Lab 4: Pipelined Processor - Part 2

Computer Organization 2024

1. Goal

A good pipelined processor should detect data & control hazards, and solve them as efficient as possible. There are mainly two solutions: forwarding & stalling. Forwarding forwards the ALU result before it is written into the register file when it is needed by other instructions in the pipeline. When forwarding cannot solve the hazard, a "bubble" should be inserted into the pipeline and stalls some stages of the pipeline.

The main goal of this lab is to implement a pipelined MIPS processor which **handles hazards by forwarding & stalling** in textbook 4.7 & 4.8. This lab will help you understand:

1. Concepts of hazard in pipelined processor.
2. How to handle different types of hazards by forwarding & stalling.
3. Several conditions in 32-bit MIPS that causes hazards.

2. Description

We recommend you to complete this lab by:

1. Copy your design in Lab 3 into pipelined.v since you should modify it to complete this lab.
2. Read textbook (p.363-371) *"4.7 Data Hazards: Forwarding versus Stalling"*, and understand the concept of forwarding, when and how to forward.
3. Follow the instructions in forwarding.v and complete the design of forwarding unit.
   Connect it into your pipelined.v .
4. Read textbook (p.372-375) *"4.7 (cont'd) Data Hazards and Stalls"*, and understand the concept of stalling, when and how to stall.
5. Follow the instructions in hazard_detection.v and complete the design of forwarding unit.
   Connect it into your pipelined.v .
6. Test your processor by the provided test `0.s` .
   You may find out that your design cannot execute `beq` correctly.
   (Optional) Try to test with your own case generated from JsSpim, see Appendix for more information.
7. Handles `beq` correctly, detail is listed in Branch. This part is quite difficult.
8. Implement `addi` then test your design by yourself.
9. Complete the report and have some ice.

Important Notes

- Same as Lab 3:
  - 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.
    - Clocking:
        - 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.
- **Do NOT stall to much!**
  It is obvious that you can just stall every instructions to avoid hazards.
  However, our tests do have time limits. If you take to many cycles, the test might fail.

(40%) Arithmetic Instructions & NOP

Same as Lab 3. 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 3, 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: As mentioned in Lab 3, Read-After-Write might not get the proper content. Do you need to stall the pipeline? Or just forwarding is enough to handle ALU instructions?

(20%) 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: When the data loaded from memory is immediately read by another instruction, can forwarding solve the hazard? Or stalling is needed?

(20%) Branch

Your branch should have **"branch delayed slot"**.
That is, no mater the branch is taken or not, the next one instruction following the branch will be executed.
If the branch is taken, other instructions **after** the delayed slot should not be executed as in Lab 3.

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

Solving hazards related to branch is not detailed mentioned in the textbook.
By reading p.377-379 *"Reducing the Delay of Branches"* , we can disassemble this into the following steps:

1. Move branch target address calculation & taken or not from EX to ID.
2. Move branch decision from MEM to ID.

   It means that you have to update PC in ID stage.
3. Add forwarding for registers used in branch decision from EX/MEM (MEM stage).

   Since branch might read registers which are not written yet, forwarding to ID stage is needed.
   Important Question: Is forwarding from MEM/WB to ID needed?
4. Add stalling:
   - Branch read registers right after an ALU instruction writes it -> 1 stall

     Since the result of ALU can only be accessed since EX/MEM, a stall is needed.
     Why can't you forward the result from ALU to ID?
   - Branch read registers right after a load instruction writes it -> 2 stalls

     Since the result of load can only be accessed since MEM/WB, 1 stall is not enough.

(10%) Add Immediate `addi`

The hidden test of `addi` requires correct `beq` !

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

Hint: You only need to modify main control unit.

