Selected Exercise for Chapter 4, Part 2				
4.8.1~3	4.9	4.10	4.12	4.16.1~3

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:

IF	ID	EX	MEM	WB
250ps	350ps	150ps	300ps	200ps

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

Alu	beq	lw	sw
45%	20%	20%	15%

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

4.8.1 Solution: Pipeline: 350 ps. Single-cycle: 250+ 350+150+300+200=1250ps

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

4.8.2 Solution: Pipeline: 350*5= 1750ps. Single-cycle: 1250ps

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?

4.8.3 Solution: Stage to split is ID because it has the largest latency. New clock cycle time is 300 ps

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:

Without Forwarding	With Full Forwarding	With ALU-ALU Forwarding Only
250ps	300ps	290ps

4.9.1 [10] <§4.5> Indicate dependences and their type.

4.9.1

Instruction sequence	Dependences
I1: OR R1,R2,R3 I2: OR R2,R1,R4 I3: OR R1,R1,R2	RAW on R1 from I1 to I2 and from I1 to I3 RAW on R2 from I2 to I3 WAR on R2 from I1 to I2 WAR on R1 from I2 to I3 WAW on R1 from I1 to I3

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

4.9.2 In the basic five-stage pipeline WAR and WAW dependences do not cause any hazards. Without forwarding, any RAW dependence between an instruction and the next two instructions (if register read happens in the second half of the clock cycle and the register write happens in the first half). The code that eliminates these hazards by inserting NOP instructions is:

Instruction sequence	
OR R1,R2,R3	
NOP	Delay I2 to avoid RAW hazard on R1 from I1
NOP	
OR R2,R1,R4	
NOP	Delay I3 to avoid RAW hazard on R2 from I2
NOP	
OR R1,R1,R2	

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

4.9.3 With full forwarding, an ALU instruction can forward a value to EX stage of the next instruction without a hazard. However, a load cannot forward to the EX stage of the next instruction (by can to the instruction after that). The code that eliminates these hazards by inserting NOP instructions is:

Instruction sequence	
OR R1,R2,R3	
OR R2,R1,R4	No RAW hazard on R1 from I1 (forwarded)
OR R1,R1,R2	No RAW hazard on R2 from I2 (forwarded)

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?

4.9.4 The total execution time is the clock cycle time times the number of cycles. Without any stalls, a three-instruction sequence executes in 7 cycles (5 to complete the first instruction, then one per instruction). The execution without forwarding must add a stall for every NOP we had in 4.9.2, and execution forwarding must add a stall cycle for every NOP we had in 4.9.3. Overall, we get:
No forwarding: (7 + 4)*250 ps = 2750 ps
With forwarding: 7*300 ps = 2100 ps. Speedup due to forwarding= 1.31

4.9.5 [10] <§4.5> Add nop instructions to this code to eliminate hazards if there is ALU-ALU forwarding only (no forwarding from the MEM to the EX stage).
4.9.5 With ALU-ALU-only forwarding, an ALU instruction can forward to the next instruction, but not to

the second-next instruction (because that would be forwarding from MEM to EX). A load cannot forward at all, because it determines the data value in MEM stage, when it is too late for ALU-ALU forwarding. We have:

Instruction sequence	
OR R1,R2,R3	
OR R2,R1,R4	ALU-ALU forwarding of R1 from I1
Nop	Because no MEM to EX forwarding
Nop	
OR R1,R1,R2	

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?

4.9.6 solution
No forwarding: (7 + 4)*250 ps = 2750 ps
With ALU-ALU forwarding only: (5+4)*290=2610ps
Speedup with ALU-ALU forwarding: 2750/2610=1.053

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:

IF	ID	EX	MEM	WB
200ps	120ps	150ps	190ps	100ps

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?

4.10.1 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. X represents this pipeline stage is not used. The Cycles are represented from left to right, and for each instruction we show the pipeline stage it is in during that cycle:

SW	IF	ID	EX	ME	X								
LW		IF	ID	EX	ME	WB							
BEQ			IF	ID	EX	X	WB						
ADD				***	***	IF	ID	EX	ME	WB			
SLT							IF	ID	EX	ME	WB		

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

We cannot add NOPs to the code to eliminate this hazard because NOPs need to be fetched from the memory just like any other instructions. Therefore, this hazard must be addressed with a hardware hazard detection unit in the processor.

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?

4.10.2 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 [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?

