Solution 4.12

4.12.1

	Pipelined	Single-cycle
a.	500ps	1650ps
b.	200ps	800ps

4.12.2

	Pipelined	Single-cycle
a.	2500ps	1650ps
b.	1000ps	800ps

4.12.3

	Stage to split	New clock cycle time
a.	MEM	400ps
b.	IF	190ps

4.12.4

a.	25%
b.	45%

4.12.5

a.	65%
b.	60%

4.12.6 We already computed clock cycle times for pipelined and single cycle organizations in 4.12.1, and the multi-cycle organization has the same clock cycle time as the pipelined organization. We will compute execution times relative to the pipelined organization. In single-cycle, every instruction takes one (long) clock cycle. In pipelined, a long-running program with no pipeline stalls completes one instruction in every cycle. Finally, a multi-cycle organization completes a `lw` in 5 cycles, a `sw` in 4 cycles (no WB), an ALU instruction in 4 cycles (no MEM), and a `beq` in 4 cycles (no WB). So we have the speed-up of pipeline

	Multi-cycle execution time is X times pipelined execution time, where X is	Single-cycle execution time is X times pipelined execution time, where X is
a.	0.15 × 5 + 0.85 × 4 = 4.15	1650ps/500ps = 3.30
b.	0.30 × 5 + 0.70 × 4 = 4.30	800ps/200ps = 4.00

Solution 4.13

4.13.1

	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

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.	`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

4.13.3 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	

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

4.13.4 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	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)

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.	`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)

4.13.6

	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)

Solution 4.20

4.20.1

	Instruction sequence	RAW	WAR	WAW

a.	`I1: lw $1,40($2)`
	`I2: add $2,$3,$3`
	`I3: add $1,$1,$2`
	`I4: sw $1,20($2)`
		($1) I1 to I3
		($2) I2 to I3, I4
		($1) I3 to I4
			($2) I1 to I2	($1) I1 to I3

b.	`I1: add $1,$2,$3`
	`I2: sw $2,0($1)`
	`I3: lw $1,4($2)`
	`I4: add $2,$2,$1`
		($1) I1 to I2
		($1) I3 to I4
			($2) I1, I2, I3 to I4
			($1) I1, I2 to I3
				($1) I1 to I3

4.20.2 Only RAW dependences can become data hazards. With forwarding, only RAW dependences from a load to the very next instruction become hazards.

Without forwarding, any RAW dependence from an instruction to one of the following three instructions becomes a hazard:

	Instruction sequence	With forwarding	Without forwarding
a.	I1: lw $1,40($2) I2: add $2,$3,$3 I3: add $1,$1,$2 I4: sw $1,20($2)		($1) I1 to I3 ($2) I2 to I3, I4 ($1) I3 to I4
b.	I1: add $1,$2,$3 I2: sw $2,0($1) I3: lw $1,4($2) I4: add $2,$2,$1	($1) I3 to I4	($1) I1 to I2 ($1) I3 to I4

4.20.3 With forwarding, only RAW dependences from a load to the next two instructions become hazards because the load produces its data at the end of the second MEM stage. Without forwarding, any RAW dependence from an instruction to one of the following 4 instructions becomes a hazard:

	Instruction sequence	With forwarding	RAW
a.	I1: lw $1,40($2) I2: add $2,$3,$3 I3: add $1,$1,$2 I4: sw $1,20($2)	($1) I1 to I3	($1) I1 to I3 ($2) I2 to I3, I4 ($1) I3 to I4
b.	I1: add $1,$2,$3 I2: sw $2,0($1) I3: lw $1,4($2) I4: add $2,$2,$1	($1) I3 to I4	($1) I1 to I2 ($1) I3 to I4

4.20.4

	Instruction sequence	RAW
a.	I1: lw $1,40($2) I2: add $2,$3,$3 I3: add $1,$1,$2 I4: sw $1,20($2)	($1) I1 to I3 (0 overrides 1) ($2) I2 to I3 (2000 overrides 31)
b.	I1: add $1,$2,$3 I2: sw $2,0($1) I3: lw $1,4($2) I4: add $2,$2,$1	($1) I1 to I2 (2563 overrides 63)

one-cycle stall if the instruction that immediately follows a load is dependent on the load. We have:

	Instruction sequence with forwarding stalls	Execution without forwarding	Values after execution
a.	I1: lw $1,40($2) I2: add $2,$3,$3 I3: add $1,$1,$2 I4: sw $1,20($2)	$1 = 0 (I4 and after) $2 = 2000 (after I4) $1 = 32 (after I4)	$0 = 0 $1 = 32 $2 = 2000 $3 = 1000
b.	I1: add $1,$2,$3 I2: sw $2,0($1) I3: lw $1,4($2) Stall I4: add $2,$2,$1	$1 = 2563 (Stall and after) $1 = 0 (after I4) $2 = 2626 (after I4)	$0 = 0 $1 = 0 $2 = 2626 $3 = 2500

