Bilkent University

CS

CS224

Design Report

Lab 5

Section-1

Ata Seren

21901575

12.04.2021

b and c-) Hazards that can occur in this pipeline:

- Compute-use (Data hazard)
  - This hazard happens when a data resulted from an instruction is not written to register file before next instruction uses it via a register.
  - Example code:
    ```
    add $s0, $t0, $t1
    and $s1, $s0, $t2
    ```
    Second instruction uses data at `$s0` but correct data is not written to `$s0` register by first instruction.
  - In **decode** stage, wrong data will be retrieved. Because of this, **execute** and **writeback** stages will use wrong data and therefore, 3 stages will be affected from this hazard.
  - To solve this hazard:
    - `nop` instructions can be used after problematic instruction to save data to register file
    - Data result of first instruction can be forwarded to execute stage, instead of writing and retrieving it from register file
- Load-use (Data hazard)
  - This hazard happens when an instruction attempts to use a register which previous `lw` instruction didn't retrieve and write desired data to register yet.
  - Example code:
    ```
    lw $s0, 0($t0)
    add $s1, $s0, $t1
    ```
    Second instruction uses `$s0` register but desired data is not written to `$s0` yet.
  - In **decode** stage, data will be acquired late by `lw` instruction. Because of this, **execute** stage won't make a correct computation. Also, because of this wrong computation, correct data may not be acquired from **memory**.
  - To solve this hazard:
    - `nop` can be used again but it will be not efficient.
    - **Forwarding** is an efficient way but it can't solve the hazard for this case. Because memory stage comes after execute stage. Therefore, correct data can't be forwarded to execute because it is too late.

        - By **stalling** next instruction, execute stage will be delayed until correct data is fetched from memory.
- Load-store (Data hazard)
    - Like load-use, this hazard happens when lw is not complete yet. However, in this hazard, sw instruction can't store correct instruction because of lw.
    - Example code:
      ```
      lw $s0, 0($t0)
      sw $s0, 4($t0)
      ```
      `sw` instruction can't store correct data because `lw` didn't fetch desired data yet.
    - In **decode** stage, data will be acquired late by `lw` instruction. Because of this, correct data can't be stored in **memory.**
    - To solve this hazard:
        - `nop` and forwarding result same like load-use hazard.
        - By **stalling** next instruction, memory stage will be delayed until correct data is fetched from memory.
- Branch (Control hazard)
    - This hazard happens when branch decision is not done until the next instruction is retrieved from the instruction memory.
    - Example code:
      ```
      beq $s0, $s1, branch
      add …
      …
      branch: …
      ```
      If `$s0` and `$s1` are equal, there must be a branch. However, instructions after `beq` can be processed even if there should be a branch and instructions should not be processed between `beq` and its label.
    - If branch hazard happens, instructions that should not be processed will be processed and this can affect all stages and results.
    - To solve this hazard:
        - A hardware implementation and flushing can solve the problem. For example, in PipelineDatapath.PNG file, a comparator that compares register values has been added to **decode** stage. Then, if there is a need for branch, pipe flushes next instruction and desired instruction will be

fetched. In this way, `beq` instruction can be processed before execute stage and be more efficient. Note: This implementation is called **early branch prediction.** This type of implementation can cause a RAW data hazard like load-use or compute-use. However, MUX implementation in execute stage solves this problem. There are other types of branch implementations too.

d-) Logic equations for each signal output by the hazard unit (I got them from book):

- For forwarding
  - if ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)

    ForwardAE = 10

    else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW)

    ForwardAE = 01

    else

    ForwardAE = 00
  - Logic equations for ForwardBE is same with ForwardAE except that ForwardBEchecks `rt` rather than `rs`.
  - ForwardAD = (rsD != 0) AND (rsD == WriteRegM) AND RegWriteM
  - ForwardBD = (rtD != 0) AND (rtD == WriteRegM) AND RegWriteM

- For stalling
  - lwstall = ((rsD = = rtE) OR (rtD = = rtE)) AND MemtoRegE
  - branchstall = BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD) OR BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD)
  - StallF = StallD = FlushE = lwstall = branchstall

e-) Test codes:

Code with no hazard:

addi $t0, $0, 1	8'h00: instr = 32'h20080001;
addi $t1, $0, 2	8'h04: instr = 32'h20090002;
addi $t2, $0, 3	8'h08: instr = 32'h200a0003;
beq $0, $t0, 5	8'h0c: instr = 32'h10080001;
add $t5, $t0, $t6	8'h10: instr = 32'h010e6820;

and $s0, $t7, $t1	8'h14: instr = 32'h01e98024;
or $s1, $t1, $t2	8'h18: instr = 32'h012a8825;
lw $s2, 12($t0)	8'h1c: instr = 32'h8d12000c;
sub $s0, $s0, $t3	8'h20: instr = 32'h020b8022;
nop	8'h24: instr = 32'h58000000;
sw $s2, 0($t2)	8'h28: instr = 32'had520000;
add $s1, $s2, $0	8'h2c: instr = 32'h02408820;

Code with compute-use hazard:

add $s0, $t0, $t1	8'h00: instr = 32'h01098020;
sub $s1, $s0, $t0	8'h04: instr = 32'h02088822;
add $s1, $s0, $s1	8'h08: instr = 32'h02118820;

Code with load-use hazard:

lw $s0, 0($s1)	8'h00: instr = 32'h8e300000;
and $s2, $s0, $s3	8'h04: instr = 32'h02139024;
or $s4, $s5, $s0	8'h08: instr = 32'h02b0a025;

Code with load-store hazard:

lw $s0, 0($t0)	8'h00: instr = 32'h8d100000;
sw $s0, 4($t1)	8'h04: instr = 32'had300004;

Code with branch hazard:

beq $s0, $s1, 3	8'h00: instr = 32'h12110002;
addi $t0, $zero, 10	8'h04: instr = 32'h2008000a;
add $t2, $s0, $zero	8'h08: instr = 32'h02005020;
and $t1, $s1, $s2	8'h0c: instr = 32'h02324824;
add $t3, $t4, $t3	8'h10: instr = 32'h018b5820;
slt $t0, $s0, $s1	8'h14: instr = 32'h0211402a;