Lab 3: Pipelined Processor - Part 1

Computer Organization 2024

1. Goal

A pipelined processor simultaneously executes multiple instructions during each clock cycle, which is much more complex than single-cycle processor that you have written in Lab 2. To make this lab easier and prepare for upcoming labs, you need to implement a pipelined processor **without hazard detection**. Since our testbench will examine hazards, you should NOT submit a hazard-free pipelined processor.

The main goal of this lab is to implement a pipelined processor **without stalling and forwarding** in textbook 4.6 which can execute a subset of 32-bit MIPS instructions. This lab will help you understand:

1. Concepts of pipelining.
2. How to pipeline datapath and control from single-cycle processor.
3. Different types of hazard and their behaviors in our implementation.

2. Description

We recommend you to complete this lab by:

1. Read textbook *"4.5 An Overview of Pipelining"* and understand the concepts of pipeline & hazard before you start this lab.
2. (Optional) Review your design in Lab 2 to recall the execution of instructions (except `lui` & `ori` since we will not implement them in this lab), especially how control signals are generated and how data flow through the datapath.
3. Read textbook (p.348-358) *"4.6 An Overview of Pipelining"* to understand how load & store instructions are executed in pipeline.
4. Finish reading textbook (p.359-363) *"4.6 An Overview of Pipelining"* to understand how the control signal are passed through pipeline registers for each stage.
    - FIGURE 4.46-49 describe the definition of each control signal and at which stage it is used.
    - FIGURE 4.50 shows the brief flow of control signals passed through stages.
5. Read textbook FIGURE 4.51 (also in Appendix) carefully because it is the processor your are going to implement in this lab.
   Take a look at the provided template code, and plan how to implement.
6. Implement all instructions mentioned below except `addi` .
   Then, test your design with provided test `0.s` like Lab 2 (checkout Lab2.pdf for more information about testing).
   (Optional) Try to test your design more detailed.
7. Implement `addi` then test your design by yourself.
8. Complete the report and have some ice cream. 🍦

Important Notes

- Same as Lab 2:

    - Instruction Memory (text) has 1 KB and starts at `0x00400000`, which should be your initial PC.
    - Data Memory (dynamic data) has 1 KB and starts at `0x10008000`, which your processor can access.
    - Please make sure that `reg_file.registers[0]` always be zero for testbench to check register file.
- In Lab 2, PC, register file & data memory are all written only when **positive edge** of `clk` comes. However, in Lab 3, we defined the clocking method to be:
    - **PC & pipeline registers** are written only when **positive edge** of `clk` comes.
    - **Register file & data memory** are written only when **negative edge** of `clk` comes.

  Don't worry! This we be handled in the template code as `clk` is negated before passed to `reg_file.clk` and `data_mem.clk`. Focus on pipeline controls first.

(%) Arithmetic Instructions & NOP

Same as Lab 2. Your processor need to support these instructions:

Instruction	Assembly	Meaning
`add`	`add rd, rs, rt`	`rd` = `rs` + `rt` (signed)
`sub`	`sub rd, rs, rt`	`rd` = `rs` - `rt` (signed)
`and`	`and rd, rs, rt`	`rd` = `rs` & `rt` (bitwise)
`or`	`or rd, rs, rt`	`rd` = `rs` | `rt` (bitwise)
`slt`	`slt rd, rs, rt`	`rd` = (`rs` < `rt`) ? 1 : 0
`nop`	`nop`	do nothing (translated into `sll $0, $0, 0`)

Same as Lab 2, you don't need to implement `sll`, but you have to make sure that your processor do nothing when executing machine code `0x00000000`.

Important Question: When do these instructions take place (write into register file)? If you write a register then immediately read from that register in the followed instruction, what content do you get? How about the next instruction?

(%) Load & Store

Same as Lab 2. Your processor need to support these instructions:

Instruction	Assembly	Meaning
`lw`	`lw rt, immd(rs)`	`rt` = mem[`rs` + `immd`] *(sign-extended)*
`sw`	`sw rt, immd(rs)`	mem[`rs` + `immd`] = `rt` *(sign-extended)*

Important Question: *Same as arithmetic instructions*.

(%) Branch

Same as Lab 2. Your processor need to support these instructions:

Instruction	Assembly	Meaning
`beq`	`beq rs, rt, immd`	if (`rs` == `rt`) PC = PC + 4 + (`immd` * 4) *(sign-extended)*

Important Question: If the branch is taken, will the instructions followed by `beq` be executed? How many? Why?

(%) Add Immediate `addi`

Different from `li` in Lab 2, `addi` is much easier to implement since the control signals and immediate calculation are simpler than `lui` & `ori` .

Instruction	Assembly	Meaning
`addi`	`addi rt, rs, immd`	`rt` = `rs` + `immd` *(sign-extended)*

(10%) Report

1. (2%) **Experimental Result**
    1. (1%) Show the waveform screen shot of the test we provided.
    2. (1%) What other cases you've tested? Why you choose them?
2. **Answer the following Questions**
    1. (3%) For each code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding.

Sequence 1	Sequence 2	Sequence 3

`lw $t0,0($t0)`
`add $t1,$t0,$t0`
	`add $t1,$t0,$t0`
	`addi $t2,$t0,#5`
	`addi $t4,$t1,#5`
		`addi $t1,$t0,#1`
		`addi $t2,$t0,#2`
		`addi $t3,$t0,#2`
		`addi $t3,$t0,#4`
		`addi $t5,$t0,#5`

    2. (1%) Explain the difference between throughput and latency.
    3. (4%) A group of students were debating the efficiency of the five-stage pipeline when one student pointed out that not all instructions are active in every stage of the pipeline. After deciding to ignore the effects of hazards, they made the following five statements. Which ones are correct? Explain why or why not.
        1. (1%) Allowing jumps, branches, and ALU instructions to take fewer stages than the five required by the load instruction will increase pipeline performance under all circumstances.
        2. (1%) Trying to allow some instructions to take fewer cycles does not help, since the throughput is determined by the clock cycle; the number of pipe stages per instruction affects latency, not throughput.
        3. (1%) You cannot make ALU instructions take fewer cycles because of the writeback of the result, but branches and jumps can take fewer cycles, so there is some opportunity for improvement.

