HW4: 4.8, 4.9, 4.10, 4.14, 4.15    deadline: 12/10

4.8

In this exercise, we examine how pipelining affects the clock cycle time of the processor. Problems in this exercise assume that individual stages of the datapath have the following latencies:

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>250ps</td>
<td>350ps</td>
<td>150ps</td>
<td>300ps</td>
<td>200ps</td>
</tr>
</tbody>
</table>

Also, assume that instructions executed by the processor are broken down as follows:

<table>
<thead>
<tr>
<th></th>
<th>alu</th>
<th>beq</th>
<th>lw</th>
<th>sw</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>45%</td>
<td>20%</td>
<td>20%</td>
<td>15%</td>
</tr>
</tbody>
</table>

4.8.1 [5] <$4.5$> What is the clock cycle time in a pipelined and non-pipelined processor?

4.8.2 [10] <$4.5$> What is the total latency of an LW instruction in a pipelined and non-pipelined processor?

4.8.3 [10] <$4.5$> If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor?

4.8.4 [10] <$4.5$> Assuming there are no stalls or hazards, what is the utilization of the data memory?

4.8.5 [10] <$4.5$> Assuming there are no stalls or hazards, what is the utilization of the write-register port of the “Registers” unit?

4.8.6 [30] <$4.5$> Instead of a single-cycle organization, we can use a multi-cycle organization where each instruction takes multiple cycles but one instruction finishes before another is fetched. In this organization, an instruction only goes through stages it actually needs (e.g., ST only takes 4 cycles because it does not need the WB stage). Compare clock cycle times and execution times with single-cycle, multi-cycle, and pipelined organization.
4.9

4.9 In this exercise, we examine how data dependences affect execution in the basic 5-stage pipeline described in Section 4.5. Problems in this exercise refer to the following sequence of instructions:

or r1, r2, r3
or r2, r1, r4
or r1, r1, r2

Also, assume the following cycle times for each of the options related to forwarding:

<table>
<thead>
<tr>
<th>Without Forwarding</th>
<th>With Full Forwarding</th>
<th>With ALU-ALU Forwarding Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>250ps</td>
<td>300ps</td>
<td>290ps</td>
</tr>
</tbody>
</table>


4.9.2 [10] <§4.5> Assume there is no forwarding in this pipelined processor. Indicate hazards and add \texttt{nop} instructions to eliminate them.

4.9.3 [10] <§4.5> Assume there is full forwarding. Indicate hazards and add \texttt{NOP} instructions to eliminate them.

4.9.4 [10] <§4.5> What is the total execution time of this instruction sequence without forwarding and with full forwarding? What is the speedup achieved by adding full forwarding to a pipeline that had no forwarding?

4.9.5 [10] <§4.5> Add \texttt{nop} instructions to this code to eliminate hazards if there is ALU-ALU forwarding only (no forwarding from the MEM to the EX stage).

4.9.6 [10] <§4.5> What is the total execution time of this instruction sequence with only ALU-ALU forwarding? What is the speedup over a no-forwarding pipeline?

4.10
4.10 In this exercise, we examine how resource hazards, control hazards, and Instruction Set Architecture (ISA) design can affect pipelined execution. Problems in this exercise refer to the following fragment of MIPS code:

```
sw r16,12(r6)
lw r16,8(r6)
beq r5,r4,Label # Assume r5!=r4
add r5,r1,r4
slt r5,r15,r4
```

Assume that individual pipeline stages have the following latencies:

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>200ps</td>
<td>120ps</td>
<td>150ps</td>
<td>190ps</td>
<td>100ps</td>
</tr>
</tbody>
</table>

4.10.1 [10] <§4.5> For this problem, assume that all branches are perfectly predicted (this eliminates all control hazards) and that no delay slots are used. If we only have one memory (for both instructions and data), there is a structural hazard every time we need to fetch an instruction in the same cycle in which another instruction accesses data. To guarantee forward progress, this hazard must always be resolved in favor of the instruction that accesses data. What is the total execution time of this instruction sequence in the 5-stage pipeline that only has one memory? We have seen that data hazards can be eliminated by adding nops to the code. Can you do the same with this structural hazard? Why?

4.10.2 [20] <§4.5> For this problem, assume that all branches are perfectly predicted (this eliminates all control hazards) and that no delay slots are used. If we change load/store instructions to use a register (without an offset) as the address, these instructions no longer need to use the ALU. As a result, MEM and EX stages can be overlapped and the pipeline has only 4 stages. Change this code to accommodate this changed ISA. Assuming this change does not affect clock cycle time, what speedup is achieved in this instruction sequence?

4.10.3 [10] <§4.5> Assuming stall-on-branch and no delay slots, what speedup is achieved on this code if branch outcomes are determined in the ID stage, relative to the execution where branch outcomes are determined in the EX stage?

4.10.4 [10] <§4.5> Given these pipeline stage latencies, repeat the speedup calculation from 4.10.2, but take into account the (possible) change in clock cycle time. When EX and MEM are done in a single stage, most of their work can be done in parallel. As a result, the resulting EX/MEM stage has a latency that is the larger of the original two, plus 20 ps needed for the work that could not be done in parallel.

