4 Solutions

Solution 4.1

4.1.1 The values of the signals are as follows:

	RegWrite	MemRead	ALUMux	MemWrite	ALUOp	RegMux	Branch
a.	1	0	0 (Reg)	0	Add	1 (ALU)	0
b.	1	1	1 (Imm)	0	Add	1 (Mem)	0

ALUMux is the control signal that controls the Mux at the ALU input, 0 (Reg) selects the output of the register file and 1 (Imm) selects the immediate from the instruction word as the second input to the ALU.

RegMux is the control signal that controls the Mux at the Data input to the register file, 0 (ALU) selects the output of the ALU and 1 (Mem) selects the output of memory.

A value of X is a "don't care" (does not matter if signal is 0 or 1)

4.1.2 Resources performing a useful function for this instruction are:

a.	All except Data Memory and branch Add unit
b.	All except branch Add unit and second read port of the Registers

4.1.3

	Outputs that are not used	No outputs
a.	Branch Add	Data Memory
b.	Branch Add, second read port of Registers	None (all units produce outputs)

4.1.4 One long path for `and` instruction is to read the instruction, read the registers, go through the ALUMux, perform the ALU operation, and go through the Mux that controls the write data for Registers (I-Mem, Regs, Mux, ALU, and Mux). The other long path is similar, but goes through Control while registers are read (I- Mem, Control, Mux, ALU, Mux). There are other paths but they are shorter, such as the PC increment path (only Add and then Mux), the path to prevent branching (I-Mem, Control, Mux uses Branch signal to select the PC + 4 input as the new value for PC), the path that prevents a memory write (only I-Mem and then Control, etc).

a.	Control is faster than registers, so the critical path is I-Mem, Regs, Mux, ALU, Mux.
b.	Control is faster than registers, so the critical path is I-Mem, Regs, Mux, ALU, Mux.

4.1.5 One long path is to read instruction, read registers, use the Mux to select the immediate as the second ALU input, use ALU (compute address), access D-Mem, and use the Mux to select that as register data input, so we have I-Mem, Regs, Mux, ALU, D-Mem, Mux. The other long path is similar, but goes through Control instead of Regs (to generate the control signal for the ALU MUX). Other paths are shorter, and are similar to shorter paths described for 4.1.4.

a.	Control is faster than registers, so the critical path is I-Mem, Regs, Mux, ALU, D-Mem, Mux.
b.	Control is faster than registers, so the critical path is I-Mem, Regs, Mux, ALU, Mux.

4.1.6 This instruction has two kinds of long paths, those that determine the branch condition and those that compute the new PC. To determine the branch condition, we read the instruction, read registers or use the Control unit, then use the ALU Mux and then the ALU to compare the two values, then use the Zero output of the ALU to control the Mux that selects the new PC. As in 4.1.4 and 4.1.5:

a.	The first path (through Regs) is longer.
b.	The first path (through Regs) is longer.

To compute the PC, one path is to increment it by 4 (Add), add the offset (Add), and select that value as the new PC (Mux). The other path for computing the PC is to Read the instruction (to get the offset), use the branch Add unit and Mux. Both of the compute-PC paths are shorter than the critical path that determines the branch condition, because I-Mem is slower than the PC + 4 Add unit, and because ALU is slower than the branch Add.

Solution 4.2

4.2.1 Existing blocks that can be used for this instruction are:

a.	This instruction uses instruction memory, both existing read ports of Registers, the ALU, and the write port of Registers.
b.	This instruction uses the instruction memory, one of the existing register read ports, the path that passed the immediate to the ALU, and the register write port.

4.2.2 New functional blocks needed for this instruction are:

a.	Another read port in Registers (to read Rx) and either a second ALU (to add Rx to Rs + Rt) or a third input to the existing ALU.
b.	We need to extend the existing ALU to also do shifts (adds a SLL ALU operation).

4.2.3 The new control signals are:

a.	We need a control signal that tells the new ALU what to do, or if we extended the existing ALU we need to add a new ADD3 operation.
b.	We need to change the ALU Operation control signals to support the added SLL operation in the ALU.