4.20.6

	Instruction sequence with forwarding stalls	Correct execution	Sequence with NOPs

a.	I1: lw $1,40($2)
	I2: add $2,$3,$3
	I3: add $1,$1,$2
	I4: sw $1,20($2)
		I1: lw $1,40($2)
		I2: add $2,$3,$3
		Stall
		Stall
		I3: add $1,$1,$2
		Stall
		Stall
		I4: sw $1,20($2)
			lw $1,40($2)
			add $2,$3,$3
			nop
			nop
			add $1,$1,$2
			nop
			nop
			sw $1,20($2)

b.	I1: add $1,$2,$3
	I2: sw $2,0($1)
	I3: lw $1,4($2)
	Stall
	I4: add $2,$2,$1
		I1: add $1,$2,$3
		Stall
		Stall
		I2: sw $2,0($1)
		I3: lw $1,4($2)
		Stall
		Stall
		I4: add $2,$2,$1
			add $1,$2,$3
			nop
			nop
			sw $2,0($1)
			lw $1,4($2)
			nop
			nop
			add $2,$2,$1

Solution 4.21

4.21.1

a.	`lw  $1,40($6)`
	`nop`
	`nop`
	`add $2,$3,$1`
	`add $1,$6,$4`
	`nop`
	`sw  $2,20($4)`
	`and $1,$1,$4`

b.	`add $1,$5,$3`
	`nop`
	`nop`
	`sw  $1,0($2)`
	`lw  $1,4($2)`
	`nop`
	`nop`
	`add $5,$5,$1`
	`sw  $1,0($2)`

4.21.2 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.

a.	`I1: lw  $7,40($6)`
	`I3: add $1,$6,$4`
	`nop`
	`I2: add $2,$3,$7`
	`I5: and $1,$1,$4`
	`nop`
	`I4: sw  $2,20($4)`
		Produce $7 instead of $1
		Moved up to fill NOP slot
		
		Use $7 instead of $1
		Moved up to fill NOP slot

b.	`I1: add $7,$5,$3`
	`I3: lw  $1,4($2)`
	`nop`
	`I2: sw  $7,0($2)`
	`I4: add $5,$5,$1`
	`I5: sw  $1,0($2)`
		Produce $7 instead of $1
		Moved up to fill NOP slot
		
		Use $7 instead of $1

4.21.3 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.

a.	I2 gets the value of $1 from before I1, not from I1 as it should.
b.	I4 gets the value of $1 from I1, not from I3 as it should.

4.21.4 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 which 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 1	2	3	4	5	Signals

a.	`lw $1,40($6)`
	`add $2,$3,$1`
	`add $1,$6,$4`
	`sw $2,20($4)`
	`and $1,$1,$4`
		IF	ID
			IF
				EX
				ID
				IF
					MEM
					***
					***
						WB
						EX
						ID
						IF
							1: PCWrite = 1, ALUin1 = X, ALUin2 = X
							2: PCWrite = 1, ALUin1 = X, ALUin2 = X
							3: PCWrite = 1, ALUin1 = 0, ALUin2 = 0
							4: PCWrite = 0, ALUin1 = X, ALUin2 = X
							5: PCWrite = 1, ALUin1 = 0, ALUin2 = 2

b.	`add $1,$5,$3`
	`sw $1,0($2)`
	`lw $1,4($2)`
	`add $5,$5,$1`
	`sw $1,0($2)`
		IF	ID
			IF
				EX
				ID
				IF
					MEM
					EX
					ID
						WB
						MEM
						EX
						ID
						IF
							1: PCWrite = 1, ALUin1 = X, ALUin2 = X
							2: PCWrite = 1, ALUin1 = X, ALUin2 = X
							3: PCWrite = 1, ALUin1 = 0, ALUin2 = 0
							4: PCWrite = 1, ALUin1 = 0, ALUin2 = 1
							5: PCWrite = 1, ALUin1 = 0, ALUin2 = 0

4.21.5 The instruction that is currently in the ID stage needs to be stalled if it depends on a value produced by the instruction in the EX or the instruction in the MEM stage. So we need to check the destination register of these two instructions. For the instruction in the EX stage, we need to check Rd for R-type instructions and Rd for loads. For the instruction in the MEM stage, the destination register is already selected (by the Mux in the EX stage) so we need to check that register number (this is the bottommost output of the EX/MEM pipeline register). The additional inputs to the hazard detection unit are register Rd from the ID/EX pipeline register and the output number of the output register from the EX/MEM

pipeline register. The Rt field from the ID/EX register is already an input of the hazard detection unit in Figure 4.60.

No additional outputs are needed. We can stall the pipeline using the three output signals that we already have.

4.21.6 As explained for 4.21.5, we only need to specify the value of the PCWrite signal, because IF/IDWrite is equal to PCWrite and the ID/EXzero signal is its opposite. We have:

	Instruction sequence	First five cycles 1	2	3	4	5	Signals

a.	`lw $1,40($6)`
	`add $2,$3,$1`
	`add $1,$6,$4`
	`sw $2,20($4)`
	`and $1,$1,$4`
		IF	ID
			IF
				EX
				ID
				IF
					MEM
					***
					***
						WB
						***
						***
						***
							1: PCWrite = 1
							2: PCWrite = 1
							3: PCWrite = 1
							4: PCWrite = 0
							5: PCWrite = 0

b.	`add $1,$5,$3`
	`sw $1,0($2)`
	`lw $1,4($2)`
	`add $5,$5,$1`
	`sw $1,0($2)`
		IF	ID
			IF
				EX
				ID
				IF
					MEM
					***
					***
						WB
						***
						***
						***
							1: PCWrite = 1
							2: PCWrite = 1
							3: PCWrite = 1
							4: PCWrite = 0
							5: PCWrite = 0