# Homework 6 – Solutions

1. [Pipelining and Data Hazards]

**4.10.1** In the pipelined execution shown below, *** represents a stall when an instruction cannot be fetched because a load or store instruction is using the memory in that cycle. Cycles are represented from left to right, and for each instruction we show the pipeline stage it is in during that cycle:

| Instruction | Pipeline Stage | Cycles |
|---|---|---|
| SW R16,12(R6)<br>LW R16,8(R6)<br>BEQ R5,R4,Lbl<br>ADD R5,R1,R4<br>SLT R5,R15,R4 | IF ID EX MEM WB<br>IF ED EX MEM WB<br>IF ID EX MEM WB<br>*** *** IF ID EX MEM WB<br>IF ID EX MEM WB | 11 |

We can not add NOPs to the code to eliminate this hazard – NOPs need to be fetched just like any other instructions, so this hazard must be addressed with a hardware hazard detection unit in the processor.

**4.10.2** This change only saves one cycle in an entire execution without data hazards (such as the one given). This cycle is saved because the last instruction finishes one cycle earlier (one less stage to go through). If there were data hazards from loads to other instructions, the change would help eliminate some stall cycles.

| Instructions Executed | Cycles with 5 stages | Cycles with 4 stages | Speedup |
|---|---|---|---|
| 5 | 4 + 5 = 9 | 3 + 5 = 8 | 9/8 = 1.13 |

**4.10.3** Stall-on-branch delays the fetch of the next instruction until the branch is executed. When branches execute in the EXE stage, each branch causes two stall cycles. When branches execute in the ID stage, each branch only causes one stall cycle. Without branch stalls (e.g., with perfect branch prediction) there are no stalls, and the execution time is 4 plus the number of executed instructions. We have:

| Instructions Executed | Branches Executed | Cycles with branch in EXE | Cycles with branch in ID | Speedup |
|---|---|---|---|---|
| 5 | 1 | 4 + 5 + 1*2 = 11 | 4 + 5 + 1*1 = 10 | 11/10 = 1.10 |

**4.10.4** The number of cycles for the (normal) 5-stage and the (combined EX/MEM) 4-stage pipeline is already computed in 4.10.2. The clock cycle time is equal to the latency of the longest-latency stage. Combining EX and MEM stages affects clock time only if the combined EX/MEM stage becomes the longest-latency stage:

| Cycle time with 5 stages | Cycle time with 4 stages | Speedup |
|---|---|---|
| 200 ps (IF) | 210 ps (MEM + 20 ps) | (9*200)/(8*210) = 1.07 |

**4.10.5**

| New ID latency | New EX latency | New cycle time | Old cycle time | Speedup |
|---|---|---|---|---|
| 180 ps | 140 ps | 200 ps (IF) | 200 ps (IF) | (11*200)/(10*200) = 1.10 |

2. [Pipelining and Data Hazards]

**4.12.1** Dependences to the 1st next instruction result in 2 stall cycles, and the stall is also 2 cycles if the dependence is to both 1st and 2nd next instruction. Dependences to only the 2nd next instruction result in one stall cycle. We have:

| CPI | Stall Cycles |
|---|---|
| 1 + 0.35*2 + 0.15*1 = 1.85 | 46% (0.85/1.85) |

**4.12.2** With full forwarding, the only RAW data dependences that cause stalls are those from the MEM stage of one instruction to the 1st next instruction. Even this dependences causes only one stall cycle, so we have:

| CPI | Stall Cycles |
|---|---|
| 1 + 0.20 = 1.20 | 17% (0.20/1.20) |

3. [Pipelining and Data Hazards]

**4.13.1**

```
ADD R5,R2,R1
NOP
NOP
LW R3,4(R5)
LW R2,0(R2)
NOP
OR R3,R5,R3
NOP
NOP
SW R3,0(R5)
```

**4.13.2** We can move up an instruction by swapping its place with another instruction that has no dependences with it, so we can try to fill some NOP slots with such instructions. We can also use R7 to eliminate WAW or WAR dependences so we can have more instructions to move up.

| | |
|---|---|
| I1: ADD R5,R2,R1<br>I3: LW R2,0(R2)<br>NOP<br>I2: LW R3,4(R5)<br>NOP<br>NOP<br>I4: OR R3,R5,R3<br>NOP<br>NOP<br>I5: SW R3,0(R5) | Moved up to fill NOP slot<br><br>Had to add another NOP here, so there is no performance gain |

