Bilkent University

Computer Engineering Dept.

CS224 – Computer Organization

“Design Report”

Lab No.5

Section No.3

Nurettin Onur VURAL

Bilkent ID: 219023330

Date: 7 April, 2021

b) The list of all hazards that can occur in this pipeline. For each hazard,

give its type (data or control), its specific name ("compute use" "load use", "load store",

"branch" etc.), the pipeline stages that are affected.

Possible Hazards in Pipeline

1) Data Hazards

- **"Compute Use" Hazard:** This hazard may occur due to updating a value of a register. When the data of the register is not updated (written back) while the next instruction is dependant on it, procesor may fail to fetch the correct/new value in this next dependant instruction(s) in Decode stage. Accordingly, as the register value is read in this stage, the former value will be obtained and thus the operation will be incorrect. This will follow with Execute and Writeback stages being affected to eventually as they get this wrong computation coming from the Decode stage. This hazard can happen in R type instructions that follow each other when one updates a register and the following wants to fetch its data.

- **"Load Use" Hazard:** The reason behind this hazard is when a lw instruction happens, **it takes** up until the end of Memory stage which may result in fetching a wrong data value on the following instruction(s) if they are operating on the data of the register used in lw. In this respect this hazard having latency of two clock cycles may affect Execute and Memory stage.

- **"Load Store" Hazard:** Similar to load use, when a lw instruction is followed by a sw being dependant to the value that is fetched from memory and being written to a register in lw, the data to be stored in memory turn out to be incorrect as this value is obtained at the end of Memory stage of lw but the sw needs to have it beforehand. Eventually the Memory stage recieves an incorrect data value to be written to the data memory.

2) Contol Hazards

- **"Branch" Hazard:** In this hazard, the pipelined processor does not know what instruction to fetch next since the decision of taking the branch or not has not been made when the next instruction is fetched. To make this decision the value obtained by comparing two registers are used which is at Memory stage. Therefore up to this stage three instructions are already fetched. In this respect the branch hazard does not directly affect the data contents (in Memory Stage etc.) but may cause calling wrong instructions.

c) For each hazard, give the solution (forwarding, stalling, flushing, combination of these), and explanation of what, when, how

Possible Solutions for Pipeline Hazards

- **"Compute Use" Hazard Solution(s):** In order to make sure that the dependant instructions make their operations with correct data values, one possible approach can be stalling before the time the dependant instruction makes the operation in Execute stage until we are certain that the operand can receive the correct value. Since this increases the number of cycles overall, a more efficient appoach comes out to be as forwarding the data from the Memory or Writeback stage of the previous instruction (that updates the data used by the dependant instruction) just before the proper result is computed in Execute stage of the dependant

instruction. Hazard Unit accomplishes this by sending the necessary signals to the inputs of ALU in Execute stage so that it chooses from original RF, M stage or WB stage data.

- **“Load Use” Hazard Solution(s):** Forwarding does not work in this hazard type since the data is read at the end of Memory stage, not in the Execution stage like it happens compute use hazards, which disables the processor to send this result to the Execution stage of the dependant instruction. Lw instruction has two-cycle latency which means a dependant instruction cannot use the data until two clock cycles pass. In this respect stalling is required so that the processor waits for these two clock cycles to pass for the dependant instruction and the followings and than the processor send the results to their proper place, in other words makes the result available before the Execution stage of the dependant istructions. Stalling is a remarkable process which disables the pipeline register in order to make sure that the data values do not change and the instruction behaves as a nop at this moment. In order to not lose any instruction, stalling process stalls all the previous stages. The processor accomplishes all these by adding enable and reset inputs to the corresponidng registers and deciding whether the pipeline registers must hold their old values or not according to the Hazard Control Unit.

- **“Load Store” Hazard Solution(s):** As in Load Use Hazards, we need to stall the dependant instruction and the followings until it is certain that they receive the data coming from the end of Memory stage of the previous instruction. The data than can be used by the ID stage and pass into the following stages.