4.2.4 Clock cycle time is determined by the critical path, which for the given latencies happens to be to get the data value for the load instruction: I-Mem (read instruction), Regs (takes longer than Control), Mux (select ALU input), ALU, Data Memory, and Mux (select value from memory to be written into Registers). The latency of this path is 400ps + 200ps + 30ps + 120ps + 350ps + 30ps = 1130ps.

	New clock cycle time
a.	1130ps (No change, Add units are not on the critical path).
b.	1230 (1130ps + 100ps, Regs are on the critical path)

4.2.5 The speed-up comes from changes in clock cycle time and changes to the number of clock cycles we need for the program:

	Benefit
a.	Speed-up is 1 (no change in number of cycles, no change in clock cycle time).
b.	We need 5% fewer cycles for a program, but cycle time is 1230 instead of 1130, so we have a speed-up of (1/0.95) × (1130/1230) = 0.97, which means we actually have a small slowdown.

4.2.6 The cost is always the total cost of all components (not just those on the critical path, so the original processor has a cost of I-Mem, Regs, Control, ALU, D-Mem, 2 Add units and 3 Mux units, for a total cost of 1000 + 200 + 500 + 100 + 2000 + 2 × 30 + 3 × 10 = 3890.

We will compute cost relative to this baseline. The performance relative to this baseline is the speed-up we computed in 4.2.5, and our cost/performance relative to the baseline is as follows:

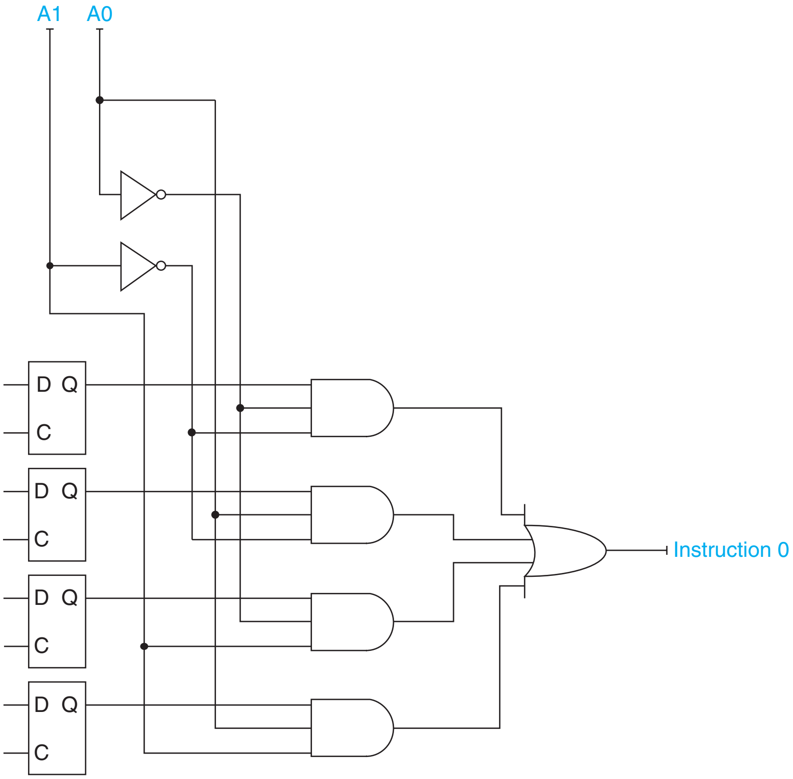
	New cost	Relative cost	Cost/Performance
a.	3890 + 2 × 20 = 3930	3930/3890 = 1.01	1.01/1 = 1.01. We are paying a bit more for the same performance.
b.	3890 + 200 = 4090	4090/3890 = 1.05	1.05/0.97 = 1.08. We are paying some more and getting a small slowdown, so out cost/ performance gets worse.