4.10.3 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

A more detailed diagram

Determined in EXE stage

SW	IF	ID	EX	ME	X								
LW		IF	ID	EX	ME	WB							
BEQ			IF	ID	EX	X	WB						
ADD				***	***	IF	ID	EX	ME	WB			
SLT							IF	ID	EX	ME	WB		

A more detailed diagram

Determined in ID stage

SW	IF	ID	EX	ME	X								
LW		IF	ID	EX	ME	WB							
BEQ			IF	ID	EX	X	WB						
ADD				***	IF	ID	EX	X	WB				
SLT						IF	ID	EX	ME	WB			

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 20ps needed for the work that could not be done in parallel.

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 [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 by10ps when branch outcome resolution is moved from EX to ID.

4.10.5

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

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.

4.10.6 The cycle time remains unchanged: a 20 ps reduction in EX latency has no effect on clock cycle time because EX is not the longest-latency stage. The change does affect execution time because it adds one additional stall cycle to each branch. Because the clock cycle time does not improve but the number of cycles increases, the speedup from this change will be below 1 (a slowdown). In 4.10.3, we already computed the number of cycles when branch is in EX stage. We have:

Cycles with branch in EX		Execution time (branch in EX)	Cycles with branch in MEM	Execution time (branch in MEM)	Speedup
a.	4+5+1*2=11	11*200 ps=2200 ps	4+5+1*3=12	12*200 ps=2400 ps	11/12=0.92

Branch is determined in MEM stage

SW	IF	ID	EX	ME	X								
LW		IF	ID	EX	ME	WB							
BEQ			IF	ID	EX	X	WB						
ADD				***	***	***	IF	ID	EX	ME	WB		
SLT								IF	ID	EX	ME	WB	

4.12 This exercise is intended to help you understand the cost/complexity/ performance trade-offs of forwarding in a pipelined processor. Problems in this exercise refer to pipelined datapaths from Figure 4.45. These problems assume that, of all the instructions executed in a processor, the following fraction of these instructions have a particular type of RAW data dependence. The type of RAW data dependence is identified by the stage that produces the result (EX or MEM) and the instruction that consumes the result (1st instruction that follows the one that produces the result, 2nd instruction that follows, or both). We assume that the register write is done in the first half of the clock cycle and that register reads are done in the second half of the cycle, so "EX to 3rd" and "MEM to 3rd" dependences are not counted because they cannot result in data hazards. Also, assume that the CPI of the processor is 1 if there are no data hazards.

EX to 1st Only	MEM to 1st Only	EX to 2nd Only	MEM to 2nd Only	EX to 1st and MEM to 2nd	Other RAW Dependences

5%	20%	5%	10%	10%	10%

Assume the following latencies for individual pipeline stages. For the EX stage, latencies are given separately for a processor without forwarding and for a processor with different kinds of forwarding.

IF	ID	EX(no FW)	EX(full FW)	EX (FW from EX/MEM only)	EX (FW from MEM/WB only)	MEM	WB
150ps	100ps	120ps	150ps	140ps	130ps	120ps	100ps

4.12.1 [10] <§4.7> If we use no forwarding, what fraction of cycles are we stalling due to 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 Cycle
1+0.35*2+0.15*1=1.85 (EX to 1, MEM to 1, EX to 1) *2 + (EX to 2, MEM to 2)*2	46%(0.85/1.85)

4.12.2 [5] <§4.7> If we use full forwarding (forward all results that can be forwarded), what fraction of cycles are we staling due to data hazards?

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 Cycle
1+0.20=1.20	17%(0.20/1.20)

4.12.3 [10] <§4.7> Let us assume that we cannot afford to have three-input Muxes that are needed for full forwarding. We have to decide if it is better to forward only from the EX/MEM pipeline register (next-cycle forwarding) or only from the MEM/WB pipeline register (two-cycle forwarding). Which of the two options results in fewer data stall cycles?

4.12.3

EX/MEM only	EX to 1st dependences : no stall (use forwarding) MEM to 1st dependences: two stalls (wait for data to be written back EX to 2nd dependences: one stall (wait for data to be written back) MEM to 2nd dependences: one stall (wait for data to be written back EX to 1st and MEM to 2nd : one stall
MEM/WB only	EX to 1st dependences: one stall (because we must wait for the instruction to complete the MEM stage to be able to forward to the next instruction) MEM to1st dependences: one-cycle stall EX to 2nd dependences: no stalls. MEM to 2nd dependences: no stall (use forwarding)

	EX to 1st and MEM to 2nd : one stall	

We compute stall cycles per instructions for each case as follows:

EX/MEM	MEM/WB	Fewer stall cycles with
0.2*2+0.05+0.1+0.1=0.65	0.05+0.2+0.1=0.35	MEM/WB

4.12.4 [10] <§4.7> For the given hazard probabilities and pipeline stage latencies, what is the speedup achieved by adding full forwarding to a pipeline that had no forwarding?

