execute the same instruction once again and stop at the first different register write.

6. We have provided several simple assembly programs as milestones for you. They are named after `check_x.S` in `run/` directories. After you implement some of instructions, you can try emulating them with `-skipstart` flag, to see if they work. Particularly, `check_1.S` can be emulated successfully without any modification: `$ ./spim -lab2 -skipstart file check_1.S`. You should read the comments in the assembly for their functions.

7. Only comparison and `syscall` uses original SPIM regfile. And those parts have been provided. Other than those, **you must not use any value directly from spim like the regfile `R[]` and `PC`.**

8. SPIM-T can also generate ground-truth traces via `$ ../spim-timingmodel/spim/spim -lab2-gen -delayed_branches file lab1.S`, including correct PC addresses and register writings. You can manually compare the outputs with your own processor.

9. You shall not change files that begin with `DO NOT MODIFY` header! They are reserved for TA and will be overwritten automatically. It is okay to modify it for convenient debugging (e.g., lowering memory access latency) but do not forget to test your final processor with these original unmodified files.

10. We only finish in-core processing. The memory is shared with SPIM. Because SPIM executes the same instruction one more time at the end of the pipeline, you may think `sw` implementation is useless. That is not true. You must implement store instructions because otherwise, the store to memory only happens after the WB stage by SPIM, and a load immediately following the store will get the stale value at the MEM stage.

11. There are some disabled `assert_msg_ifnot` callings in the decoder and LSU. Enabling them would help with your implementation in a stricter way.

12. Our SPIM-T framework is extended to support more functionalities than SPIM-T in lab 1. You can use them by adding flags. Here is the full list:

- `file <assembly>` will load assembly from a file.
- `-display` will print every instruction executed by SPIM out to the screen.
- `-lab1` will do cycle estimation of lab 1. The code for cycle estimation is in `statistics.cpp`.
- `-redirect <inputfile> <outputfile>` will read inputs from `<inputfile>` and output to `<outputfile>`.
- `-skipstart` changes the entry point from `__start` to `main()`. In other word, no more instructions automatically added at the beginning of your program.
- `-lab2` will replace the original SPIM core with your own processor.
- `-lab2-gen` will print correct execution trace generated by original SPIM core.

3. Handle Read-After-Write Hazards

In this 5-stage pipelined processor, register operands are read at the ID stage but are written at the WB stage. This raises an issue when two consecutive instructions access the same register with a read-after-write dependency. In Lab 2 we use pipeline stalls to deal with such read-after-write hazards. That means, the latter instruction is not allowed to read the register until the first instruction gets committed (the result gets written back to the register).

Here are two ways to implement stalls:

- Use a Boolean array as "spinlocks" to the register file. Each register will be locked if an instruction indicates to write to it at the ID stage, and unlocked when the write actually happens at the WB stage. Any following instruction that tries to read a locked register in the ID stage should stall. You can use the `reg_used` array in `timing_register.h` for this purpose. Literally stalling an instruction seems abstract. Instead you can try re-decoding the same stalled instruction (this is also how real hardware designs do) by keeping to feed it to the ID stage each cycle until the lock is released.
- Or, you can treat the global scheduler as a priority queue which picks the **earliest unstalled events** and passes them to the correct stages. You can implement such hazard checks in `timingmodel.h::Comp_TimingEvent`, which is called by the global scheduler in `scheduler.cpp`. Hazard checks should be carefully considered. If you want, you can also rewrite the whole scheduler

structure, for example, using STL `priority_queue`. Remember that, our 5-stage pipeline processes instructions in their program order. Your scheduler needs to keep the instructions in order when doing scheduling. If you really want to try implementing an out-of-order processor, talk with the TA first.

Remember that the latencies of the five stages are not uniform: the MEM stage needs 150 cycles when handling memory requests. Your code should cover all possible stall situations.

Once we have implemented stalls, the global scheduler would select the earliest event with the lowest `current_cycle` and forward it to the receiver. Under one stage, the latter instruction always has the higher `current_cycle` than the previous ones. In this way can we totally emulate all in-order events.

Do we need to resolve structure hazards and control hazards? One hardware component can only accommodate one instruction at one cycle because the global scheduler would forward instruction only when the receiving stage is available. If you are interested, check the use of `available_cycle` in each timing component. As for control hazard, we should determine all branch results in ID stage. Then the branch delay slot you implemented in the previous task will solve the issue for us.

After you finish this task, you should generate the same output as it is in Lab 1. Try `./spim -lab2 -redirect input outputfile file lab1.S`. Also the cycle statistics should be at least close to your Lab 1 result. If they are not the same (notice this does not necessarily mean the implementation is incorrect), you should be able to explain the reasons of the differences in your write-up.