Solution 4.3

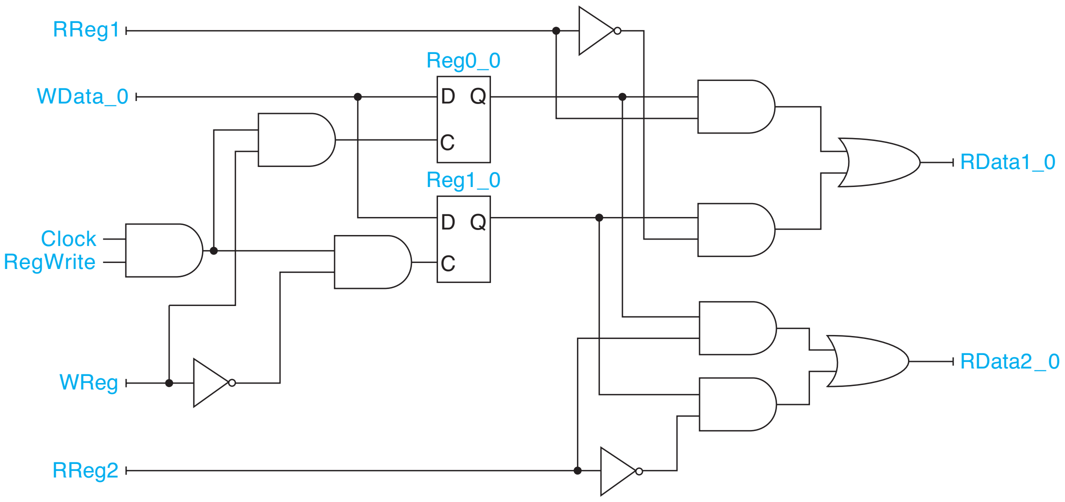
4.3.1

a.	Both. It is mostly flip-flops, but it has logic that controls which flip-flops get read or written in each cycle
b.	Both. It is mostly flip-flops, but it has logic that controls which flip-flops get read or written in each cycle

4.3.2

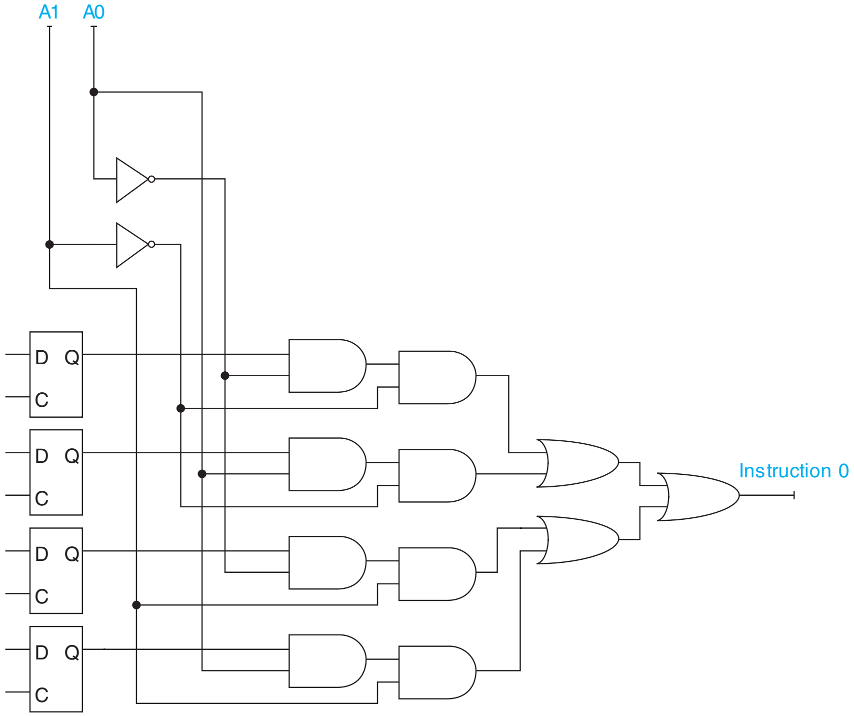
a.	 A1 A0 D Q C D Q C D Q C D Q C Instruction 0 This shows the lowermost bit of each word. This schematic is repeated 7 more times for the remaining seven bits. Note that there are no connections for D and C flip-flop inputs because datapath figures do not specify how instruction memory is written.

b.



