Bilkent University

Computer Engineering

CS224 – Computer Organization

Design Report

Lab # 5

Section # 4

Mannan Abdul

21801066

11/05/2020

Part b)

List of all possible hazards:

1. Compute-use: This is a data hazard. This hazard affects decode stage of the current instruction. It occurs when an instruction attempts to use the value of a register that has not been updated with the previous instruction's computational result. E.g. when the previous instruction is add.
2. Load-use: This is also a data hazard. This hazard affects the decode stage of the current instruction. It occurs when an instruction attempts to use the value of a register that has not been loaded with the previous instruction's value. E.g. when the previous instruction is lw.
3. Load-store: This is also a data hazard which affects the decode stage of the current instruction. It occurs when an instruction tries to store a value that hasn't been loaded into the target register yet by the previous instruction. E.g. when the previous instruction is lw.
4. Compute-store: This too is a data hazard which affects the decode stage of the current instruction. It occurs when an instruction tries to store a value that has been computed but not written into the appropriate register by the previous instruction. E.g. when the previous instruction is sub.
5. Compute-branch: This is a control hazard, this occurs when the current instruction has to make a branch decision but will make the wrong one because the correct value though computed, hasn't been updated to the register yet. E.g. when the previous instruction is add.
6. Load-branch: This is also a control hazard, it occurs when the current instruction has to make a branch decision but will again make the wrong one because the correct value has not been loaded into the register needed. E.g. when the previous instruction is lw.

*Branch: This is a control hazard that occurs when the pipeline doesn't know what instruction to fetch next because the branch decision is made after the next instruction is fetched but we have remedied this by modifying the pipeline to make this decision in the decode stage.

**for store hazards, decode stage is affected because they need the value stored in the registers to compute the offset and determine where data needs to be stored.

Part c)

Solutions for Hazards:

1. Compute-use & compute-store: Because we have already computed the required value for the current instruction in the previous instructions' execute stage, we can forward

the value from the memory or the writeback stage of the previous instruction to the execute stage in the current instruction. This is done by looping back the value computed by the ALU in the memory stage back to the execute stage and into a mux whose control signals are determined by the hazard unit which will allow this value to continue forward when needed.

2. Load-use & load-store: because the value needed is not determined by the execute stage of the previous instruction but read at the end of it, we need to stall to get the appropriate value into the place we need. We stall the next instructions and their previous stages too and wait till the data is available in the writeback stage and then forward it to the execute stage of the current instruction. When we stall a stage, we stall all previous stages too so that no subsequent instructions are lost. The pipeline register immediately after the stalled stage is flushed so that no bogus information is forwarded. The stalling is done by adding enable inputs to the pc register and the fetch to decode pipeline register and a clear input to the decode to execute register so that it can be flushed.
3. Compute-branch & load branch: if we have the result of an ALU instruction in the memory stage, we can forward it to the equality comparator we use in the decode stage to make the branch decision. This is done by 2 new muxs that are controlled by the hazard unit. If we the result of an ALU instruction is in the execute stage or the result of a lw instruction is in the memory stage, we then need to stall the pipeline at the decode stage until the result is ready so that the correct branch decision is made. After stalling, we will need to again flush the decode to execute register while asserting the enable inputs of the pc register and the fetch to decode register so that no data is lost.

Part d)

Logic equations for the Hazard Unit:

if ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)

    ForwardAE = 10

else if ((rsE != 0) AND (rsW== WriteRegW) AND RegWriteW)

    ForwardAE = 01

else ForwardAE = 00

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

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

StallF = StallD = FlushE = lwstall OR branchstall

Part e)

Test Program 1 (No hazards):

addi $t1, $zero, 5

addi $t2, $zero, 6

addi $t3, $zero, 7

addi $t4, $zero, 8

sw $t1, 0($zero)

sw $t2, 8($zero)

sw $t3, 16($zero)

sw $t4, 24($zero)

Machine Code:

0x20090005

0x200a0006

0x200b0007

0x200c0008

0xac090000

0xac0a0008

0xac0b0010

0xac0c0018

Test Program 2 (Compute use):

```
addi $t1, $zero, 1
addi $t2, $zero, 2
add $t3, $t1, t2
addi $t4, $zero, 1
sub $t3, $t3, $t4
sw $t1, 0($zero)
sw $t2, 8($zero)
sw $t3, 16($zero)
sw $t4, 24($zero)
```

Machine Code:

```
0x20090001
0x200a0002
0x012a5820
0x200c0001
0x016c5822
0xac090000
0xac0a0008
0xac0b0010
0xac0c0018
```

Test Program 3 (load use):

```
addi $t1, $zero, 9
sw $t1, 0($zero)
lw $t2, 0($zero)
add $t3, $t1, $t2
```

sw $t2, 8($zero)

sw $t3, 16($zero)

Machine code:

0x20090009

0xac090000

0x8c0a0000

0x012a5820

0xac0a0008

0xac0b0010

Test Program 4 (load-branch):

addi $t1, $zero, 7

addi $t2, $zero, 8

sw $t1, 0($zero)

addi $t2, $t2, -1

lw $t3, 0($zero)

beq $t3, $t2, 1

addi $t2, $t2, -1

sw $t2, 8($zero)

Machine Code:

0x20090007

0x200a0008

0xac090000

0x214affff

0x8c0b0000

0x116a0001

0x214affff

0xac0a0008