Bilkent University

Computer Engineering

Computer Architecture

Design Report

Lab 05

CS224-006

Izaan Aamir

22001488

12th April, 2022

b) 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", "load-store", "branch" etc.), the pipeline stages that are affected.

1. **Branch Hazard**
   Type: Control Hazard
   Pipeline Affected Stages: 3 instructions are computed unnecessarily if the branch is taken. This happens due to the delay in branch decisions.

2. **Load-store Hazard**
   Type: Data Hazard
   Pipeline Affected Stages: The data stored into the memory uses the wrong data value to be stored so it affects the memory stage.

3. **Load-use Hazard**
   Type: Data Hazard
   Pipeline Affected Stages: Execute and Memory stages are affected due to the latency and it uses the wrong data values.

4. **Compute-use Hazard**
   Type: Data Hazard
   Pipeline Affected Stages: The decode stage will be affected by this hazard as the newer (updated) values haven't been read from the register file (RF). This causes the execute and the write-back stages to be affected as well as it uses the wrong data values coming from the decode stage.

c) For each hazard, give the solution (forwarding, stalling, flushing, combination of these), and explanation of what, when, how

1. **Branch Hazard**

   This hazard occurs because the branching decision is made in the memory stage. This causes the next 3 instructions to be fetched. In the case that the instruction is to jump, the 3 instructions are loaded unnecessarily. This hazard occurs when the branch instruction is computed at the end of the memory stage.

   Solution: There are two possible solutions to it. One is to use an additional comparator to make the branch decision earlier in the decode stage and flush the unnecessary instructions if it is a jump instruction. We can also stall the pipeline for 3 clock cycles as well whenever it is a branch instruction.

2. **Load-store Hazard**
This hazard occurs when a load instruction such as lw is followed by a store instruction such as sw. Due to latency, the correct data is not accessed by the subsequent instruction and causes incorrect data to be written to memory. This hazard occurs when we have the load and store instruction following one another using the same register.

Solution: The solution is to stall the pipeline until the correct data values are fetched from memory.

3. **Load-use Hazard**
This hazard occurs when data from the memory needs to be accessed and written back to the register. For instructions like lw, they are not written back to the register file until the write-back stage. This causes incorrect values to be accessed for the subsequent instructions using those register values. This hazard occurs when an instruction uses the values of a register right after the register is used to load the values from the memory.

Solution: The solution is to stall the decode and the fetch stage for the next 2 instructions until the correct values are available. We need to flush these instructions execute stages as well.

5. **Compute-use Hazard**
This hazard occurs when the correct values have not been updated in the register file until the write-back following a previous instruction that used that same register. This causes the next instruction to use the wrong data values.

Solution: The solution is to forward the data from the execute or write-back stage of the current instruction to the next instruction's execute stage. Alternatively, we can also use stalling.

d) 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 pipelined processor computes correctly

<u>**1. Stalling and Flushing**</u>

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

StallF = StallD = FlushE = lwstall

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)*

<u>2. Forwarding</u>

- <u>ForwardAD:</u>

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

- <u>ForwardBD:</u>

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

- <u>ForwardAE:</u>

```
if ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)
    then ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW)
    then ForwardAE = 01
else
        ForwardAE = 00
```

- <u>ForwardBE:</u>

```
if ((rtE != 0) AND (rtE == WriteRegM) AND RegWriteM)
    then ForwardBE = 10
else if ((rtE != 0) AND (rtE == WriteRegW) AND RegWriteW)
    then ForwardBE = 01
else
        ForwardBE = 00
```

e) Think about the implementation of sracc instruction in pipelined MIPS processor. Write down all hazards that sracc might cause, and their solutions. If some of these hazards are similar to the ones you listed in Part 1-b, you can refer to them. 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 sracc works correctly under any circumstances.

<u>sracc hazards:</u>

sracc is an R-type instruction which uses an additional read port as well as an adder to compute the result. Firstly, it will cause a compute-use hazard as the result of the sracc instruction will be saved to the rd register in the write-back stage and the next instruction won't be able to use its value right afterwards if it uses the rd register of the sracc instruction as a rf or rs register value.

Since the sracc instruction uses an adder to compute the result after the ALU computation after the execution stage. This adder overlaps with the memory stage which implies that the adder computation is not completed before the memory stage hence we cannot forward the data from the execute stage to the next instruction as it will use incorrect data values in its calculations causing another data hazard.

Solution: The solution is to use forwarding to use the result of the adder from the memory and write-back stage and use it in the second and third instructions. For the instruction right after the sracc instruction, we need to stall the pipeline for one clock cycle. Then we can use forwarding to send the data value to the execute stage of the instruction. Furthermore, we will need to use an additional signal in the hazard unit which will be inputted to the rs/rd mux as well as the adder mux. This will resolve the issues that we are facing in due to compute-use hazard.

<u>MIPS ASSEMBLY PROGRAM:</u>

SRACC HAZARDS:

```
addi $t0, $zero, 0x0001

addi $t1, $zero, 0x00CC

addi $t2, $zero, 0x0007

sracc $t2, $t1, $t0

sw $t2, 0 ($t1) //$t2 == 0x0007?
```

COMPUTE-USE HAZARDS:

```
add $t1, $zero, $zero

addi $t0, $zero, 0x00CC
addi $t1, $t0, 0x0002 //$t1 == 0x00CC?
and $t2, $t0, $t1 //$t2 == 0x00CE
```

LOAD-USE HAZARDS:

//Assuming word stored at $t1 = 0xFFEE

```
addi $s0, $zero, 0x0010
lw $s0, 0 ($t1)
add $s2, $zero, $s0 //$s2 == 0x0010 OR 0xFFEE?
```

LOAD-STORE HAZARDS:

```
addi $t0, $zero, 0x0010
addi $t1, $zero, 0x00CC
sw $t0, 0($t1)
lw $s0, 0 ($t1)
add $s0, $zero, $s0 //$s0 == 0x0010 OR 0x00CC?
```

BRANCH HAZARDS:

```
addi $t0, $zero, 0x0000
addi $t1, $zero, 0x0001
beq $t0, $zero, 0x0001
addi $t1, $t1, 0x0010 //Skipped
add $t1, $0, $t1 // //$t0 == 0x0011 OR 0x0001?
```

NO HAZARDS:

```
addi $t0, $zero, 0x002
addi $t1, $zero, 0x00F0
addi $t2, $zero, 0x0000
addi $t3, $zero, 0x0112
addi $t4, $t0, 0x0002

sw $t1, 0 ($t0)

sracc $t3, $t4, $t0

beq $t2, $zero, 0x0004

addi $t2, $t2, 0x0010

slt $t5, $t1, $t3

lw $t6, 0 ($t0)
```