This is the schematic for the lowermost bit, it needs to be repeated 7 more times for the remaining bits. RReg1 is the Read Register 1 input, RReg2 is the Read Register 2 input, WReg is the Write Register input, WData is the Write Data input. RData1 and RData2 are Read Data 1 and Read Data 2 outputs. Data outputs and input have "_0" to denote that this is only bit 0 of the 8-bit signal.

4.3.3

a.	 A1 A0 D Q C D Q C D Q C D Q C Instruction 0
b.	No change, there are no gates with more then 2 inputs in the schematic.

4.3.4 The latency of a path is the latency from an input (or a D-element output) to an output (or D-element input). The latency of the circuit is the latency of the path with the longest latency. Note that there are many correct ways to design the circuit in 4.3.2, and for each solution to 4.3.2 there is a different solution for this problem.

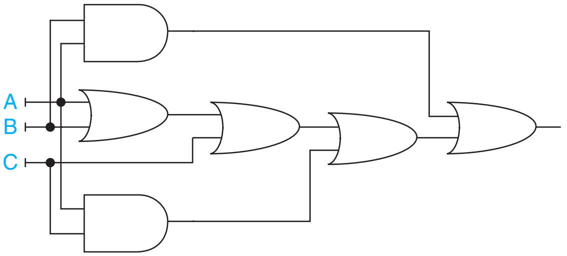
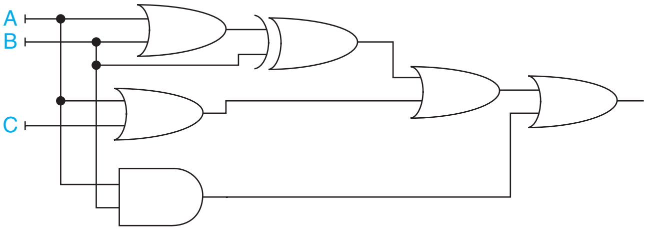
4.3.5 The cost of the implementation is simply the total cost of all its components. Note that there are many correct ways to design the circuit in 4.3.2, and for each solution to 4.3.2 there is a different solution for this problem.

4.3.6

a.	Because multi-input AND and OR gates have the same latency as 2-input ones, we can use many-input gates to reduce the number of gates on the path from inputs to outputs. The schematic shown for 4.3.2 turns out to already be optimal.
b.	A three-input or a four-input gate has a lower latency than a cascade of two 2-input gates. This means that shorter overall latency is achieved by using 3- and 4-input gates rather than cascades of 2-input gates. In our schematic shown for 4.3.2, we should replace the three 2-input AND gates used for Clock, RegWrite, and WReg signals with two 3-input AND gates that directly determine the value of the C input for each D-element.

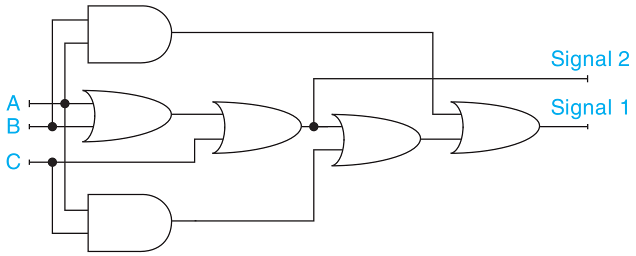
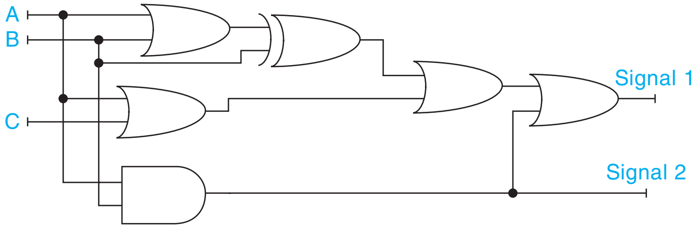
Solution 4.4

4.4.1 We show the implementation and also determine the latency (in gates) needed for 4.4.2.

	Implementation	Latency in gates
a.		4
b.		4

4.4.2 See answer for 4.4.1 above.

4.4.3

	Implementation
a.	
b.	

