1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 By pipelining the CPU datapath, each single instruction will execute faster (latency is reduced), resulting in a speed-up in performance.

1.2 A pipelined CPU datapath results in instructions being executed with higher throughput (than the single-cycle CPU).

1.3 Through adding additional hardware, we can implement two 'read' ports as well as a 'write' port to the RegFile (where registers can be accessed). This solves the hazard of two instructions reading and writing to the same register simultaneously.

1.4 All data hazards can be resolved with forwarding.

1.5 As stalling reduces performance significantly, we generally prefer other solutions to fixing pipeline hazards, even at the cost of complexity or hardware. In a modern-day pipelined CPU, are there still use-cases for stalling to resolve potential hazards? If so, describe a program that would.
2 Pipelining Registers

In order to pipeline, we separate the datapath into 5 discrete stages, each completing a different function and accessing different resources on the way to executing an entire instruction.

In the IF stage, we use the Program Counter to access our instruction as it is stored in IMEM. Then, we separate the distinct parts we need from the instruction bits in the ID stage and generate our immediate, the register values from the RegFile, and other control signals. Afterwards, using these values and signals, we complete the necessary ALU operations in the EX stage. Next, anything we do in regards with DMEM (not to be confused with RegFile or IMEM) is done in the MEM stage, before we hit the WB stage, where we write the computed value that we want back into the return register in the RegFile.

These 5 stages, divided by registers as shown in the figure, allow the datapath to provide a pipeline for multiple instructions to operate at the same time, each accessing different resources. A pipelined datapath is provided for you on the last page. Use it to answer the following questions.

2.1 What is the purpose of the new registers?

2.2 Looking at the way PC is passed through the datapath, there are two places where +4 is added to the PC, once in the IF and MEM stage. Why do we add +4 to the PC again in the memory stage?

2.3 Why do we need to save the instruction in a register multiple times?

3 Performance Analysis

<table>
<thead>
<tr>
<th>Component</th>
<th>Delay (ps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register clk-to-q</td>
<td>30</td>
</tr>
<tr>
<td>Register setup</td>
<td>20</td>
</tr>
<tr>
<td>Register hold</td>
<td>10</td>
</tr>
<tr>
<td>Mux</td>
<td>25</td>
</tr>
<tr>
<td>Branch comp.</td>
<td>75</td>
</tr>
<tr>
<td>ALU</td>
<td>200</td>
</tr>
<tr>
<td>Imm. Gen.</td>
<td>15</td>
</tr>
<tr>
<td>Memory read</td>
<td>250</td>
</tr>
<tr>
<td>Memory write</td>
<td>200</td>
</tr>
<tr>
<td>RegFile read</td>
<td>100</td>
</tr>
<tr>
<td>RegFile setup</td>
<td>20</td>
</tr>
</tbody>
</table>

Given above are sample delays for each of the datapath components and register timings. In the questions below, use these in conjunction with the pipelined datapath on the last page to answer them.
3.1 What would be the fastest possible clock time for a single cycle datapath? Recall from last week’s discussion that one instruction which exercises the critical path is lw.
(HINT: \( t_{\text{clk-cycle}} \geq t_{\text{clk-to-q}} + t_{\text{longest-combinational-path}} + t_{\text{setup}} \))

3.2 What is the fastest possible clock time for a pipelined datapath?

3.3 What is the speedup from the single cycle datapath to the pipelined datapath? Why is the speedup less than 5×?

4 Hazards
One of the costs of pipelining is that it introduces pipeline hazards. Hazards, generally, are issues with something in the CPU’s instruction pipeline that could cause the next instruction to execute incorrectly.

The 5-stage pipelined CPU introduces three types: structural hazards (hardware not sufficient), data hazards (using wrong values in computation), and control hazards (executing the wrong instruction).

