CS224 - SPRING 2023 -

Preliminary Design Report

Lab-05

Section 5

Tolga Han Arslan

22003061

8.05.2023

Part 1. Preliminary Work (50 points)

b) [10 points] The list of all hazards that can occur in this pipeline. For each hazard, give its type (data or control), its specific name ("compute-use" "load-use", "branch" etc.), the pipeline stages that are affected.

Data Hazards:

1. **Compute Use:** This type of hazard occurs when an instruction writes to a particular register and the subsequent instruction reads that register. The subsequent register tries to read data from the register before the WB stage of previous instruction is done, hence it reads incorrect data value.
   Decode, Execute and Writeback stages are affected. Decode stage reads incorrect value since previous instructions Writeback is not done, Execute and Writeback stages perform calculations with wrong data values.
   addi $t0, $zero, 5
   addi $t1, $t0, 7

   The addi instruction writes $t0 in the first half of cycle 5, but the subsequent addi instruction tries to read $t0 in cycle 3 (its execute stage), hence it reads a wrong value.

2. **Load Use:** This type of hazard occurs when there is a lw instruction and subsequent instruction tries to read the register that the lw instruction writes. The memory stage of lw instruction is not done when the subsequent instruction is trying to read that register.
   lw $t1, 0x60($zero)
   addi $t2, $t1, 4
   The addi instruction tries to read $t1 register's value at cycle 3 but memory stage of the previous lw instruction is not done yet (cycle 4) , hence it reads incorrect value.

   Execute stage is affected in this type of hazard since it performs operation with reading incorrect values.

3. **Load Store:** This type of hazard can occur if a store word instruction uses the same register that the previous load word instruction uses.

   lw $t0, 60($zero)

sw $t0, 60($zero)

Here, lw instruction's memory stage is not done yet when sw instruction tries to read the value of $t0. Hence, it reads a wrong value. Memory stage is affected because an incorrect value is written to the memory.

Control Hazards:

1. **Branch:** This type of hazard occurs particularly with branch instructions. Branch decision is not done until the end of memory stage, hence previous stages might have the instruction values that needs to be branched, which causes a delay in the pipeline.

   20 beq $t1, $t2, 40

   24 and $t0, $s0, $s1

   28 or $t1, $s4, $s0

   2C sub $t2, $s0, $s5

   …

   64 slt $t3, $s2, $s3

   Delay of the branch decision causes a fetch of unnecessary 3 instructions (branch misprediction penalty), particularly "and", "or", "sub" instructions here. Hence, fetch, decode, and execute stages are affected since the branch decision is completed at the end of the memory stage (3 cycles).

c) **[10 points]** For each hazard, give the solution (forwarding, stalling, flushing, combi-nation of these), and explanation of what, when, how

Solutions:

1. **How to solve Compute Use:** New values of registers which are available after the end of alu computation can be forwarded the next instructions execute stage with the help of new hardware (multiplexers) and their control signals ForwardAE and ForwardBE. Therefore, correct values will be used in the next instruction.

2. **How to solve Load Use:** This type of hazard cannot be solved with forwarding since lw instruction does not finish reading the data until the end of memory stage. Instead, pipeline can hold up the operation until the data is available (stalling). Stalling can be done by adding enable inputs to fetch and decode pipeline registers and a clear input to execute pipeline register. In case of lwstall, decode and fetch registers hold their values, and flush signal is asserted to clear the contents of the execute stage pipeline register.
3. **How to solve Load Store:** Similarly with load use, stalling can be used to solve this type of hazards.
4. **How to solve Branch:** One possible solution is stalling the pipeline until the branch decision is made. But, this will degrade the system performance since the pipeline would have to be stalled for 3 cycles at every branch. Instead, branch misprediction penalty can be reduced to 1 by making the branch decision at an earlier stage (decode) with faster hardwares (equality comparator). Instructions between the branch and target need to be flushed if branch is done. Similarly with lwstall, branchstall operation requires new clear and enable signals for pipeline registers so that incorrectly fetched instruction can be flushed when a branch is taken. Furthermore, a new RAW data hazard in decode stage occurs because of early branch decision. This hazard can be solved with forwarding the result (alu instruction in memory stage) to the equality comparator with two new multiplexers. But if the result of alu instruction is in the execute stage or the result of a lw instruction is in the memory stage, pipeline must be stalled at the docde stage until the result is ready.

d) **[10 points]** Give the logic equations for each signal output by the hazard unit, as a function of the input signals that come to the hazard unit. This hazard unit should handle all the data and control hazards that can occur in your pipeline (listed in b) so that your pipe-lined processor computes correctly.