4.4.4

a.	There are four OR gates on the critical path, for a total of 136ps
b.	The critical path consists of OR, XOR, OR, and OR, for a total of 510ps

4.4.5

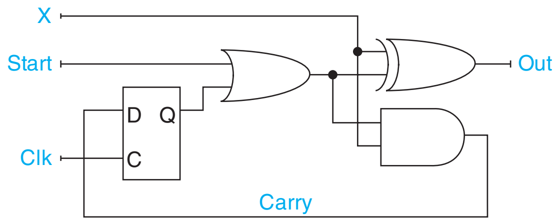
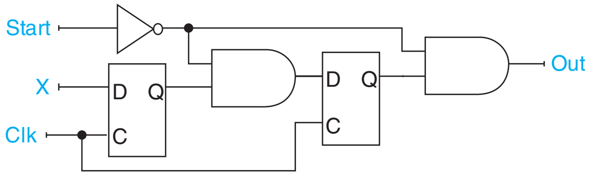
a.	The cost is 2 AND gates and 4 OR gates, for a total cost of 16.
b.	The cost is 1 AND gate, 4 OR gates, and 1 XOR gate, for a total cost of 12.

4.4.6 We already computed the cost of the combined circuit. Now we determine the cost of the separate circuits and the savings.

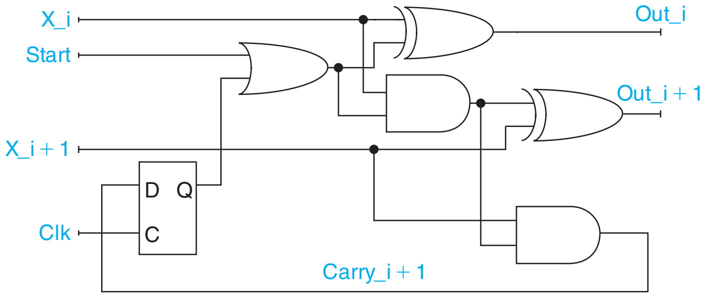
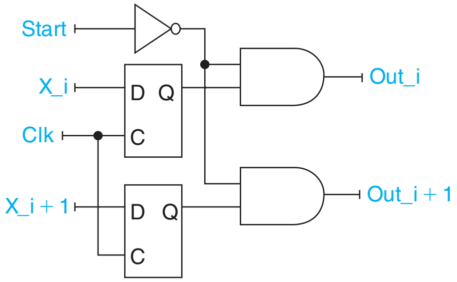
	Combinend cost	Separate cost	Saved
a.	16	22 (+2 OR gates)	(22 – 16)/22 = 27%
b.	12	14 (+1 AND gate)	(14 – 12)/14 = 14%

Solution 4.5

4.5.1

a.	
b.	

4.5.2

a.	 X_i Start X_i + 1 Clk D Q C Out_i Out_i + 1 Carry_i + 1
b.	 Start X_i Clk X_i + 1 D Q C D Q C Out_i Out_i + 1

4.5.3

	Cycle time	Operation time
a.	90ps (OR, AND, D)	32 × 90ps = 2880ps
b.	170ps (NOT, AND, D)	32 × 170ps = 5440ps

4.5.4

	Cycle time	Speed-up
a.	120ps (OR, AND, AND, D)	(32 × 90ps)/(16 × 120ps) = 1.50
b.	90ps (NOT, AND)	(32 × 170ps)/(16 × 90ps) = 3.78

4.5.5

	Circuit 1	Circuit 2
a.	14 (1 AND, 1 OR, 1 XOR, 1 D)	20 (2 AND, 1 OR, 2 XOR, 1 D)
b.	29 (1 NOT, 2 AND, 2 D)	29 (1 NOT, 2 AND, 2 D)

4.5.6

	Cost/Performance for Circuit 1	Cost/Performance for Circuit 2	Circuit 1 versus Circuit 2
a.	14 × 32 × 90 = 40320	20 × 16 × 120 = 38400	Cost/performance of Circuit 2 is better by about 4.7%
b.	29 × 32 × 170 = 157760	29 × 16 × 90 = 41760	Cost/performance of Circuit 2 is better by about 73.5%

