Bilkent University
CS Department
CS224

# Design Report

Lab 5

Section 5

Name: Esad İsmail Tök
Student ID: 21801679

Lab Day: 9 April 2021

# Part 1-b

1)
Type: Data Hazard
Specific Name: Compute-use
Affected Pipeline Stages: The Decode stage of a fetched instruction is affected when the compute use hazard occurs. The wrong data value in the source registers are taken in the Decode Stage. Therefore the computations done in Execute and Writeback Stages would also be wrong.

2)
Type: Data Hazard
Specific Name: Load-use
Affected Pipeline Stages: Load-use hazard causes a two-cycle latency and depending on whether the memory is written or not the Execute and the Memory Stages would be effected.

3)
Type: Data Hazard
Specific Name: Load-store
Affected Pipeline Stages: In the storing instruction, the Memory Stage is effected because wrong data from the register file would be stored in the data memory.

4)
Type: Control Hazard
Specific Name: Branch
Affected Pipeline Stages: the branch decision is made in the Memory Stage. And until this decision is made 3 instructions are processed and those instructions are currently in Fetch, Decode, and Execute stages and if the branch is taken the instructions in those stages are wrong instructions.

# Part 1-c

1)
Type: Data Hazard
Specific Name: Compute-use
Solution: Forwarding
What to do to Solve it: The result of the previous instruction which the next instruction is dependent on can be accessed before it is written to the RF. Therefore The we can take the result of the from the beginning of the Memory stage or from the beginning of the Writeback stage and select the source data of the operand with multiplexer.
When this Hazard Occurs: This type of hazard occurs when an instruction has a source operand which is upgraded by a previous instruction but not yet written into the register file. Namely when the source operand of an instruction depends on the same destination register of the previous instruction.
How this Hazard Occurs: The resulting data of an R-type instruction is written to the register file in the Writeback stage. However there may be maximum of 2 instructions after that instruction which has the destination operand of the former R-type instruction as their source operand (The third instruction after that instruction is not a problem because of the early write late read principle.) Therefore the letter instructions would have the wrong datas as their source operands in this hazard.

2)
Type: Data Hazard
Specific Name: Load-use
Solution: Combination of the Stalling, Flushing, and Forwarding
What to do to Solve it: To solve this hazard, it is not enough to forward data alone. We also need to stall the instructions before the Execute stage that are the instructions in the Fetch and Decode stages. At the same time we need to flush the instruction in the execute stage, namely when the posedge of the clock the values in the pipeline register of the execute stage will be all 0's. After those steps the data will be available to forward to the earlier stages and the forwarding will be done.
When this Hazard Occurs: This Hazard occurs when we have an instruction that is one ofter a load word instruction and using the source register that is same as the destination register of the load word instruction.
How this Hazard Occurs: The resulting data of the lw instruction is not accessible until the end of the Memory stage. So regular forwarding does not prevent this hazard if the instruction is one ofter the lw instruction. Therefore when the source operands that same as the destination register of the lw instruction will have the wrong (not upgraded) data value inside. Therefore the instruction after the lw instruction will do a wrong computation.

# Part 1-c

3)
Type: Data Hazard
Specific Name: Load-store
Solution: Combination of Stalling and Flushing
What to do to Solve it: I order to solve this problem we need to stall the pipeline stages that are before the write of the RF is a solution for this hazard. And simultaneously we need to flush the current pipeline stage to prevent a wrong execution and to introduce a bubble in the processor which cause us to lost a cycle.
When this Hazard Occurs: This hazard occurs when there are 2 consecutive instructions in which the first instruction is lw and the latter instruction is a sw instruction and when they use the same rt register as their operands.
How this Hazard Occurs: The resulting data of the lw instruction is not accessible until the end of the Memory stage. So if the one after instruction of the lw instruction is a sw instruction and if this instruction uses the same rt register as the lw instruction, then the value of this register is not ready when the sw instruction is trying to use it. In such a case the Load-store hazard occurs in the processor because the operand of the sw would have the wrong data when execution.