Data Forwarding Logic:

$$if\ \big((\ rsE\ !=0)\ AND\ (rsE == WriteRegM)\ AND\ RegWriteM\big)\ then$$

$$ForwardAE = 10$$

$$else\ if\ \big((rsE\ !=0)\ AND\ (rsE == WriteRegW)\ AND\ RegWriteW\big)\ then$$

$$ForwardAE = 01$$

$$else \qquad ForwardAE = 00$$

- Forwarding logic for SrcB (ForwardBE) is identical except that it checks rt rather than rs.

Data Stalling and Flushing Logic:

lwstall = ((rsD == rtE) OR (rtD == rtE)) AND MemtoRegE

StallF = StallD = FlushE = lwstall

Control Forwarding Logic:

ForwardAD = (rsD ! = 0) AND (rsD == WriteRegM) AND RegWriteM

ForwardBD = (rtD ! = 0) AND (rtD == WriteRegM) AND RegWriteM

Control Stalling Logic:

branchstall =

BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD)

OR

BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD)

$$StallF = StallD = FlushE = lwstall\ OR\ branchstall$$

e) **[20 points]** Think about extending the pipelined processor with the instruction corre-sponding to your section on the table given below. The instructions are either identical or very similar to the ones you implemented in Lab 4. Write down all hazards the new instruc-tion can cause, and their solutions. You can refer to the hazards you listed in Part 1-b if some of them also occur with the new instruction. Then, for each hazard you listed, write a small test program in MIPS assembly that will show whether the pipelined processor is working even in the presence of the hazard. You should also write a test program with no hazards. The tests should verify that the new instruction works correctly under any circumstances. **Note that, your design should avoid stalls if it is possible to use forwarding. Any kind of extra stall unnecessarily will cause you to lose points.**