Solution 4.6

4.6.1 I-Mem takes longer than the Add unit, so the clock cycle time is equal to the latency of the I-Mem:

a.	400ps
b.	500ps

4.6.2 The critical path for this instruction is through the instruction memory, Sign-extend and Shift-left-2 to get the offset, Add unit to compute the new PC, and Mux to select that value instead of PC + 4. Note that the path through the other

Add unit is shorter, because the latency of I-Mem is longer that the latency of the Add unit. We have:

a.	400ps + 20ps + 2ps + 100ps + 30ps = 552ps
b.	500ps + 90ps + 20ps + 150ps + 100ps = 860ps

4.6.3 Conditional branches have the same long-latency path that computes the branch address as unconditional branches do. Additionally, they have a long-latency path that goes through Registers, Mux, and ALU to compute the PCSrc condition. The critical path is the longer of the two, and the path through PCSrc is longer for these latencies:

a.	400ps + 200ps + 30ps + 120ps + 30ps = 780ps
b.	500ps + 220ps + 100ps + 180ps + 100ps = 1100ps

4.6.4

a.	All instructions except jumps that are not PC-relative (jal, jalr, j, jr)
b.	Loads and stores

4.6.5

a.	None. I-Mem is slower, and all instructions (even NOP) need to read the instruction.
b.	Loads and stores.

4.6.6 Of the two instruction (`bne` and `add`), `bne` has a longer critical path so it determines the clock cycle time. Note that every path for `add` is shorter or equal to than the corresponding path for `bne`, so changes in unit latency will not affect this. As a result, we focus on how the unit's latency affects the critical path of `bne`:

a.	This unit is not on the critical path, so changes to its latency do not affect the clock cycle time unless the latency of the unit becomes so large to create a new critical path through this unit, the branch add, and the PC Mux. The latency of this path is 230ps and it needs to be above 780ps, so the latency of the Add-4 unit needs to be more 650ps for it to be on the critical path.
b.	This unit is not used by BNE nor by ADD, so it cannot affect the critical path for either instruction.

Solution 4.7

4.7.1 The longest-latency path for ALU operations is through I-Mem, Regs, Mux (to select ALU operand), ALU, and Mux (to select value for register write). Note that the only other path of interest is the PC-increment path through Add (PC + 4) and Mux, which is much shorter. So for the I-Mem, Regs, Mux, ALU, Mux path we have:

a.	400ps + 200ps + 30ps + 120ps + 30ps = 780ps
b.	500ps + 220ps + 100ps + 180ps + 100ps = 1100ps

4.7.2 The longest-latency path for lw is through I-Mem, Regs, Mux (to select ALU input), ALU, D-Dem, and Mux (to select what is written to register). The only other interesting paths are the PC-increment path (which is much shorter) and the path through Sign-extend unit in address computation instead of through Registers. However, Regs has a longer latency than Sign-extend, so for I-Mem, Regs, Mux, ALU, D-Mem, and Mux path we have:

a.	400ps + 200ps + 30ps + 120ps + 350ps + 30ps = 1130ps
b.	500ps + 220ps + 100ps + 180ps + 1000ps + 100ps = 2100ps

4.7.3 The answer is the same as in 4.7.2 because the lw instruction has the longest critical path. The longest path for sw is shorter by one Mux latency (no write to register), and the longest path for add or bne is shorter by one D-Mem latency.

4.7.4 The data memory is used by lw and sw instructions, so the answer is:

a.	20% + 10% = 30%
b.	35% + 15% = 50%

4.7.5 The sign-extend circuit is actually computing a result in every cycle, but its output is ignored for add and not instructions. The input of the sign-extend circuit is needed for addi (to provide the immediate ALU operand), beq (to provide the PC-relative offset), and lw and sw (to provide the offset used in addressing memory) so the answer is:

a.	15% + 20% + 20% + 10% = 65%
b.	5% + 15% + 35% + 15% = 70%

4.7.6 The clock cycle time is determined by the critical path for the instruction that has the longest critical path. This is the lw instruction, and its critical path goes through I-Mem, Regs, Mux, ALU, D-Mem, and Mux so we have:

a.	I-Mem has the longest latency, so we reduce its latency from 400ps to 360ps, making the clock cycle 40ps shorter. The speed-up achieved by reducing the clock cycle time is then 1130ps/1090ps = 1.037
b.	D-Mem has the longest latency, so we reduce its latency from 1000ps to 900ps, making the clock cycle 100ps shorter. The speed-up achieved by reducing the clock cycle time is then 2100ps/2000ps = 1.050