Write-up

You should write a design document for Lab 2. There is no specific format required, but you should demonstrate how your processor works, in a clear way. You should at least include the following parts:

Methodology

How did you finish the lab? Show your debugging process if there is anything interesting.

Describe the storage cost of your `TimingEvent` structure. You are encouraged to use only necessary variables on control path and data path in this structure because in real hardware designs the number of signals maintained is related to the final chip energy and area. You should describe your valid signals during each stage by completing the following table:

Timing Event Type	Valid signals	Size count

IF -> ID	pc_addr, inst*, ...	12B
ID -> EX	alu_src_1, alu_src_2, ALUOp, MemToReg, MemRead, MemWrite, RegWrite, inst_is_syscall, reg_wb_id, ...	??B
EX -> MEM	alu_result, MemToReg, MemRead, MemWrite, RegWrite, inst_is_syscall, reg_wb_id, …	??B
MEM -> WB	reg_wb_id, reg_wb_val, RegWrite, inst_is_syscall, ...	??B
All events	current_cycle, execute_cycles, start_cycle, ...	??B

Describe the solution you used to do instruction classification. How does this method help reduce the design complexity of the later stages?

Describe the solution of resolving RAW hazards. What are the pros and cons of your solution in terms of storage cost, computation complexity, and other relevant aspects?

Compare the final cycle counts with Lab 1. You should list at least three reasons why your statistics are more (or less) accurate than Lab 1. Remember that Lab 1 implementation is just a rough estimation without any real pipeline stages. The code of Lab 1 counters is in `statistics.cpp::update_latency_pipeline`. For each reason, provide how much difference in the number of cycles that you estimate.

Results

We can obtain a detailed timing report since we actually emulates the whole program. For example, we can record the stall cycles for each instruction and get average statistics over different instruction types. Make sure that when the pipeline is stalled, the stall cycles are only attributed to the first instruction that gets stalled, and not to the following ones. For example, if a `sw` instruction is stalled by a RAW hazard at the MEM stage for 1 cycle, we only count 1 cycle stall towards this `sw` instruction, but not for others currently in EX, ID, or IF stages. Note that memory instructions need 150 cycles for executing, not for stalling, but they may stall the preceding stages.

Fill in all the blanks below.

`Lab 1 (total instructions count, cycle count, CPI):`

`Lab 2 (total instructions count, cycle count, CPI):`

`Lab 2 average stall cycles of branch instructions:`

`Lab 2 average stall cycles of memory instructions:`

```
Lab 2 average stall cycles of register instructions:

Lab 2 average stall cycles of all instructions:
```

Question Answering

- From the statistic results above, could you quantitatively explain the overall CPI using the average stall cycles?
- For each instruction, we only compare the changed register value and the PC value with the ground-truth. Does this guarantee the correctness? Why or why not?
- ALU is just some combinational logic. Why does it need one cycle?
- How would the emperor handle an exception (e.g., integer overflow) using this framework, if the entry point of the corresponding exception handler is given?
- Can the emperor implement an out-of-order processor using this design? If he can, how should the lab framework be modified? If he can not, please briefly write down the major reason for him.
- Can the emperor implement a processor with RAW forwarding using this design? If he can, how should the lab framework be modified? If he can not, please briefly write down the major reason for him.

Submission

Create a folder with the name of lab2---, e.g., lab2-2018011222-xiaoming-wang. Put the following files in it. Compress the folder as a .zip package and upload to learn.tsinghua.edu.cn (网络学堂) by **April 24**. You may submit for multiple times, and we will grade based on the latest submission.

```
├── lab2-designdocument.pdf --- Your Lab 2 design document
├── lab2.S --- The assembly you used for emulating with your own proces
sor
├── CPU/* --- Your implementation of 5-stage processor. Remember that w
e will overwrite files unrelated in this handout and files marked with 
`Unmodified`.
```

Grading policy

Plagiarism is **strictly** forbidden in this lab. Peer discussion is **not** suggested. Contact TA if you are in trouble.

- 30% - Functionality: Correct implementation of all instructions from Lab 1.

- 40% - Timing: Lab 1 result with your verbal defense should be close to Lab 2 results.
- 30% - Design document.

Functionality

We will emulate your program to generate output pixel files with two test cases. One is a 8 * 8 small image and the other is a 256 * 256 big image. Then we compare them with the pre-processed ground-truth. More matches mean a higher correctness score. Note that we will use different image pairs for grading!

In this lab we will check your detailed processor implementation. Please have enough comments in your code.

If you did not finish Lab 1 completely, we will test your implementation with our own assembly program, with instructions including `lui, ori, add, syscall, j, jr, addiu, sw, sb, jal, sh, bne, beq, bgez, lw, lbu, lhu, sllv, addu, add, sll, sra, sltu`.

Timing

The cycle results from your own processor should be as close as your Lab 1 results. You can explain the reasons of the differences in your write-up.

Good luck, your Majesty!