2.a

	Instruction sequence	Dependences
a.	I1: lw $1,40($6) I2: add $6,$2,$2 I3: sw $6,50($1)	RAW on $1 from I1 to I3 RAW on $6 from I2 to I3 WAR on $6 from I1 to I2 and I3
b.	I1: lw $5,-16($5) I2: sw $5,-16($5) I3: add $5,$5,$5	RAW on $5 from I1 to I2 and I3 WAR on $5 from I1 and I2 to I3 WAW on $5 from I1 to I3

2.b 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.	`lw $1,40($6)`
	`add $6,$2,$2`
	`nop`
	`sw $6,50($1)`
		Delay I3 to avoid RAW hazard on $1 from I1

b.	`lw $5,-16($5)`
	`nop`
	`nop`
	`sw $5,-16($5)`
	`add $5,$5,$5`
		Delay I2 to avoid RAW hazard on $5 from I1
		
		Note: no RAW hazard from on $5 from I1 now

4.13.2 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	

a.	`lw $1,40($6)`
	`add $6,$2,$2`
	`sw $6,50($1)`
		No RAW hazard on $1 from I1 (forwarded)

b.	`lw $5,-16($5)`
	`nop`
	`sw $5,-16($5)`
	`add $5,$5,$5`
		Delay I2 to avoid RAW hazard on $5 from I1
		Value for $5 is forwarded from I2 now
		Note: no RAW hazard from on $5 from I1 now

4.13.3 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	Speed-up due to forwarding
a.	(7 + 1) × 300ps = 2400ps	7 × 400ps = 2800ps	0.86 (This is really a slowdown)
b.	(7 + 2) × 200ps = 1800ps	(7 + 1) × 250ps = 2000ps	0.90 (This is really a slowdown)

2.e 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.	`lw $1,40($6)`
	`add $6,$2,$2`
	`nop`
	`sw $6,50($1)`
		Can't use ALU-ALU forwarding, ($1 loaded in MEM)

b.	`lw $5,-16($5)`
	`nop`
	`nop`
	`sw $5,-16($5)`
	`add $5,$5,$5`
		Can't use ALU-ALU forwarding ($5 loaded in MEM)

2.f

	No forwarding	With ALU-ALU forwarding only	Speed-up with ALU-ALU forwarding
a.	(7 + 1) × 300ps = 2400ps	(7 + 1) × 360ps = 2880ps	0.83 (This is really a slowdown)
b.	(7 + 2) × 200ps = 1800ps	(7 + 2) × 220ps = 1980ps	0.91 (This is really a slowdown)

1.a 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

a.	`lw $1,40($6)`
	`beq $2,$0,Lbl`
	`add $2,$3,$4`
	`sw $3,50($4)`
		`IF ID EX MEM WB`
		`IF ED EX MEM WB`
		`IF ID EX MEM WB`
		`*** IF ID EX MEM WB`
			9

b.	`lw $5,-16($5)`
	`sw $4,-16($4)`
	`lw $3,-20($4)`
	`beq $2,$0,Lbl`
	`add $5,$1,$4`
		`IF ID EX MEM WB`
		`IF ED EX MEM WB`
		`IF ID EX MEM WB`
		`*** *** *** IF ID EX MEM WB`
		`IF ID EX MEM WB`
			12

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

1.b 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 instruction, the change would help eliminate some stall cycles.

	Instructions Executed	Cycles with 5 stages	Cycles with 4 stages	Speed-up
a.	4	4 + 4 = 8	3 + 4 = 7	8/7 = 1.14
b.	5	4 + 5 = 9	3 + 5 = 8	9/8 = 1.13

1.c 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	Speed-up
a.	4	1	4 + 4 + 1 × 2 = 10	4 + 4 + 1 × 1 = 9	10/9 = 1.11
b.	5	1	4 + 5 + 1 × 2 = 11	4 + 5 + 1 × 1 = 10	11/10 = 1.10

1.d The number of cycles for the (normal) 5-stage and the (combined EX/MEM) 4-stage pipeline is already computed in 4.14.2. 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	Speed-up
a.	130ps (MEM)	150ps (MEM + 20ps)	(8 × 130)/(7 × 150) = 0.99
b.	220ps (MEM)	240ps (MEM + 20ps)	(9 × 220)/(8 × 240) = 1.03

1.e

	New ID latency	New EX latency	New cycle time	Old cycle time	Speed-up
a.	180ps	80ps	180ps (ID)	130ps (MEM)	(10 × 130)/(9 × 180) = 0.80
b.	150ps	160ps	220ps (MEM)	220ps (MEM)	(11 × 220)/(10 × 220) = 1.10

1.f The cycle time remains unchanged: a 20ps 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 speed-up from this change will be below 1 (a slowdown). In 4.14.3 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)	Speed-up
a.	4 + 4 + 1 × 2 = 10	10 × 130ps = 1300ps	4 + 4 + 1 × 3 = 11	11 × 130ps = 1430ps	0.91
b.	4 + 5 + 1 × 2 = 11	11 × 220ps = 2420ps	4 + 5 + 1 × 3 = 12	12 × 220ps = 2640ps	0.92