Solution 4.8

4.8.1 To test for a stuck-at-0 fault on a wire, we need an instruction that puts that wire to a value of 1 and has a different result if the value on the wire is stuck at zero:

a.	Bit 7 of the instruction word is only used as part of an immediate/offset part of the instruction, so one way to test would be to execute ADDI $1, zero, 128 which is supposed to place a value of 128 into $1. If instruction bit 7 is stuck at zero, $1 will be zero because value 128 has all bits at zero except bit 7.
b.	The only instructions that set this signal to 1 are loads. We can test by filling the data memory with zeros and executing a load instruction from a non-zero address, e.g., LW $1, 1024(zero). After this instruction, the value in $1 is supposed to be zero. If the MemtoReg signal is stuck at 0, the value in the register will be 1024 (the Mux selects the ALU output (1024) instead of the value from memory).

4.8.2 The test for stuck-at-zero requires an instruction that sets the signal to 1 and the test for stuck-at-1 requires an instruction that sets the signal to 0. Because the signal cannot be both 0 and 1 in the same cycle, we cannot test the same signal simultaneously for stuck-at-0 and stuck-at-1 using only one instruction. The test for stuck-at-1 is analogous to the stuck-at-0 test:

a.	We can use ADDI $1, zero, 0 which is supposed to put a value of 0 in $1. If Bit 7 of the instruction word is stuck at 1, the immediate operand becomes 128 and $1 becomes 128 instead of 0.
b.	We cannot reliably test for this fault, because all instructions that set the MemtoReg signal to zero also set the ReadMem signal to zero. If one of these instructions is used as a test for MemtoReg stuck-at-1, the value written to the destination register is “random” (whatever noise is there at the data output of Data Memory). This value could be the same as the value already in the register, so if the fault exists the test may not detect it.

4.8.3

a.	It is possible to work around this fault, but it is very difficult. We must find all instructions that have zero in this bit of the offset or immediate operand and replace them with a sequence of “safe” instruction. For example, a load with such an offset must be replaced with an instruction that subtracts 128 from the address register, then the load (with the offset larger by 128 to set bit 7 of the offset to 1), then subtract 128 from the address register.
b.	We cannot work around this problem, because it prevents all instructions from storing their result in registers, except for load instructions. Load instructions only move data from memory to registers, so they cannot be used to emulate ALU operations “broken” by the fault.

4.8.4

a.	If MemRead is stuck at 0, data memory is read for every instruction. However, for non-load instructions the value from memory is discarded by the Mux that selects the value to be written to the Register unit. As a result, we cannot design this kind of test for this fault, because the processor still operates correctly (although inefficiently).
b.	To test for this fault, we need an instruction whose opcode is zero and MemRead is 1. However, instructions with a zero opcode are ALU operations (not loads), so their MemRead is 0. As a result, we cannot design this kind of test for this fault, because the processor operates correctly.

4.8.5

a.	If Jump is stuck-at-1, every instruction updates the PC as if it were a jump instruction. To test for this fault, we can execute an ADDI with a non-zero immediate operand. If the Jump signal is stuck-at-1, the PC after the ADDI executes will not be pointing to the instruction that follows the ADDI.
b.	To test for this fault, we need an instruction whose opcode is zero and Jump is 1. However, the opcode for the jump instruction is non-zero. As a result, we cannot design this kind of test for this fault, because the processor operates correctly.

4.8.6 Each single-instruction test "covers" all faults that, if present, result in different behavior for the test instruction. To test for as many of these faults as possible in a single instruction, we need an instruction that sets as many of these signals to a value that would be changed by a fault. Some signals cannot be tested using this single-instruction method, because the fault on a signal could still result in completely correct execution of all instruction that trigger the fault.