4.12.4. In 4.12.1 and 4.12.2, we have already computed the CPI without forwarding and with full forwarding. Now we compute time per instruction by taking into account the clock cycle time:

Without forwarding	With forwarding	Speedup
1.85*150 ps=277.5 ps	1.20*150 ps=180 ps	1.54

4.12.5 [10] <§4.7> What would be the additional speedup (relative to a processor with forwarding) if we added time-travel forwarding that eliminates all data hazards? Assume that the yet-to-be-invented time-travel circuitry adds 100 ps to the latency of the full-forwarding EX stage.

4.12.5 We already computed the time per instruction for full forwarding in 4.12.4. Now we compute time-per instruction with time-travel forwarding and the speedup over full forwarding:

With full forwarding	Time-travel forwarding	Speedup
1.20*150 ps=180 ps	1*250 ps=250 ps	0.72

4.12.6 [20] <§4.7> Repeat 4.12.3 but this time determine which of the two options results in shorter time per instruction.

4.12.6

EX/MEM	MEM/WB	Shorter time per instruction with
1.65*150 ps=247.5	1.35*150 ps = 202.5 ps	MEM/WB

Exercise 4.16

This exercise examines the accuracy of various branch predictors for the following repeating pattern (e.g., in a loop) of branch outcomes: T, NT, T, T, NT

4.16.1 [5] <4.8> What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?

4.16.1 Solution: always-taken 3/5=60%, always not-taken 2/5=40/%

4.16.2 [5] <4.8> What is the accuracy of the two-bit predictor for the first four branches in this pattern, assuming that the predictor starts off in the bottom left state from Figure 4.63 (predict not taken).

4.16.2 First 4 branches are T, N, T, T

Repeated	T	N	T	T

Pattern				
Predictor value at time of prediction	0	1	0	1
Predicted action	N	N	N	N
Prediction result in steady state	I	C	I	I

Accuracy = 1/4=25%

4.16.3 [10] <4.8> What is the accuracy of the two-bit predictor if this pattern is repeated forever?

4.16.3 Solution

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).

Repeated Pattern	T	N	T	T	N	T	N	T	T	N	T	N	T	T	N	T	N	T	T	N
Predictor value at time of prediction	0	1	0	1	2	1	2	1	2	3	2	3	2	3	3	2	3	2	3	3
Predicted action	N	N	N	N	T	N	T	N	T	T	T	T	T	T	T	T	T	T	T	T
Prediction result in steady state	I	C	I	I	I	I	I	I	C	I	C	I	C	I	I	C	I	C	C	S

4.16.4 [30] <4.8> Design a predictor that would achieve a perfect accuracy if this pattern is repeated forever. Your predictor should be a sequential circuit with one output that provides a prediction (1 for taken, 0 for not taken) and no inputs other than the clock and the control signal that indicates that the instruction is a conditional branch.

4.16.4 Solution: The predictor should be an N-bit shift register, where N is the number of branch outcomes in the target pattern. Th e shift register should be initialized with the pattern itself (0 for NT, 1 for T), and the prediction is always the value in the left most bit of the shift register. The register should be shift ed after each predicted branch.

4.16.5 [10] <4.8> What is the accuracy of your predictor from 4.16.4 if it is given a repeating pattern that is the exact opposite of this one?

4.16.5 Solution: Since the predictor's output is always the opposite of the actual outcome of the branch instruction, the accuracy is zero.

4.16.6 [20] <§4.8> Repeat 4.16.4, but now your predictor should be able to eventually (after a warm-up period during which it can make wrong predictions) start perfectly predicting both this pattern and its opposite. Your predictor should have an input that tells it what the real outcome was. Hint: this input you're your predictor determine which of the two repeating patterns it is given.

4.16.6 The predictor is the same as in part d, except that it should compare its prediction to the actual outcome and invert (logical NOT) all the bits in the shift register if the prediction is incorrect. This predictor still always perfectly predicts the given pattern. For the opposite pattern, the first prediction will be incorrect, so the predictor's state is inverted and after that the predictions are always correct. Overall, there is no warm-up period for the given pattern, and the warm-up period for the opposite pattern is only one branch.