Bilkent University
Computer Engineering Department
Computer Organization
CS 224

Design Report
Lab 5
Section 1

Arda Önal
21903350

12 April 2021

b)

Compute-use hazard:
Data hazard type
Execute, Memory or WriteBack stages are affected.

Load-use hazard:
Data hazard type
Fetch, Decode, Execute, WriteBack stages are affected.

Load-store hazard:
Data hazard type
Fetch, Decode, Execute WriteBack stages are affected.

Branch hazard:
Control hazard type
Fetch and Decode stages are affected.

c)

Compute-use hazard: It is caused if there is data dependency in either the next instruction or in two instructions later. If we are writing a result that we have computed to a register then using that register in next instructions, this hazard occurs. Specifically, If the "rd" value of our current instruction is the same as the "rs" or "rt" values of the next instruction or the instruction after that, it is said that there is data dependency. This hazard is solved by data forwarding whenever there is data dependency. Data that is either in Memory stage or WriteBack stage is forwarded to Execute stage to avoid this hazard. If the data dependency is in the next instruction, data that is in the Memory stage is forwarded to the Execute stage. If the data dependency is 2 instructions later, data that is in the WriteBack stage is forwarded into the Execute stage. If data dependency is 3 instructions later, there won't be a hazard because the computed result will already be written in the register file.

Load-use hazard: It is caused if there is data dependency in either the next instruction or in two instructions later. If we are writing data with the load word instruction to a register then using that register in next instructions, this hazard occurs. Data forwarding on a load-use hazard is not possible therefore, stalling is required to solve this hazard. If the "rs" value in the Decode stage equals the "rt" value in the Execute stage or the "rt" value in the Decode stage equals the "rt" value in the Execute stage and the MemtoRegE signal is 1, we have to stall the processor. To stall the processor, program count is stopped. We stall the Fetch stage, Decode stage and we have to flush the execute stage. By flushing the Execute stage, and stalling Fetch and Decode stages, the instruction flushed will simply be repeated in the next clock cycle, but this time with correct data. Data is also forwarded if necessary.

Load-store hazard: same as Load-use hazard

Branch hazard: This hazard occurs because branch is not determined until the 4th stage of the pipeline. In this implementation, early-branch hardware is implemented, which checks if we branch or not in the Decode stage. There is a subtractor which checks if "rs" and "rt" of the register file are equal and sends the result to the Fetch stage to update the pc if they are equal. Furthermore, if there is data dependency, we have to forward the data in the Memory stage to the Decode stage in order to avoid another hazard. The processor predicts that branch will not be taken for most of the time, therefore it starts to execute the next instructions. If the branch is taken, the processor flushes the instructions that shouldn't be taken to avoid any hazard. This mechanism saves clock cycles most of the time and makes the processor more efficient.

d)

Data Forwarding Logic:

ForwardAE: (inputs are: rsE, WriteRegM, RegWriteM, WriteRegW, RegWriteW)

```
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
```

ForwardBE: (inputs are: rtE, WriteRegM, RegWriteM, WriteRegW, RegWriteW)

```
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
```

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

Stalling Logic: (inputs are: rsD**,** rtD**,** rtE, MemtoRegE, BranchD, RegWriteE, WriteRegE, MemtoRegM, WriteRegM. outputs are: StallF, StallD, FlushE)

StallF = (((rsD==rtE) OR (rtD==rtE)) AND MemtoRegE) OR (BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD) OR BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD))

StallD = ((rsD==rtE) OR (rtD==rtE)) AND MemtoRegE OR (BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD) OR BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD))

FlushE = ((rsD==rtE) OR (rtD==rtE)) AND MemtoRegE OR (BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD) OR BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD))

e)

A test program with no hazards:

add $t0, $t1, $t2	0x012a4020
sub $t3, $t4, $t5	0x018d5822
sw $t5, 0($t6)	0xadcd0000
addi $t7, $t7, -1	0x21effff

A test program that has compute-use hazard:

add $t0, $t1, $t2	0x012a4020
addi $t0, $0, -1	0x2008ffff
sll $t0, $t0, 2	0x00084080

A test program that has load-use hazard:

addi $t1, $0, 8	0x20090008
lw $t0, 0($t1)	0x8d280000
addi $t2, $t0, -1	0x210affff

A test program that has load-store hazard:

addi $t1, $0, 8	0x20090008
lw $t0, 0($t1)	0x8d280000
sw $t1, 0($t0)	0xad090000

A test program that has branch hazard:

addi $t1, $0, 8	0x20090008
addi $t0, $0, 4	0x20080004
beq $t0, $t1, label1	0x11090002
addi $t1, $t1, 8	0x21290008
addi $t0, $t0, 4	0x21080004
label1: addi $t0, $t0, 24	0x21080018