1. **How many stages is the datapath you’ve drawn? (i.e. How many cycles does it take to execute 1 instruction?)**

As shown in the figure, we have divided a total of three stages in the data path. Also since we have arranged the pc reg side by side with the IMEM and BIOS, it takes three clock cycles to execute an instruction.

1. **How do you handle ALU → ALU hazards?**

**addi x1, x2, 100**

**addi x2, x1, 100**

To solve ALU → ALU hazards, we added data forwarding to the pipeline. since in ALU → ALU hazards, the previous instruction's output of the previous instruction may be used as the input of the next instruction in ALU → ALU hazards, it would be a waste of clock cycles if forwarding is not used. This hazard lies in the fact that the output of alu needs to wait for the end of the third phase before it can be stored back in the memory for the next cycle. Therefore, we decided to connect the output of ALU to the input of the branch comparator after passing it through the stage3 registers again and filtering it through mux.

1. **How do you handle ALU → MEM hazards?**

**addi x1, x2, 100**

**sw x1, 0(x3)**

To solve ALU → MEM hazards, the same data forwarding used to solve ALU → ALU hazards is used.This is due to the fact that we found that in ALU → MEM hazards, the output of the previous instruction conflicted with the input of the next instruction still lies in the fact that the output of alu needs to wait for the end of the third phase before it can be stored back in the memory for the next cycle, which is consistent with the conflict in the previous problem. So we can solve this problem by using the same forwarding.

1. **How do you handle MEM → ALU hazards?**

**lw x1, 0(x3)**

**addi x1, x1, 100**

To solve MEM → ALU hazards, we have to use data forwarding as well. in this case, the output of MEN may be used in the ALU computation of the next instruction hence the need to wait for an extra clock cycle. So we solve this problem by connecting the output of mem directly to the input of the branch comparator, using the mux selection signal.

1. **How do you handle MEM → MEM hazards?**

**lw x1, 0(x2)**

**sw x1, 4(x2)**

**Also consider:**

**lw x1, 0(x2)**

**sw x3, 0(x1)**

To solve the MEM → MEM hazards, we also use data forwarding, in which case the output of the MEM may be used as input to the next MEM and therefore needs to wait for an additional clock cycle. This conflict is actually not fundamentally different from the one in the previous problem, so we can use the previous data forwarding line to solve this problem.

1. **Do you need special handling for 2 cycle apart hazards?**

**addi x1, x2, 100**

**nop**

**addi x1, x1, 100**

To solve the 2 cycle apart hazards, we need to ensure that the PC remains unchanged during no operation and enter an instruction that has no effect. So we can add a +0 option to the PC control +4 logic and select +4 or +0 via a mux, and we can also add a mux after the IMEM output to select whether to enter the current instruction from the decoder or the default null instruction (e.g., addi x0, x0, 0).

1. **How do you handle branch control hazards? (What is the mispredict latency, what prediction scheme are you using, are you just injecting NOPs until the branch is resolved, what about data hazards in the branch?)**

For handling branch control hazards, the result of the branch comparison is determined in the execute stage. If the branch is not taken, the program proceeds as usual. However, if the branch is taken, we stall for one cycle and flush the instruction in the decode stage, updating the PC counter to the branch's target address. Therefore, our prediction scheme assumes that branches are always not taken. The mispredict latency is an additional cycle.

Regarding data hazards, if the branch instruction depends on a register value that has not yet been written back to the RegFile, we address this by using data forwarding from the ALU and memory stages. These forwarded values are fed into the branch comparator to prevent data hazards.\

1. **How do you handle jump control hazards? Consider jal and jalr separately. What optimizations can be made to special-case handle jal?**

In jump control hazards, the jal and jalr instructions will jump to another place in the program that is not PC + 4, so we want to know where they jump before the whole instruction ends to save time.

For the jal instruction, the new pc position is equal to the current pc + imm, and imm can be learned at the same time we recognize the jal instruction. So we build an adder that adds the number imm to the current pc and feeds it into pcsel, so that we get the new pc in advance.

For the jalr instruction, the new pc position is equal to the Reg[rs1] + imm. Although we can still add a line connect the output of the decoder and the adder to obtain the new pc, this data path may be too long to significantly increase the delay. So we believe there is no need to do so.

1. **What is the most likely critical path in your design?**

The crtical path should be at the EX stage. this is because the EX satge first contains a bunch of muxes and an alu, which will introduce huge latency to the line. Also considering data forwarding, EX stage receives data forwarding from MEM & WB stage, which is a time consuming process and further increases its total latency.

1. **Where do the UART modules, instruction, and cycle counters go? How are you going to drive uart\_tx\_data\_in\_valid and uart\_rx\_data\_out\_ready (give logic expressions)?**

In MEM&WB stage, along with BIOS, DMEM, and IMEM.

uart\_tx\_data\_in\_valid = (write\_enable && address == 32'h80000008)

uart\_rx\_data\_out\_ready = (read\_enable && address == 32'h80000004)

**11. What is the role of the CSR register? Where does it go?**

1. Role of the CSR Register: The CSR register indicates the program’s status in simulation. A value of 1 means success, while values above 1 indicate a test failure.

2. Placement: It can be placed in the Execute stage, ensuring every instruction can access it.

**12. When do we read from BIOS for instructions? When do we read from IMem for instructions? How do we switch from BIOS address space to IMem address space? In which case can we write to IMem, and why do we need to write to IMem? How do we know if a memory instruction is intended for DMem or any IO device?**

(1) Read from BIOS when [31:28] is 4'b0100 and address type is PC or Data, typically during startup.

(2) Read from IMem when [31:28] is 4'b0001 with PC address type, usually when there are instructions in IMem after BIOS completes.

(3) Switch to IMem by setting PC[30] to 1, which directs the multiplexer to select IMem.

(4) Write to IMem when [31:28] is 4'b001x with Data type, usually for loading a program via UART.

(5) Use the high bits [31:28] to distinguish between DMem, IMem, BIOS, or IO device targets.