Structural Hazards
Structural hazards occur when more than one instruction needs to use the same datapath resource at the same time. In the standard 5-stage pipeline, there aren’t structural hazards, unless there are active changes to the pipeline. The structural hazards that could exist are prevented by RV32I’s hardware requirements.

There are two main causes of structural hazards:
Pipelining

• **Register File:** The register file is accessed both during ID, when it is read to decode the instruction, and the corresponding register values; and during WB, when it is written to in the rd register. If the RegFile only had one port, then it wouldn’t work since we have one instruction being decoded and another writing back.

  – We resolve this by having separate read and write ports. However, this only works if the read/written registers are different.

  – To account for reads and writes to the same register, processors usually write to the register during the first half of the clock cycle, and read from it during in the second half. This is an implementation of the idea of **double pumping**, which is when data is transferred along data buses at double the rate, by utilising both the rising and falling clock edges in a clock cycle.

• **Main Memory:** Main memory is accessed for both instructions and data. If memory could only support one read/write at a time, then instruction A going through IF and attempting to fetch an instruction from memory cannot happen at the same time as instruction B attempting to read (or write) to data portions of memory.

  – Having a separate instruction memory (abbreviated IMEM) and data memory (abbreviated DMEM) solves this hazard.

Something to remember about structural hazards is that they can always be resolved by adding more hardware.

Data Hazards
Data hazards are caused by data dependencies between instructions. In CS 61C, where we always assume that instructions go through the processor in order, we see data hazards when an instruction reads a register before a previous instruction has finished writing to that register.

There are two types of data hazards:

• **EX-ID:** this hazard exists because the output from the execute stage is not written back to the RegFile until the writeback stage, yet can be requested by the subsequent instruction in the decode stage.

• **MEM-ID:** this hazard exists because the output from the memory access stage is not written back to the RegFile until the writeback stage, but can be requested from the decode stage, just as in EX-ID.

Solving Data Hazards
For all questions, assume **no branch prediction or double-pumping** (i.e. write-then-read in one cycle for RegFile).

Forwarding
Most data hazards can be resolved by forwarding, which is when the result of the EX or MEM stage is sent to the EX stage for a following instruction to use.
Look for data hazards in the code below, and figure out how forwarding could be used to solve them.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. addi t0, a0, -1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. and s2, t0, a0</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. sltiu a0, t0, 5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Imagine you are a hardware designer working on a CPU’s forwarding control logic. How many instructions after the addi instruction could be affected by data hazards created by this addi instruction?

Stalls

Look for data hazards in the code below. One of them cannot be solved with forwarding—why? What can we do to solve this hazard?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>C8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. addi s0, s0, 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. addi t0, t0, 4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. lw t1, 0(t0)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. add t2, t1, x0</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Say you are the compiler and can re-order instructions to minimize data hazards while guaranteeing the same output. How can you fix the code above?
Detecting Data Hazards

Say we have the $rs_1$, $rs_2$, $RegWEn$, and $rd$ signals for two instructions (instruction $n$ and instruction $n + 1$) and we wish to determine if a data hazard exists across the instructions. We can simply check to see if the $rd$ for instruction $n$ matches either $rs_1$ or $rs_2$ of instruction $n + 1$, indicating that such a hazard exists (why does this make sense?).

We could then use our hazard detection to determine which forwarding paths/number of stalls (if any) are necessary to take to ensure proper instruction execution. In pseudo-code, part of this could look something like the following:

```plaintext
if (rs1(n + 1) == rd(n) && RegWen(n) == 1) {
    set ASel for (n + 1) to forward ALU output from n
}
if (rs2(n + 1) == rd(n) && RegWen(n) == 1) {
    set BSel for (n + 1) to forward ALU output from n
}
```

Control Hazards

Control hazards are caused by jump and branch instructions, because for all jumps and some branches, the next PC is not PC + 4, but the result of the ALU available after the EX stage. We could stall the pipeline for control hazards, but this decreases performance.

Besides stalling, what can we do to resolve control hazards?