    4. (1%) Instead of trying to make instructions take fewer cycles, we should explore making the pipeline longer, so that instructions take more cycles, but the cycles are shorter. This could improve performance.
3. (optional) **Problems Encountered & Solution**

   List some important problem you've met during this lab and there solution.

4. (optional) **Feedback**

   Any thing you want to say to TA team about this lab. How can we improve the lab?

3. Submission

our submission must be a zip file named `Lab3_ID.zip` where `ID` is your student ID, and structured as below:

```
Lab3_123456789.zip      # There should be NO sub-directory like Lab3_123456789/
├── Lab3_123456789.pdf  # Report (Must be PDF)
└── src                 # contains your source code
    ├── alu_control.v   # your ALU Control (if you have one) form Lab1
    ├── alu.v           # your ALU (and associated files) from Lab 1
    ├── control.v       # your Control (if you have one) form Lab1
    ├── data_mem.v      # it will be replaced when judging
    ├── instr_mem.v     # it will be replaced when judging
    ├── reg_file.v      # your register file from Lab 1
    └── pipelined.v     # Must included
```

There should be only one module per `.v` file, and the module name should be the same as file name.
The report should be named `Lab3_ID.pdf` and we ONLY accept PDF, any other format will not be scored.
If you want to use System Verilog, the filename must ends with `.sv` .
Do NOT include any testbench, test case and other irrelevant files in your submission.

Before you submit, make sure to pass the testbenches we provided.

!! Any Plagiarism is NOT allowed !!
Any late submission gets only 80% of original score.
Any submission after E3 window is closed will not be accepted!

Appendix: Design Diagram for This Lab

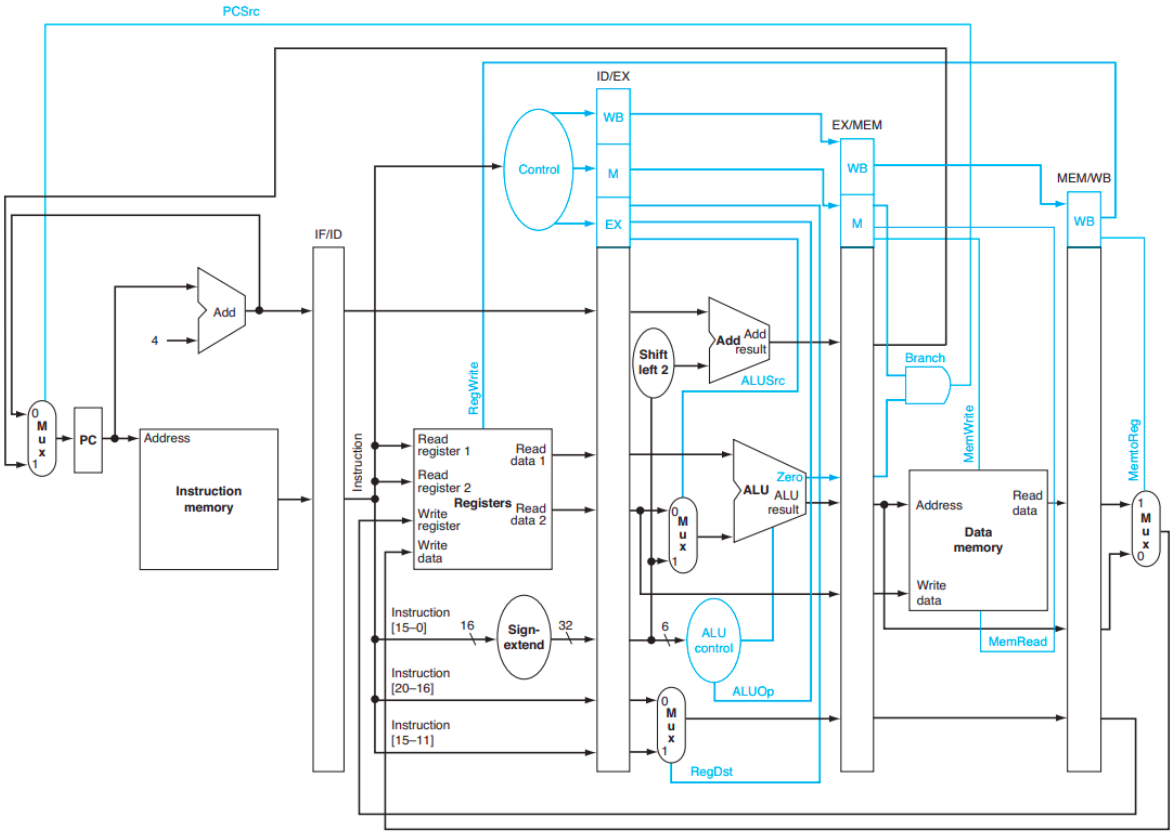

FIGURE 4.51 The pipelined datapath of Figure 4.46, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

Appendix: Test with Assembly Program

The testing flow is identical to Lab 2. Given the correct `.mem` file and `tb_pipelined.sv` can test your design.

You can use JsSPIM mentioned in Lab 2. However, you should be aware that the behavior of processor in Lab 3 is very different to which in Lab 2.