- **“Branch” Hazard Solution(s):** One approach is to stall the pipeline until the branch decision is made but as this increases the total number of cycles, it is not quite effective. A more efficient way is trying to predict the decision beforehand. As the decision may be wrong, in this case the following fetched instrucitons must be cancelled. This branch prediciton penalty is dealt by flushing the instrucitons, placing the necessary signals to the set clr/reset for the corresponding pipleine registers. Besides it must be noted that making the branch decision earlier at decode stage by the help of a comparator is much faster and it reduces the number of instructions to be flushed in case of wrong decision.

d) Give the logic equations for each signal output by the hazard unit, as a function of the input signals that come to the hazard unit. This hazard unit should handle all the data and control hazards that can occur in your pipeline (listed in b) so that your pipelined processor computes correctly

// to solve hazards by forwarding

// Source A (rs)

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

ForwardAE = 10

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

ForwardAE = 01

else

ForwardAE = 00

// Source B (rt)

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

ForwardBE = 10

else if ((rtE != 0) AND (rtE == WriteRegW) AND RegWriteW)

ForwardBE = 01

else

ForwardBE = 00

// decode stage forwarding

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

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

// to solve hazards by stalling-flushing

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

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

OR

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

// upgraded stall for branch

StallF = StallD = FlushE = lwstall OR branchstall

e) **Write small test programs, in MIPS assembly, that will show whether the pipelined processor is working or not. Each of your test programs should be designed to catch problems, if there are any, in the execution of MIPS instruc tions in your pipelined m a-chine. Write:**

- **No hazards**

Assembly Form	**Machine Instruction Form**
addi $s0, $zero, 0x0002	8'h00: instr = 32'h20100002;
addi $s1, $zero, 0x0007	8'h04: instr = 32'h20110007;
addi $s3, $zero, 0x0004	8'h08: instr = 32'h20130004;
addi $s4, $zero, 0x0005	8'h0c: instr = 32'h20140005;
add $s0, $s1, $s0	8'h10: instr = 32'h02308020;
add $s1, $zero, $zero	8'h14: instr = 32'h00008820;
sub $s2, $s3, $s4	8'h18: instr = 32'h02749022;
lw $t0, 0x0008($zero)	8'h1c: instr = 32'h8c080008;
and $t1, $t2, $t3	8'h20: instr = 32'h014b4824;
sw $t4, 0x0008($zero)	8'h24: instr = 32'hac0c0008;
addi $s2, $s3, 5	8'h28: instr = 32'h22720005;

- **Compute use hazard**

Assembly Form	**Machine Instruction Form**
addi $s1, $zero, 2	8'h00: instr = 32'h20110002;
addi $s2, $zero, 7	8'h04: instr = 32'h20120007;
addi $s4, $zero, 6	8'h08: instr = 32'h20140006;
add $s0, $s1, $s2	8'h0c: instr = 32'h02328020;
add $s3, $s0, $s4	8'h10: instr = 32'h02149820;
sub $s4, $s3, $zero	8'h14: instr = 32'h0260a022;
addi $s5, $s3, 6	8'h18: instr = 32'h22750006;

- **Load use hazard**

Assembly Form	**Machine Instruction Form**
addi $s1, $zero, 1	8'h00: instr = 32'h20110001;
add $t0, $zero, $zero	8'h04: instr = 32'h00004020;
add $t1, $zero, $zero	8'h08: instr = 32'h00004820;
add $t2, $zero, $zero	8'h0c: instr = 32'h00005020;
lw $s0, 0x0004($s1)	8'h10: instr = 32'h8e300004;
add $s2, $s3, $s0	8'h14: instr = 32'h02709020;
sub $s3, $s0, $s4	8'h18: instr = 32'h02149822;

- **Load store hazard**

Assembly Form	Machine Instruction Form
addi $s1, $zero, 1	8'h00: instr = 32'h20110001;
add $t0, $zero, $zero	8'h04: instr = 32'h00004020;
add $t1, $zero, $zero	8'h08: instr = 32'h00004820;
add $t2, $zero, $zero	8'h0c: instr = 32'h00005020;
lw $s0, 0x0004($s1)	8'h10: instr = 32'h8e300004;
sw $s0, 0x0008($t0)	8'h14: instr = 32'had100008;

- **Branch hazard**

Assembly Form	Machine Instruction Form
addi $t0, $zero, 0x0007	8'h00: instr = 32'h20080007;
beq $zero, $zero, 0x0006	8'h04: instr = 32'h10000004;
sub $t1, $zero, $zero	8'h08: instr = 32'h00004822;
or $t2, $zero, $zero	8'h0c: instr = 32'h00005025;
add $t3, $t0, $zero	8'h10: instr = 32'h01005820;
or $t4, $t0, $zero	8'h14: instr = 32'h01006025;
add $t4, $zero, $zero	8'h18: instr = 32'h00006020;
add $t3, $zero, $zero	8'h1c: instr = 32'h00005820;
add $t2, $zero, $zero	8'h20: instr = 32'h00005020;
add $t1, $zero, $zero	8'h24: instr = 32'h00004820;