4)
Type: Control Hazard
Specific Name: Branch
Solution: Combination of Stalling, Flushing, and Forwarding
What to do to Solve it: I order to solve this hazard, we can move the branch decision from the Memory stage to the Decode stage by moving the AND operation of the branch decision and the BTA calculation to the Decode stage. Performing early branch resolution we will presume that the next instruction will have address of PC + 4, and if the branch is taken we will flush the pipeline register of the decode stage to prevent the wrong execution of the previously and wrongly taken instruction and then fetch the instruction in the BTA. By solving this problem we will have a possible data hazard. To solve those hazards as well we need to forward the resulting data to the multiplexers of the branch decision logic when needed. We also need to stall the processor if the instruction that is one before the branch instruction has the destination register that is same as the source operand of the branch instruction. Or if we have lw instruction that is 2 instructions before the branch instruction has the same destination register with the source operand of the branch instruction.
When this Hazard Occurs: This hazard occurs when we have a branch instruction in which we do not know what instruction is to be fetched as the next. And also in the early branch prediction, additional hazards may occur when the previous instructions have the destination registers same as the source registers of the branch instruction. And also when we have lw instruction 2 before the branch instruction.
How this Hazard Occurs: When we pick the next instruction address as the PC + 4, there is a possibility that the branch may been taken. And in this case we would have fetched 3 wrong instructions instead of the one in the branch target address in this case the processor would do wrong computations for those instructions and the state of the processor might be wrongly changed because of this hazard.

# Part 1-d

**The forwarding logic for Compute-use Hazards:**

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

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

**The stalling and flushing logic for Load-use Hazards:**

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

StallF = StallD = FlushE = lwstall

**The forwarding logic for Branch Hazards that is occurred because of early branch resolution:**

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

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

# Part 1-d

**The stalling and flushing logic for Branch Hazards that is occurred because of early branch resolution:**

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

# Part 1-e

**Test program with no hazards:**

| **Assembly:** | **Address and Machine Code:** |
|---|---|
| addi $t0, $zero, 12 | 8'h00: 32'h2008000C |
| addi $t1, $zero, 8 | 8'h04: 32'h20090008 |
| addi $t2, $zero, 5 | 8'h08: 32'h200A0005 |
| addi $t3, $zero, 4 | 8'h0c: 32'h200B0004 |
| add $t2, $t0, $t1 | 8'h10: 32'h01095020 |
| or $t2, $t0, $t1 | 8'h14: 32'h01095025 |
| sub $t0, $t1, $t1 | 8'h18: 32'h01294022 |
| addi $s0, $zero, 10 | 8'h1c: 32'h2010000A |
| beq $t3, $zero, 1 | 8'h20: 32'h11600001 |
| lw $t1, 0($t2) | 8'h24: 32'h8D490000 |

**Test program with compute-use hazards:**

| **Assembly:** | **Address and Machine Code:** |
|---|---|
| addi $t0, $zero, 12 | 8'h00: 32'h2008000C |
| addi $t1, $zero, 8 | 8'h04: 32'h20090008 |
| add $t2, $t0, $t1 | 8'h08: 32'h01095020 |

**Test program with load-use hazards:**

| **Assembly:** | **Address and Machine Code:** |
|---|---|
| addi $t0, $zero, 12 | 8'h00: 32'h2008000C |
| addi $t1, $zero, 8 | 8'h04: 32'h20090008 |
| addi $t2, $zero, 5 | 8'h08: 32'h200A0005 |
| addi $t3, $zero, 4 | 8'h0c: 32'h200B0004 |
| sw $t0, 0($t1) | 8'h10: 32'hAD280000 |
| lw $s0, 0($t1) | 8'h14: 32'h8D300000 |
| add $t4, $s0, $t2 | 8'h18: 32'h020A6020 |
| add $t3, $t2, $s0 | 8'h1c: 32'h01505820 |

**Test program with branch hazards:**

| **Assembly:** | **Address and Machine Code:** |
|---|---|
| addi $t0, $zero, 12 | 8'h00: 32'h2008000C |
| addi $t1, $zero, 8 | 8'h04: 32'h20090008 |
| add $s0, $zero, $t1 | 8'h08: 32'h00098020 |
| beq $t1, $s0, 1 | 8'h0c: 32'h11300001 |
| add $t3, $t0, $t1 | 8'h10: 32'h01095820 |
| add $t4, $t0, $zero | 8'h14: 32'h01006020 |