BILKENT UNIVERSITY
COMPUTER SCIECE
DESIGN REPORT
LAB05

Zülal Nur Hıdıroğlu
Sec-06
21903125

b) All hazards in pipeline processor

In pipeline processor there are main two types of hazards that can be extended to subtitles according to the hazard. The first type is data hazards which an instruction tries to read a register that has not yet been written back by previous instruction. The second type is control hazard occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place. Data hazard has three different subtitles while control hazard has one in this lab report.

Data Hazard

1. Compute-use dependency

The reason of this hazard is an instruction cannot use the register because it is not written back by the previous instruction yet. For example;

add $t0, $t1, $t2

sub $t3, $t0, $t5

In this example $t0 is causing dependency because while first instruction tries to write value of it at fifth cycle which is write back stage of first instruction, second instruction needs its value at third cycle which is decoding state for second instruction.

Load-Use dependency

It is caused when an instruction tries to read the value of a register while its value is not set yet by lw instruction. For example;

lw $t0, 0($t1)

add $t2, $t0, $t1

In the given example, $t0 creates dependency because its value is not loaded when add instruction tries to reach $t0.

2. Load-store dependency

It is similar to load-use dependency. In this case the value is stored by sw instruction that is not set yet by lw instruction. For example;

lw $t0, 0($t1)

sw $t0, 0($t2)

In the given example, sw instruction tries to read the value of $t0 at decode stage which is third cycle of pipeline. However, lw instruction sets its value at the end of forth cycle which is possible to reach the value at fifth cycle.

Control Hazard

1. Branch Hazard

The problem in branch, the decision is taken at the fourth stage which is mem stage of the pipeline. In this case next instructions until mem stage will be processed. For example;

beq $t0, $t1, done

sub $t5, $t4, $t3

done : addi $t0, $t0,1

In the given example, since the target decided at mem stage, until first instruction comes to the mem stage, the processor will apply pc+4, not the target. Therefore, even the condition is satisfied, processor does not apply target after instruction immediately.

c) Solutions for Hazards

1. Compute-use dependency

Since there is no way to go back in time, there are three solutions to solve this problem. First one is using nop for two cycles to write the value of $t0 in the first half of the cycle the instruction, to read the value in the second half. The second solution for this problem is stalling. It is similar to nop which disables the pipeline that contents of it does not change. These two solutions are not efficient because there will be waste of two cycles in the case when either nop or stalling used. The efficient way of solving this problem is data forwarding. In this given

example, it is possible to reach the value of $t0 after execute stage of first instruction. Therefore, it can be used for the second instruction.

2. Load-use dependency

To solve this problem, unlike compute-use, data forwarding cannot be used because the value of $t0 is set after mem stage of lw instruction therefore add instruction at execute stage cannot reach $t0's value. Alternatively, nop and stalling can be used to solve this problem. Waiting for one clock cycle or stalling for one clock cycle will solve the hazard.

3. Load-store dependency

Since, value is set after mem stage, data forwarding is not a solution for this problem. Again, nop and stalling will solve this problem. But stalling is more efficient then using nop.

4. Brach hazard

There are two types of solution for branch hazard. First one is software solution. In this solution data forwarding is not possible because the next instruction is decided at mem stage. As second software solution nop can be used but it is not efficient. The hardware solution is using stalling. However, in the case which stalling used, flushing for three instruction has to occur because until beq instruction comes to mem stage, three instructions will occur. If branch is taken, these three instructions have to be flushed.

There is second hardware solution for this problem. In this solution and gate for branch is carried to decode stage which compares the data immediately after register file. If branch is taken, only one cycle should be flushed.

d)

Logic equation for forwarding

```
if((rsE!=0) AND (rsE == WriteRegM) AND RegWriteM) then
   ForwardAE = 10
 else if((rsE!=0)AND (rsE == WriteRegW) AND RegWriteM) then
   ForwardAE =01
```

```
 if((rtE!=0)AND (rtE == WriteRegM) AND RegWriteM) then
   ForwardBE =10
 else if((rtE!=0) AND (rtE == WriteRegW) && RegWriteM) then
   ForwardBE = 01
else
        ForwardBE = 00
 end
```

Logic equation for forwarding in branch hazard

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

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

Logic equation for lw stall

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

StallF = StallD = FlushE = lwStall

Logic Equation for branch stall

branchStall = (BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD))

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

StallF = StallD = FlushE = lwStall OR branchStall

e) Test Program with no hazard

addi $v0, $zero, 5	0x20020005;
addi $v1, $zero, 12	0x2003000c;
addi $a0, $zero, 10	0x2004000a;
addi $a3, $zero,20	0x20070014;
sw $a3, 68($v1)	0xac670044;
and $a1, $v0, $v1	0x00432824;
or $a1, $a3, $a0	0x00e42825;
addi $a3, $a3, 1	0x20e70001;
lw $a0, 0($v1)	0x8c640000;
slt $v0, $a1, $a1	0x00a5102a;

Compute-use hazar

addi $a3, $v1, -9	0x 2067fff7;
or $a0, $a3, $v0	0x 00e22025;
and $s1, $a3, $a0	0x 00e48824;

Load-use and load-store dependency

addi $a0, $zero, 4	0x 20040004;
addi $a1, $zero,5	0x 20050005;
sw $a0, 0($a1)	0x aca40000;
lw $a2, 0($a0)	0x 8c860000;
add $a3, $a2, $a0	0x 00c43820;

Branch hazard

addi $a0, $zero, 4	0x 20040004;
addi $a1, $zero, 5	0x 20050005;
beq $a0, $zero, 1	0x 1080ffec;
addi $a1, $a1,1	0x 20a50001;
done :	