Lab 6 Prelab:

Prelab Questions

1. A data hazard exists in an inorder processor when an instruction reads from a register that is being written to by a previous instruction. This type of hazard is commonly referred to as a Read-AfterWrite (RAW) hazard. RAW data hazards must be handled properly in order to maintain correct code execution. One way to deal with data hazards is to stall the pipeline in the ID stage when the hazard is detected. For the data path shown in Figure 2, how many cycles would the pipeline need to be stalled when a RAW exists between two back to back R-type instructions? What about the case where an R-type instruction immediately follows a load? Likewise, can a branch instruction cause a RAW? What about a store instruction? Explain your answers for each of these!

2. Simply stalling the pipeline to resolve data hazards increases the processors CPI (clock cycles per instruction), thereby reducing its overall efficiency. What improvement might you make to the design to reduce the number of stalls? For back to back R-type instructions, can we eliminate stalls all together? Why or why not? What about RAW data hazards involving load instructions?

3. Control hazards exist when the processor executes a jump or branch (i.e. changes the path of execution). Explain the reason for a control hazard involving a branch and provide a method for resolving this hazard. Do the same for jump.

4. Structural Hazards exist when two or more instructions need to use a resource in the data path at the same time. Explain how having separate memories for data and instructions avoids a structural hazard in the MIPS pipeline. Do any unresolved structural hazards exist?

5. What modification to Figure 2 would we have to make to handle the aforementioned hazards? Please provide some detail.

Answers:

1. RAW Hazards or Read After Write Hazards are when a Register Read occurs before the Write back to the register has been rendered. Let us assume that we make use of Stalls to resolve this Hazard
   Let us Assume the following 2 instructions are executed:
   ADDI R3, R2, #1
   ADD R1, R2, R3
   Here there is a RAW hazard as R3 needs to be written back before it can be used in Instruction 2.
   Suppose we fetch Instruction 1 in Cycle 1, without a Hazard we would be able to fetch Instruction 2 in Cycle 2 and Decode in Cycle 3.
   However due to the Hazard the Decode in Cycle 3 is stalled for 2 Cyclew so that the Write Back is completed(Here we are assuming that the write back happens in the first half of the cycle and the read register happens in the second half which means that write of I1 and Read of I2 can occur concurrently.

Instruction	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**
ADD1 R3, R2, #1	F	D	E	M	W	X	X	X
ADD R1, R2, R3		F	D	STALL(D)	STALL(D)	E	M	W

The case where R type instruction follows a Load is very similar to the above case. We would still have to wait for the Write back of the load instruction (C5 over here too) to get completed so that we can decode the second instruction. Assuming no extra delays associated with the Load type instruction the depiction of the Pipeline would be similar to the above table (except instruction 1 is a Load type instruction).

Instruction	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**
LW R2,100(R0)	F	D	E	M	W	X	X	X
ADD R1, R2, R3		F	D	STALL(D)	STALL(D)	E	M	W

Both a Branch Type and a Store Type instruction can cause RAW Hazards. Since for Both Branch Type and Store Type we don't write back into the GPRs these should be the instructions reading data from a register.
For eg.
ADD R1, R2, R3
BEQ R1,R0, Loop

This is a Branch condition where the Branch reads from R1 before R1 is being written into. This may potentially cause a Hazard.

ADD R1, R2, R3
SW 0(R1), R3.
This is an example of RAW Hazard involving store instructions where the Result is stored in the wrong memory location. (R1 is not yet updated)

2. One way to resolve Data Hazards would be to enable Forwarding. In forwarding we have Registers after each stage to store the intermediate value. For eg. between the F and D stage we would have a F/D Register. These register values can be forwarded to subsequent instructions so that they can check if there is a hazard and make use of the updated values.(Previously decoded values are ignored)
   Now lets check the case where 2 back to back R type instructions are executed:
   ADDI R3, R2, #1
   ADD R1, R2, R3

Instruction	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
ADDI R3, R2, #1	F	D	E	M	W	X	X
ADD R1, R2, R3		F	D	E	M	W	X

   As we can see after C3 when I2 is decoded and ready for execution we already have the ALU output ready in the EX/MEM register and that value is forwarded to I2 and used to perform Execution of the second instruction.

