HW#6 (CSC390)- Due 04/22/2016 by midnight

Q1. One way to improve branch performance is to reduce the cost of the taken branch. Thus far, we have assumed the next PC for a branch is selected in the MEM stage, but if we move the branch execution earlier in the pipeline, then fewer instructions need be flushed and thereby pipelining would become more efficient. The designers observed that many branches rely only on simple tests (equality or sign, for example) and that such tests do not require a full ALU operation but can be done with at most a few gates. Equality can be tested by first exclusive ORing their respective bits and then ORing all the results. Figure 1 shows the datapath and control signals of such a MIPS processor. (Pages 316-320 of your text)

Show what happens when the branch is taken in the following instruction sequence, assuming the pipeline is optimized for branches that are not taken and the branch execution is moved to the ID stage:

10 add $10, $4, $6

14 beq $1, $7, 8

18 or $13, $2, $6

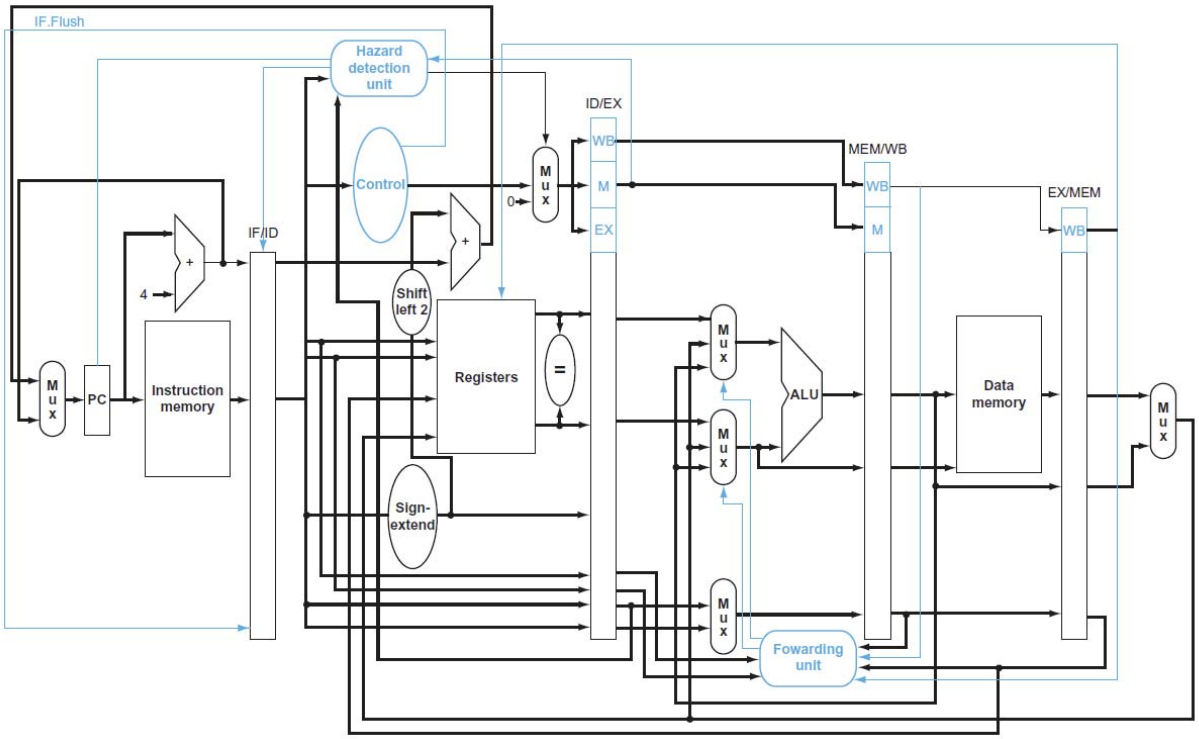
22 and $12, $2, $5

24 add $14, $4, $2

.....

50 sw $4, 50($7)

Use figure 1 to show what happen when a branch is taken. Specifically, use Figure 4.62 as a reference and indicate the input/output values of the PC, adders, pipeline register data and control signals.



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?

4.8.2 [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?

Exercise 4.13

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:

	Instruction Sequence

a.	`SW R16,-100(R6)`
	`LW R4,8(R16)`
	`ADD R5,R4,R4`

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

4.13.1 [10] <4.5> Indicate dependences and their type.

4.13.2 [10] <4.5> Assume there is no forwarding in this pipelined processor. Indicate hazards and add `NOP` instructions to eliminate them.

4.13.3 [10] <4.5> Assume there is full forwarding. Indicate hazards and add `NOP` instructions to eliminate them.

4.13.4 [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?

4.13.5 [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).

4.13.6 [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?

Solution:

4.13.1

	Instruction Sequence	Dependences
a.	I1: SW R16,-100(R6) I2: LW R4,8(R16) I3: ADD R5,R4,R4	RAW on R4 from I2 to I3

4.13.2 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	

a.	SW R16,-100(R6)
	LW R4,8(R16)
	NOP
	NOP
	ADD R5,R4,R4
		Delay I3 to avoid RAW hazard on R4 from I2

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

	Instruction Sequence	

a.	SW R16,-100(R6)
	LW R4,8(R16)
	NOP
	ADD R5,R4,R4
		Delay I3 to avoid RAW hazard on R4 from I2
		Value for R4 is forwarded from I2 now

4.13.4 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 4.13.2, and execution forwarding must add a stall cycle for every NOP we had in 4.13.3. Overall, we get:

	No Forwarding	With Forwarding	Speedup Due to Forwarding
a.	(7 + 2) × 250ps = 2250ps	(7 + 1) × 300ps = 2400ps	0.94 (This is really a slowdown)

4.13.5 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	

a.	`SW R16,-100(R6)`
	`LW R4,8(R16)`
	`ADD R5,R4,R4`
		ALU-ALU forwarding of R4 from I2

4.13.6

	No Forwarding	With ALU-ALU Forwarding Only	Speedup with ALU-ALU Forwarding
a.	(7 + 2) × 250ps = 2250ps	7 × 290ps = 2030ps	1.11