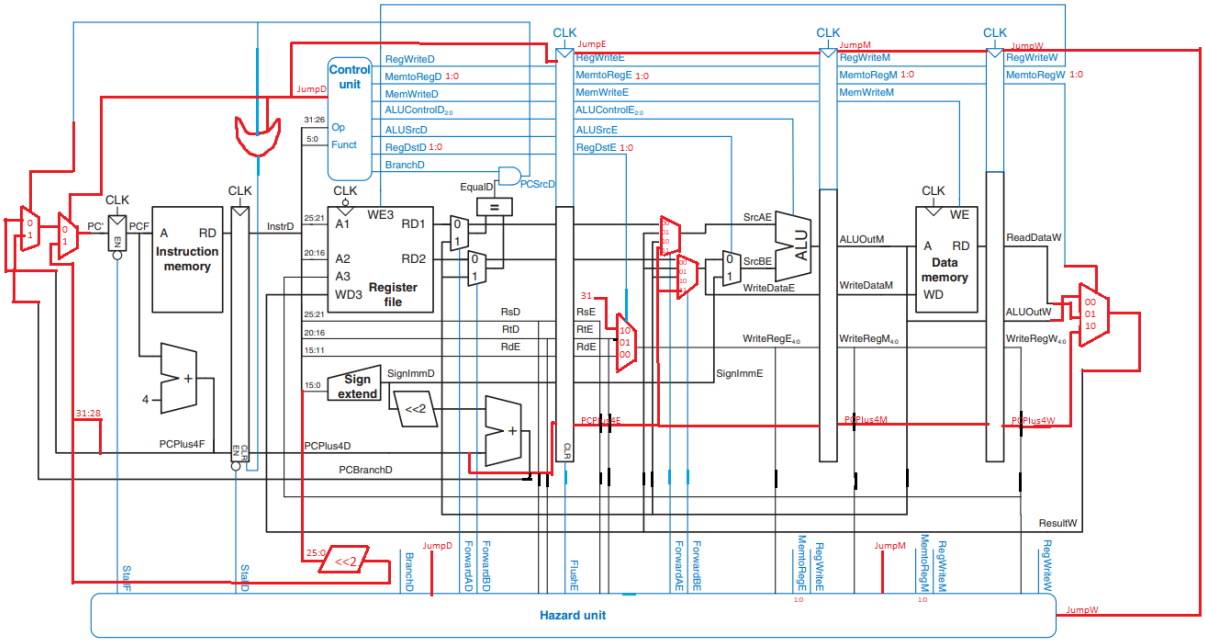

Figure 7.58 Pipelined processor with full hazard handling

Jal instruction is similar to jump instruction except the fact that it writes the register $ra. In order to implement jal instruction pipelined datapath are modified in above image and controller should be modified with a new jumpD signal as well as the changed two bit RegDstD and MemtoRegD signals. Above implementation of jal (jump and link) instruction cause **two new hazards**.

First one, similar to the beq instruction jump takes in decode stage, so the subsequent stage in the fetch stage must be flushed. To solve this, stalling is used with the new jumpD signal and hazard control unit modified accordingly.

Second hazard is about the read of $ra register after the jal instruction. This is similar to compute use hazards in part-b and can be solved by forwarding the $ra register's value to subsequent instructions. Since $ra register's value is determined in the fetch stage (PCPlus4F) this value will be held by pipeline registers and forwarded if any subsequent instruction reads $ra register. Hazard control unit for ForwardAE and ForwardBE is modified to solve this issue along with the multiplexers, also jumpD signal is held for all stages to implement correct logic in hazard unit. Modified hazard control unit is given.

New hazard control unit:

Data Forwarding Logic:

$$\text{if}((\text{rsE} != 0)\ \text{AND jumpM AND}\ (\text{rsE} == 31))\ \text{then}$$

$$\text{ForwardAE} = 11$$

$$\text{else if}((\text{rsE} != 0)\text{AND jumpW AND}\ (\text{rsE} == 31))\ \text{then}$$

$$\text{ForwardAE} = 11$$

$$else\ if\ ((rsE != 0)\ AND\ (rsE == WriteRegM)\ AND\ RegWriteM)\ then$$

$$ForwardAE = 10$$

$$else\ if\ ((rsE != 0)\ AND\ (rsE == WriteRegW)\ AND\ RegWriteW)\ then$$

$$ForwardAE = 01$$

$$else \qquad ForwardAE = 00$$

- Forwarding logic for SrcB (ForwardBE) is identical except that it checks rt rather than rs.

Data Stalling and Flushing Logic:

lwstall = ((rsD == rtE) OR (rtD == rtE)) AND MemtoRegE == 01

StallF = StallD = FlushE = lwstall

Control Forwarding Logic:

ForwardAD = (rsD ! = 0) AND (rsD == WriteRegM) AND RegWriteM

ForwardBD = (rtD ! = 0) AND (rtD == WriteRegM) AND RegWriteM

Control Stalling Logic:

branchstall =

BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD)

OR

BranchD AND (MemtoRegM == 01) AND (WriteRegM == rsD OR WriteRegM = = rtD)

StallF = StallD = FlushE = lwstall OR branchstall OR jumpD

Test cases for jal instruction:

No hazard:

0x00 addi $ra, $ra, 0

0x04 jal 0xc

0x08 nop

0x0c addi $t0, $t1, 12

0x10 addi $t1, $t2, 4

```
0x14    addi    $t3, $zero, 0
0x18    addi    $t3, $ra, 0         #$t3 should contain $ra value 0x08
```

With hazards:

```
0x00    addi $ra, $ra, 0
0x04    jal 0x08
0x08    addi $t0, $ra, 0            #wrong read $ra register
0x0c    addi $t1, $ra, 0            #wrong read $ra register
0x10    addi $t2, $zero, 4
0x14    jal 0x1c
0x18    addi $t3, $zero, 0          #this must be flushed
0x1c    add $t4, $t3, $t2
0x20    add $t5, $t4, $t3
```