   Now lets take the case of a Load Instruction

Instruction	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
LW R1,100(R0)	F	D	E	M	W	X	X
ADD R1, R2, R3		F	D	STALL	E	M	W

   As in the Load instruction we have to wait for the MEM stage of the Load instruction to complete to start forwarding, we CANNOT eliminate all stalls with the help of forwarding. The MEM/WB register value can be forwarded to I2 after C4, and this reduces the number of stalls from 2 to 1.

   Out of Order execution: Apart from forwarding we can also make use of Out of Order execution. This can be both static(compiler) and dynamic(hardware) based. In out of order execution we fill the existing stalls with independent instructions so that the overall CPI reduces. This technique is often coupled with Loop Unrolling where a Loop is unrolled numerous times by the compiler to reduce loop maintenance instructions and achieve better scheduling.

3. Control Hazards are types of Hazards where the Instruction order deviates from the expected sequential order. Our fundamental expectation is that after each instruction is executed the PC is updated with PC + 4 and this is the next instruction to be executed. However, for Branch and Jump type instructions this is not the case.
   For eg.
   Loop: ADDI R1, R1, #-1
   BEQ R1, Loop
   ADDI R1,R1,#1

   In this case if branch is taken that is R1 is equal to 0 then instead of ADDI R1,R1,#1 we would be required to execute ADDI R1,R1,#-1. But the pipeline would have already fetched I3. Hence after the execute stage of the Branch when it is resolved this instruction would need to be flushed.

A Jump is similar however there is 1 less instruction flushed in Jump since a Jump is an unconditional branch. Hence once we decode the instruction to be a Jump we can fetch the correct instruction after the next cycle.

To solve this issue we can make use of **Branch Prediction**. Branch Prediction is a technique where we predict if a Branch will be taken or not based on the historical behavior of the current branch or nearby branches. It so happens that Branch behavior is consistent each time a branch is executed and almost always a Branch is executed many times. This way we can avoid flushing fetched instructions and stalling the pipeline. The same Branch predictor tables like the Branch History Table and the Branch target Buffer can be used to predict both Branches AND Jumps.

4. Structural Hazards occur when the same functional unit is required for more than one instruction. When we have separate memories for Data and Instructions we avoid Structural hazards because this enables us to perform Data fetch for one instruction and Instruction Fetch for another instruction concurrently. Say in the Pipeline the MEM part of a Load Instruction and the Fetch Part of a R type instruction occur simultaneously. If the Data and Instruction memory is the same we risk encountering a Structural Hazard. However, with separate memories we avoid this Hazard. Even though we avoid this hazard there exist some Structural Hazards which need to be kept in mind. One such example is if we Write and Read to the same memory location in 2 different instructions. Consequent instructions like these need to be scheduled in some other way as such scheduling may result in unwanted stalls in the pipeline.

5. We can implement all of the above techniques using the following additions to the MIPS Block Diagram:

   a) Forwarding: We require 4 Registers namely F/D, D/EX, EX/MEM, MEM/WB so that we can store the intermediate results. We would also need a MUX at each stage to choose from the Forwarded value or the Instruction Value. The select signal for this MUX would also need to be designed to classify if the current instruction is making use of any registers from the previous instructions.
   b) Branch Prediction: For the most basic type of Branch Prediction we need atleast two register tables. We would need the BHT or the Branch History Table which would be indexed using the Address. The BHT tells two things i) If the Instruction is a Branch ii) If the Instruction is a Branch is it Taken. If both these conditions satisfy then the corresponding BHT entry would be 1. If we encounter a 1 in the BHT we then index into the BTB or the Branch Target Buffer to check the new PC value and then for the subsequent fetch (at this stage Branch is not yet resolved) we would use the new PC value and fetch. We would also need to have write backs to these 2 tables to keep the BHT and BTB updated of any changes in Behavior.
   c) Structural Hazards: To resolve the structural hazards we can just add more functional units. For eg. we can have a separate ALU to perform DIV and MULT so that smaller ALU operations don't get stalled. Similarly we can have more read and write buses to the Memory and Register module.