Bilkent University

Computer Engineering

CS-224

Design Report

Lab 05

Section 01

Maryam Shahid

21801344

20th December 2020

Part 1

b) Hazards that can occur in this pipeline are as follows:

Data hazards:

1. **Compute-use**

   Decode stage of the instruction is affected. This is because incorrect data is fetched from register. Thus execute and write back stage will carry out operations using incorrect data values.

2. **Load-use**

   Execute and memory stages can be affected if memory is written. This is due to two late clock cycle instructions.

3. **Load-store**

   This will affect the memory stage as incorrect data will be stored in the memory location from the register.

Control hazards:

1. **Branch**

   Since branch stage will be made in the memory stage, it will cause a delay and three instructions will be fetched unnecessarily in case branch is taken.

c) For each hazard, the following solutions and information can be given:

1. **Compute-use**

   Solution: Instead of waiting for the write back stage, data can be forwarded to execute stage of next instruction which will allow the following instruction to use the correct data value. Delaying the pipeline can also be used.

   What, when and how: This hazard happens when after execute stage, data is not written for write back stage. This causes the next instruction to read incorrect data from the previous instruction's destination register during its decode stage. To understand when this hazard occurs, we can use a given R type instruction add as an example. This is when

the R type instruction's rd is not yet written due to incompletion of the previous instruction's computation.

2. **Load-use**

   Solution: Since this hazard cannot be solved by forwarding, stalling the pipeline till data is present is a better option.

   What, when and how: Instructions that require reading data from memory cannot load data from till the end of the memory stage. Because of this, the next instruction is unable to access data at execute stage

3. **Load-store**

   Solution: Stalling pipeline is an efficient way of allowing memory load till the next instruction gets data at its decode stage.

   What, when and how: This occurs when data is required to be stored right after being loaded from memory. This happens when successive lw and sw instructions are called using the same rt register.

4. **Branch**

   Solution: A solution for this hazard can be to stall the pipeline for three cycles whenever a branch instruction is executed. Another way to fix this can be to use additional hardware and make the branch decision earlier in the pipeline. Flushing unnecessarily executed instructions is also required to avoid branch misprediction.

   What, when and how: Branch decision is made in the memory stage which causes a delay. This results in the next instruction being fetched beforehand. It happens when a branch decision is required at the end of memory stage.

d) Logic equations for each signal output by the hazard unit are the following:

Logic equation for forwarding

```
if (rsE != 0)
  if (rsE == writeRegM AND regWriteM) then
    forwardaE = 10
  else if (rsE == writeRegW AND regWriteW) then
    forwardAE = 01
if (rtE != 0)
  if (rtE == writeRegM AND regWriteM) then
    forwardbE = 10
  else if (rsE == writeRegW AND regWriteW) then
    forwardbE = 01
```

Logic of equation for stalling and flashing

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

stallF = stallD = flushE = lwstall

e) Test programs:

// no hazard

```
addi $s0, $0,  7
addi $s1, $0,  5
addi $s2, $0,  0
addi $s3, $0,  15
and  $s2, $0,  $s1
or   $s2, $s0, $s1
and  $s2, $s0, $s1
sub  $s2, $s0, $s1
slt  $s2, $s0, $s1
sw   $s0, 2($s1)
lw   $s1, 0($s0)
beq  $s0, $0, 1
addi $s2, $0, 10
addi $s1, $0, 12
```

// compute use hazard

```
addi $s0, $0,  6
addi $s1, $s0, 5
add  $s2, $s1, $s0
```

// load-use and load-store hazard

```
addi $s0, $0, 18
addi $s1, $0, 8
addi $s2, $0, 5
addi $s3, $0, 4
sw $s0, 0($s1)
lw $s1, 0($s0)
add $s2, $s1, $s2
sub $v0, $v1, $s3
```

// branch hazard

```
addi $s1, $0, 1
beq  $s1, $0, 2
addi $s2, $0, 3
addi $s2, $t0, 4
addi $s2, $t0, 5
addi $s2, $0, 6
sw $s2, 6($0)
```