**Pipeline**

Q1 A 5-stage pipelined RISCV datapath are illustrated in Fig. 1. Now we have the time information for each component tabulated in Table 1. What is the clock time and frequency of a pipelined CPU (ignore the branch comp and imm.)? (3 points)



Fig. 1 5-stage pipelined RSICV datapath

Table 1

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Element | Register clk-to-q | Register Setup | MUX | ALU | Mem Read | Mem Write | RegFile Read | RegFile Setup |
| Parameter | tclk-to-q | tsetup | tmux | tALU | tMEMread | tMEMwrite | tRFread | tRFsetup |
| Delay(ps) | 30 | 20 | 25 | 250 | 400 | 500 | 200 | 50 |

**the Memory stage takes the longest time:**

READ: tclk-to-q + tmemread + tmux + tsetup = 30+400+25+20 = 475 ps

WRITE: tclk-to-q + tmemwrite = 30 + 500 = 530 ps

fclk,pipe = 1/tclk,pipe <= 1/ (530 ps) = 1.887 GHz

**Hazards**

Data hazards occur due to data dependencies among instructions. Forwarding can solve many data hazards.

Q2. Spot the data dependencies in the code below and figure out how forwarding can resolve

data hazards. (5 points)



The REG step for instructions 2 and 3 depend on data in the registers only available after the WB step of instruction 1. We can forward the ALU output of the first instruction to the EX stages of future instructions

Q3. In general, under what conditions will an EX stage need to take in forwarded inputs from

previous instructions? Where should those inputs come from in regards to the current cycle?

Assume you have the signals ALUout(n), rt(n), rs(n), regWrite(n), and regDst(n), where n is 0

for the signal of the current instruction being executed by the EX stage, -1 for the previous, etc.

For example,

Forward ALUout(-1) if (rs2(0) == regDst(-1) || rs1(0) == regDst(-1)) && regWrite(-1)

Please give other conditions. (6 points)

Forward ALUout(-2) if (rs2(0) == regDst(-2) || rs1(0) == regDst(-2)) && regWrite(-2)

Forward ALUout(-3) if (rs2(0) == regDst(-3) || rs1(0) == regDst(-3)) && regWrite(-3)

Q4. Spot the data dependencies in the code below and figure out why forwarding cannot

resolve this hazard. What can we do to solve this data hazard? (6 points)



The add instruction needs the value of t0 in the beginning of C3, but it is ready at the end of C3.



We can insert a nop into the load delay slot as shown above, or even better we can reorder instructions and fill up the load delay slot with an instruction to avoid performance loss.

Q5. Given the RISC-V code below and a pipelined CPU with no forwarding, how many hazards

would there be? What types are each hazard? Consider all possible hazards from all pairs of

instructions. How many stalls would there need to be in order to fix the data hazard(s)? What about the

control hazard(s)? (6 points)



**hazards from 1-2, 1-3, 2-3, 4-5**

**4-5 is a control hazard, all others are data hazard.**

**Assuming concurrent reads and writes to registers are possible, two stalls for instruction 2 are needed for register t0 between 1 and 2, two stalls are needed for the register s2 between 2 and 3.**

**Either flush the pipeline or use branch prediction to deal with the control hazard.**

1. Under what circumstances can we execute instruction 4 in the code above before executing any others? (5 points)

Instruction 4 has a RAW hazard with instruction 3 because it depends on the value of register x6 that is being loaded by instruction 3. Therefore, it cannot execute first. It does not have any dependencies on any other instruction, since stores do not change architectural state until they are committed, so it can execute as soon as instruction 3 completes.

1. Now let’s assume that we execute instruction 5 before all other instructions, but instruction 5 causes an exception (e.g., page fault). We want to provide precise exceptions in this processor. What happens with instructions 1, 2, 3, 4, and 6 before execution switches to the OS handler? What should happen if instructions, 1, 2, 3, or 4 also raise an exception? (5 points)

To provide precise exceptions, we have to execute and commit all instructions prior to instruction 5 before switching to the OS handler. Instruction 6 must be killed (thus not commit) because it’s after 5. If instructions 1, 2, 3, or 4 also raise an exception, the earliest instruction to have an exception should take precedent, and all later instructions should be flushed from the pipeline.

1. How can we always be able to execute loads and stores out of order before their addresses are known? What is the downside and how is it handled? Specifically, assume that we executed instruction 5 before instruction 4, but then realized that |x6 – x3| < 4. (5 points)

We can speculatively assume that addresses of all loads and stores are non- overlapping and issue them before knowing their addresses. Once addresses become known, if we realize that we shouldn’t have reordered some loads and stores, we have to terminate the ones we shouldn’t have executed as well as any further instructions that depend on them. In the example, we have to terminate instruction 5 as well as 6 once we figure out that |x6-x3| < 4. Once instruction 4 completes, we then re-execute instructions 5 and 6.