**4.13.3** With forwarding, the hazard detection unit is still needed because it must insert a one-cycle stall whenever the load supplies a value to the instruction that immediately follows that load. Without the hazard detection unit, the instruction that depends on the immediately preceding load gets the stale value the register had before the load instruction.

| Code executes correctly (for both loads, there is no RAW dependence between the load and the next instruction). |
|---|

**4.13.4** The outputs of the hazard detection unit are PCWrite, IF/IDWrite, and ID/EXZero (which controls the Mux after the output of the Control unit). Note that IF/IDWrite is always equal to PCWrite, and ED/ExZero is always the opposite of PCWrite. As a result, we will only show the value of PCWrite for each cycle. The outputs of the forwarding unit is ALUin1 and ALUin2, which control Muxes that select the first and second input of the ALU. The three possible values for ALUin1 or ALUin2 are 0 (no forwarding), 1 (forward ALU output from previous instruction), or 2 (forward data value for second-previous instruction). We have:

| Instruction sequence | First five cycles<br>1 2 3 4 5 | Signals |
|---|---|---|
| ADD R5,R2,R1<br>LW R3,4(R5)<br>LW R2,0(R2)<br>OR R3,R5,R3<br>SW R3,0(R5) | IF ID EX MEM WB<br>IF ID EX MEM<br>IF ID EX<br>IF ID<br>IF | 1: PCWrite=1, ALUin1=X, ALUin2=X<br>2: PCWrite=1, ALUin1=X, ALUin2=X<br>3: PCWrite=1, ALUin1=0, ALUin2=0<br>4: PCWrite=1, ALUin1=1, ALUin2=0<br>5: PCWrite=1, ALUin1=0, ALUin2=0 |

4. [Pipelining and Data Hazards]

**4.15.1** Each branch that is not correctly predicted by the always-taken predictor will cause 3 stall cycles, so we have:

| Extra CPI |
|---|
| 3*(1 – 0.45)*0.25 = 0.41 |

**4.15.2** Each branch that is not correctly predicted by the always-not-taken predictor will cause 3 stall cycles, so we have:

| Extra CPI |
|---|
| 3*(1 – 0.55)*0.25 = 0.34 |

**4.15.3** Each branch that is not correctly predicted by the 2-bit predictor will cause 3 stall cycles, so we have:

| Extra CPI |
|---|
| 3*(1 – 0.85)*0.25 = 0.113 |

5. [Pipelining and Data Hazards]

**4.16.1**

| Always Taken | Always not-taken |
|---|---|
| 3/5 = 60% | 2/5 = 40% |

**4.16.2**

| Outcomes | Predictor value at time of prediction | Correct or Incorrect | Accuracy |
|---|---|---|---|
| T, NT, T, T | 0,1,0,1 | I,C,I,I | 25% |

**4.16.3** The first few recurrences of this pattern do not have the same accuracy as the later ones because the predictor is still warming up. To determine the accuracy in the "steady state", we must work through the branch predictions until the predictor values start repeating (i.e., until the predictor has the same value at the start of the current and the next recurrence of the pattern).

| Outcomes | Predictor value at time of prediction | Correct or Incorrect (in steady state) | Accuracy in steady state |
|---|---|---|---|
| T, NT, T, T, NT | 1st occurrence: 0,1,0,1,2<br>2nd occurrence: 1,2,1,2,3<br>3rd occurrence: 2,3,2,3,3<br>4th occurrence: 2,3,2,3,3 | C,I,C,C,I | 60% |

6. [Exceptions]

**4.17.1**

| Instruction 1 | Instruction 2 |
|---|---|
| Invalid target address (EX) | Invalid data address (MEM) |

**4.17.2** The Mux that selects the next PC must have inputs added to it. Each input is a constant address of an exception handler. The exception detectors must be added to the appropriate pipeline stage and the outputs of these detectors must be used to control the pre-PC Mux, and also to convert to NOPs instructions that are already in the pipeline behind the exception-triggering instruction.

**4.17.3** Instructions are fetched normally until the exception is detected. When the exception is detected, all instructions that are in the pipeline after the first instruction must be converted to NOPs. As a result, the second instruction never completes and does not affect pipeline state. In the cycle that immediately follows the cycle in which the exception is detected, the processor will fetch the first instruction of the exception handler.