İhsan Doğramacı Bilkent University Department of Computer Engineering Computer Organization CS224

Design Report Lab 5 Section 5 Erkin Aydın 22002956

13 April 2022

Part b:

Compute-Use Hazard:

Reason: This happens when in the current instruction it is tried to read a value where the previous instruction is not finished and will update the value, in other words, where the new value is not written back. This hazard would result on reading the wrong value.

Type: *Data Hazard*

Pipeline Stages Affected: 1) *Execute* and 2) *WriteBack*

Load-Use Hazard:

Reason: This happens when we load a value from the memory to a register and when the load operation is not finished, we read the very same register that will be loaded with a new value afterwards. This hazard would result on reading the wrong value.

Type: *Data Hazard*

Pipeline Stages Affected: 1) *Execute* and 2) *Memory*

Load-Store Hazard:

Reason: This happens when a value is loaded from the memory to a register and it is tried to store a value to the memory from the same register. For example, it can occur where a "*sw*" instruction comes right after a "*lw*" instruction, and the same register is being used, meaning, rt fields of these instructions are the same.

Type: *Data Hazard*

Pipeline Stages Affected: 1) *Memory*

Branch Hazard:

Reason: This happens when the branch decision is not taken where the next instruction is fetched.

Type: *Control Hazard*

Pipeline Stages Affected: 3 instructions, at maximum(consider the possibility that we are at the end of the instruction memory), will be fetched unnecessarily if the branch is taken.

Part c:

Compute-Use Hazard:

Forwarding is the solution. The new data can be *forwarded* to the execute stage of the next instruction. *Stalling* is also a possible solution, but it will be ineffective compared to *forwarding*.

Load-Use Hazard:

Stalling is the solution. We should wait until the new value of the register is written.

Load-Store Hazard:

Stalling is the solution. It is basically the same idea used in *load-use hazard*.

Branch Hazard:

There are two solutions: *Early Branch Decision* or *Stalling*. If the branch decision is made early, then the flushing the fetched instructions are necessary. If stalling is chosen, then the pipeline should be *stalled* for 3 cycles.

Part d: This part is done exactly as it was shown in the course book.

Solving Data Hazards:

Forwarding:

```
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
```

A Little Note: ForwardBE is exactly the same, except that it checks rt instead of rs.

Stalling&Flushing

lwstall = ((rsD = = rtE) OR (rtD = = rtE)) AND MemtoRegE StallF =
StallD = FlushE = lwstall

Solving Control Hazards:

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

branchstall = BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD)
OR
BranchD AND RegWriteM AND (WriteRegM == rsD OR WriteRegM == rtD)

StallF = StallD = FlushE = lwstall OR branchstall

Part e:

Since *sracc* will use all three registers' values and update one of them, this can result in following hazards:

1)If the previous instruction was to update any of the *rs*, *rt* or *rd* register, either by a load operation or an R-type instruction, we would compute the new value of the *rd* register wrong. If the previous operation is a load operation, then this is a *load-use hazard*, the solution is *stalling*. It the previous instructions result is ready at its *Execute* stage, then this is a *compute*-use hazard, and can be solved by *forwarding.*

2)If we try to store the value of the *rd* register where it is not updated yet, we will store the previous value of the register to the memory. Solution of this hazard is *stalling*.

Test Programs: Each code block, with consecutive instructions, should be tested separately.

Testing *rt*, *rs* and *rd* fields and their hazards:

rt:

```
la $t3, x #assume a data exists in the memory.
addi $t0, $t0, 4
addi $t1, $t1, 16
addi $t2, $t2, 2
sracc $t0, $t1, $t2
lw $t2, 0($t3)
sracc $t0, $t1, $t2
```

rs:

```
la $t3, x #assume a data exists in the memory.
addi $t0, $t0, 4
addi $t2, $t2, 2
addi $t1, $t1, 16
sracc $t0, $t1, $t2
lw $t1, 0($t3)
sracc $t0, $t1, $t2
```

rd:

```
la $t3, x #assume a data exists in the memory.
addi $t2, $t2, 2
addi $t1, $t1, 16
addi $t0, $t0, 4
sracc $t0, $t1, $t2
sw $t0, 0($t3)
```

Testing without hazards:

```
addi $s0, $zero, 4
addi $s1, $zero, 64
addi $s2, $zero, 128
addi $s3, $zero, 64
or $t0, $s0, $s1
and $t1, $s1, $s2
add $t2, $s0, $s3
slt $s4, $s0, $s1
sracc $t3, $s3, $s0
add $s5, $s2, $s3
sub $s6, $s0, $s1
lw $t3, 0x0c($s2)
sw $t2, 0x0c($s3)
```