(10%) Report

1. (1%) **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. (8%) **Answer the following Questions**
   1. (2%) List out the equation to detect EX & MEM hazard in forwarding unit. Which part of the equation in textbook p.369 is wrong?
   2. (2%) In forwarding for `beq` , is forwarding from MEM/WB to ID needed? Why?
   3. (2%) Briefly explain how you insert 2 stalls when `beq` reads registers right after `lw` writes it.
   4. (2%) `sw` right after `lw` is quite common since copy and paste a data from one address to another is used frequently. In textbook, a stall is followed by the `lw` in this case. Is it possible to remove this stall? How?

3. Submission

our submission must be a zip file named `Lab4_ID.zip` where `ID` is your student ID, and structured as below:

```
Lab4_123456789.zip          # There should be NO sub-directory like
Lab4_123456789/
├── Lab4_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
    ├── forwarding.v        # your Forwarding Unit (if you have one)
    ├── hazard_detection.v  # your Hazard Detection Unit (if you have one)
    ├── 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.
Includes all files needed to make sure your design can compiles.
The report should be named `Lab4_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: Approximate Design Diagram for This Lab

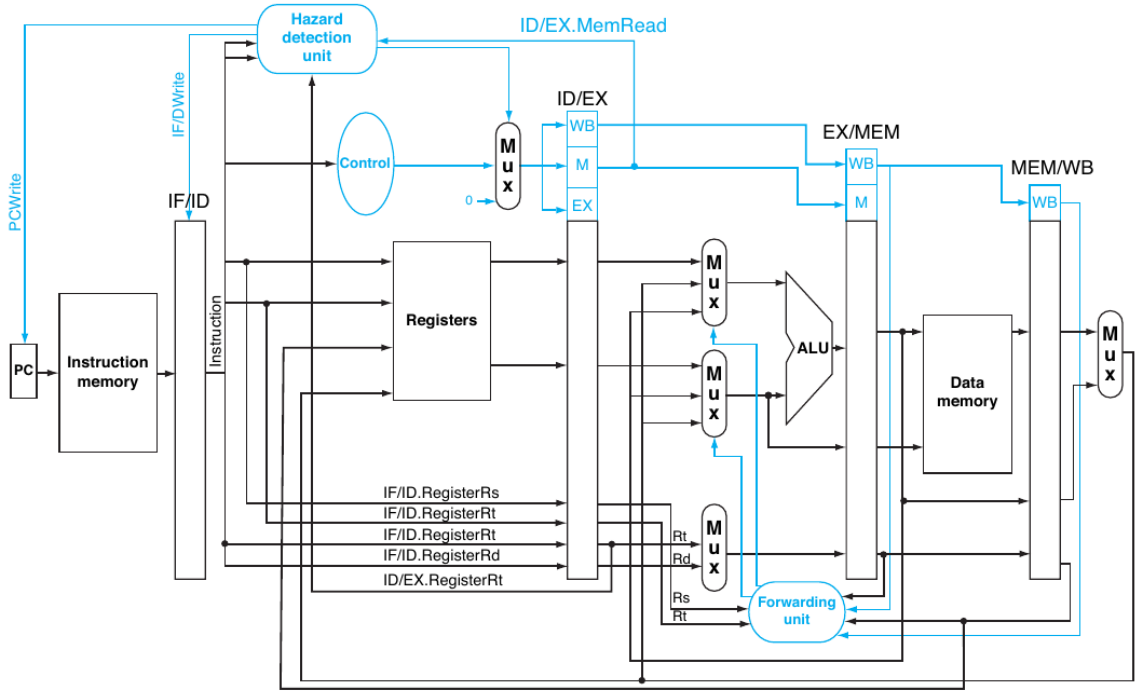

FIGURE 4.60 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

Be careful that this diagram did not show how to handle branch properly.

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 the JsSPIM designed for Lab 4, whose difference to Lab 2 is that the instruction in branch delayed slot will e executed no mater the branch is taken or not.
Your design should have the exactly same outcome with JsSPIM!

Be careful to use the correct JsSPIM for Lab4 .