Preliminary Work for Lab05

Ali Eren Günaltılı

21801897

CS 224 – 04

CS

Pipeline Stages = IM-Decode-Exe-Dmem-WBack

Part B-) List of all hazards that can occur and affected stages

-Structural Hazards: which means one memory is available but in our case this is neglectable since we have two different unique memory for instruction and data.

-**Data Hazards**: Data dependencies, read after write hazards, load-use hazards, store-use hazards.

-**Control Hazards**: Branch hazards

I write affected stages by considering the incorrect version of hazards. I only try to demonstrate where the hazards occurs at the first place among the stages.

a.RAW (read after write)

Affected Stages = decode stage is affected since we are reaching the wrong value from register file at that current time. For instance if we consider these two lines :

Number1 = add $t0, $t1, $t2

Number2 = add $t3, $t0, $t2

t0 register contain wrong value since we haven't write back the data to t0 when we reach the decode stages of instruction number2. Therefore, we fetch the wrong value of t0 from number2's decode stage. As decode stage is wrong, it makes mistake for every stages after that. In execution we calculating the wrong two value and in write back we write the wrong value to given register.

b.Load-Use

Affected Stages = In case of affecting the stages it is very similar with RAW since we still trying to reach the wrong value from decode stage.

Although affected stages are similar they have two different solution as we discuss later.

c.Store-Use

Affected Stages = Same as two previous hazards. Decode stage is affected because of that every stages come after that gives wrong results.

d.Branch Hazard

Affected Stages = As we predicted the pc's value it may cause 3 instructions loss after we proceed them. It affects the next 3 instruction's stages.

Part C-) Data Hazards and their explanations

a.RAW

It happens when we trying to use register just after we trying to writeBack to it. An example code lines are:

```
Add $t0 , $t1, $t2
Sub $t3, $t0, $t1
```

This happens since when we are decode stage of the sub instruction we haven't write back the right value to t0. In other word, t0 value has not updated and isn't right when we try to fetch from register file at sub instruction. We can solve it by forwarding. Since we need the right instruction at sub's execution stage and we can take it from the previous ALU's result stalling is not needed. But hazard can also be solved with stalling too.

b.Load-Use

It happens when we trying to use right after loading some value to it.

```
Lw $t0 , 0($t1)
Add $t2, $t0, $t3
```

At this point hazard occurres since when we decode stage of add instruction fetching the t0 value from register file we take the wrong value since we haven't write back the value to t0.

We can solve this hazard by using both hazard and stalling. Only using forwarding is not enough since we have the value to be written t0 at the end of

the data memory stage for load word instruction we cannot directly forwarding to the add instruction . We have to stall till the forwarding is available.

c. Store-Use

If we try to reach value right after store it this error occurres.

Sw $t1, 0($t0)

Add $t2, $t1, $t3

We try to reach the wrong t1 value at add instruction line since we haven't write back 0($t0) to the t1 register. We reach the value at the end of the Dmem stage but we need it into the add instruction's execution. To do that we need to stall then forwarding it.

d.Branch Hazard

Branch instruction cause hazard since in the pipeline processor we always take the pc ← pc + 4 automatically. However, if the branch instruction make pcSrc 1, we have to take pc ← pc + 4 + BTA. To do that at the first place we have to consider branch instruction as a normal instruction and make prediction in a way until we check the given condition is satisfied or not. If given condition is satisfied then we have to flush undesired instruction and put right value to the pc.

10 Beq $t1 , $t2, 20

14 And $t0 , $s1, $t3

18 Or $t1, $s4, $s0

1c Sub $t1, $s0 , $s5

34 add $t1, $t1, $0

By given datapath png we can see that our beq condition's result will be available at the end of the decode stage. What it means is, we only go one clock cycle further so we only take the and instruction at that point. If t1 == t2 is satisfied we flush the and instruction and make the pc ← 34 as beq instruction suggests. But this solution also cause another problem. As we take the t1 and t2 from their decoded stage, what happens if we change one of them previous instruction. For instance:

Add $t0, $t2 ,$t3

Beq $t0 , $t1 , Done

By considering this order we can say that when add instruction is in EXE stage our beq instruction is inside the Decode stage. As we haven't write the right value to t0 that cause error and as we cannot directly forwarding it we have to stall one step to make forwarding available for t0's real value.

Part D – Logic Equations for Each Signal Output

Hazard Unit Logic for Forwarding

For rs in execution stage:

if((rsEXE != 0) && (rsEXE == WriteRegM) && RegWriteM)

ForwardAE = 10 // this case also means AE = 11 too since we have choose from the closest stage.

else if((rsEXE != 0) && (rsEXE==WriteRegW) && RegWriteW)

ForwardAE = 01

else

ForwardAE = 00

For rt in execution stage:

if((rtEXE != 0) && (rtEXE == WriteRegM) && RegWriteM)

ForwardBE = 10

else if((rtEXE != 0) && (rtEXE==WriteRegW) && RegWriteW)

ForwardBE = 01

else

ForwardBE = 00

For Branch instruction checker in decoder:

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

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

Hazard Unit Logic for Stalling and Flushing

lwstall = ((rsD == rtE) || (rtD == rtE)) && MemtoRegE

stallF = stallD = FlushE = lwstall

Part E

With No Hazards

```
add $t0 , $t1, $t2
sub $t3, $t1, $t2
sll $t1, $t1 , 20
or $t0 , $t0 , $t1
add $t3, $0 , $0
sw $t3, 0($t1)
done:
add $t3, $t2, $t2
lw $t0, 0($0)
addi $t1 , $0 , 5
addi $t4, $0, 6
beq $t1, $t4, done
addi $t0 , $t0, 5
```

With RAW Hazard

```
add $t0 , $t1 , $t2
sub $t1, $t0 , $t3
```

With Load-Use Hazard

```
lw $t1 , 0($t0)
add $t2, $t1, $t3
```

With Store-Use Hazard

```
sw $t2, 0($t0)
lw $t3, 0($t0)
```

With Branch Hazard

```
addi $t1, $0 , 6
beq $t2, $t3 , exit
add $t0 , $t0 , $t0
or $t1, $t0 , $t1
sll $t4, $t4 , 10
exit:
```

Another Type of Branch Hazard

```
addi $t1, $0 , 6
beq $t1, $t0 , exit
add $t2, $t4, $0
sub $t4, $t4, $t5
exit:
```