4.10.5 [10] <§4.5> Given these pipeline stage latencies, repeat the speedup calculation from 4.10.3, taking into account the (possible) change in clock cycle time. Assume that the latency ID stage increases by 50% and the latency of the EX stage decreases by 10ps when branch outcome resolution is moved from EX to ID.
4.10.6  [10] <§4.5> Assuming stall-on-branch and no delay slots, what is the new
clock cycle time and execution time of this instruction sequence if beq address
computation is moved to the MEM stage? What is the speedup from this change?
Assume that the latency of the EX stage is reduced by 20 ps and the latency of the
MEM stage is unchanged when branch outcome resolution is moved from EX to
MEM.

4.14

4.14  This exercise is intended to help you understand the relationship between
delay slots, control hazards, and branch execution in a pipelined processor. In
this exercise, we assume that the following MIPS code is executed on a pipelined
processor with a 5-stage pipeline, full forwarding, and a predict-taken branch
predictor:

```
lw r2,0(r1)
lbl1: beq r2,r0,labell2 # not taken once, then taken
       lw r3,0(r2)
       beq r3,r0,labell1 # taken
       add r1,r3,r1
labell2: sw r1,0(r2)
```

4.14.1  [10] <§4.8> Draw the pipeline execution diagram for this code, assuming
there are no delay slots and that branches execute in the EX stage.

4.14.2  [10] <§4.8> Repeat 4.14.1, but assume that delay slots are used. In the
given code, the instruction that follows the branch is now the delay slot instruction
for that branch.

4.14.3  [20] <§4.8> One way to move the branch resolution one stage earlier is to
not need an ALU operation in conditional branches. The branch instructions would
be "bez rd,labell" and "bnez rd,label", and it would branch if the register has
and does not have a zero value, respectively. Change this code to use these branch
instructions instead of beq. You can assume that register R8 is available for you
to use as a temporary register, and that an seq (set if equal) R-type instruction can
be used.
Section 4.8 describes how the severity of control hazards can be reduced by moving branch execution into the ID stage. This approach involves a dedicated comparator in the ID stage, as shown in Figure 4.62. However, this approach potentially adds to the latency of the ID stage, and requires additional forwarding logic and hazard detection.

**4.14.4** [10] \(\text{§4.8}\) Using the first branch instruction in the given code as an example, describe the hazard detection logic needed to support branch execution in the ID stage as in Figure 4.62. Which type of hazard is this new logic supposed to detect?

**4.14.5** [10] \(\text{§4.8}\) For the given code, what is the speedup achieved by moving branch execution into the ID stage? Explain your answer. In your speedup calculation, assume that the additional comparison in the ID stage does not affect clock cycle time.

**4.14.6** [10] \(\text{§4.8}\) Using the first branch instruction in the given code as an example, describe the forwarding support that must be added to support branch execution in the ID stage. Compare the complexity of this new forwarding unit to the complexity of the existing forwarding unit in Figure 4.62.

### 4.15

**4.15** The importance of having a good branch predictor depends on how often conditional branches are executed. Together with branch predictor accuracy, this will determine how much time is spent stalling due to mispredicted branches. In this exercise, assume that the breakdown of dynamic instructions into various instruction categories is as follows:

<table>
<thead>
<tr>
<th>R-Type</th>
<th>BEQ</th>
<th>JMP</th>
<th>LW</th>
<th>SW</th>
</tr>
</thead>
<tbody>
<tr>
<td>40%</td>
<td>25%</td>
<td>5%</td>
<td>25%</td>
<td>5%</td>
</tr>
</tbody>
</table>

Also, assume the following branch predictor accuracies:

<table>
<thead>
<tr>
<th>Always-Taken</th>
<th>Always-Not-Taken</th>
<th>2-Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>45%</td>
<td>55%</td>
<td>85%</td>
</tr>
</tbody>
</table>

**4.15.1** [10] \(\text{§4.8}\) Stall cycles due to mispredicted branches increase the CPI. What is the extra CPI due to mispredicted branches with the always-taken predictor? Assume that branch outcomes are determined in the EX stage, that there are no data hazards, and that no delay slots are used.

**4.15.2** [10] \(\text{§4.8}\) Repeat 4.15.1 for the “always-not-taken” predictor.

**4.15.3** [10] \(\text{§4.8}\) Repeat 4.15.1 for the 2-bit predictor.

**4.15.4** [10] \(\text{§4.8}\) With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions in a way that replaces a branch instruction with an ALU instruction? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.
4.15.5  [10] <§4.8> With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions in a way that replaced each branch instruction with two ALU instructions? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

4.15.6  [10] <§4.8> Some branch instructions are much more predictable than others. If we know that 80% of all executed branch instructions are easy-to-predict loop-back branches that are always predicted correctly, what is the accuracy of the 2-bit predictor on the remaining 20% of the branch instructions?