Solution 4.9

4.9.1

	Binary	Hexadecimal
a.	100011 00110 00001 0000000000101000	8CC10028
b.	000101 00001 00010 1111111111111111	1422FFFF

4.9.2

	Read register 1	Actually read?	Read register 2	Actually read?
a.	6 (00110_b)	Yes	1 (00001_b)	Yes (but not used)
b.	1 (00001_b)	Yes	2 (00010_b)	Yes

4.9.3

	Read register 1	Register actually written?
a.	1 (00001_b)	Yes
b.	Either 2 (00010_b) of 31 (11111_b) (don't know because RegDst is X)	No

4.9.4

	Control signal 1	Control signal 2
a.	RegDst = 0	MemRead = 1
b.	RegWrite = 0	MemRead = 0

4.9.5 We use I31 through I26 to denote individual bits of Instruction[31:26], which is the input to the Control unit:

a.	RegDst = NOT I31
b.	RegWrite = (NOT I28 AND NOT I27) OR (I31 AND NOT I29)

4.9.6 If possible, we try to reuse some or all of the logic needed for one signal to help us compute the other signal at a lower cost:

a.	RegDst = NOT I31
	MemRead = I31 AND NOT I29

b.	MemRead = I31 AND NOT I29
	RegWrite = (NOT I28 AND NOT I27) OR MemRead

Solution 4.10

To solve problems in this exercise, it helps to first determine the latencies of different paths inside the processor. Assuming zero latency for the Control unit, the critical path is the path to get the data for a load instruction, so we have I-Mem, Mux, Regs, Mux, ALU, D-Mem and Mux on this path.

4.10.1 The Control unit can begin generating MemWrite only after I-Mem is read. It must finish generating this signal before the end of the clock cycle. Note that MemWrite is actually a write-enable signal for D-Mem flip-flops, and the actual write is triggered by the edge of the clock signal, so MemWrite need not arrive before that time. So the Control unit must generate the MemWrite in one clock cycle, minus the I-Mem access time:

	Critical path	Maximum time to generate MemWrite
a.	400ps + 30ps + 200ps + 30ps + 120ps + 350ps + 30ps = 1160ps	1160ps – 400ps = 760ps
b.	500ps + 100ps + 220ps + 100ps + 180ps + 1000ps + 100ps = 2200ps	2200ps – 500ps = 1700ps

4.10.2 All control signals start to be generated after I-Mem read is complete. The most slack a signal can have is until the end of the cycle, and MemWrite and RegWrite are both needed only at the end of the cycle, so they have the most slack. The time to generate both signals without increasing the critical path is the one computed in 4.10.1.

4.10.3 MemWrite and RegWrite are only needed by the end of the cycle. RegDst, Jump, and MemtoReg are needed one Mux latency before the end of the cycle, so they are more critical than MemWrite and RegWrite. Branch is needed two Mux latencies before the end of the cycle, so it is more critical than these. MemRead is needed one D-Mem plus one Mux latency before the end of the cycle, and D-Mem has more latency than a Mux, so MemRead is more critical than Branch. ALUOp must get to ALU control in time to allow one ALU Ctrl, one ALU, one D-Mem, and one Mux latency before the end of the cycle. This is clearly more critical than MemRead. Finally, ALUSrc must get to the pre-ALU Mux in time, one Mux, one ALU, one D-Mem, and one Mux latency before the end of the cycle. Again, this is more critical than MemRead. Between ALUOp and ALUSrc, ALUOp is more critical than ALUSrc if ALU control has more latency than a Mux. If ALUOp is the most critical, it must be generated one ALU Ctrl latency before the critical-path signals can go through Mux, Regs, and Mux. If the ALUSrc signal is the most critical, it must be generated while the critical path goes through Mux and Regs. We have

	The most critical control signal is	Time to generate it without affecting the clock cycle time
a.	ALUOp (50ps > 30ps)	30ps + 200ps + 30ps – 50ps = 210ps
b.	ALUSrc (100ps > 55ps)	100ps + 220ps = 320ps

For the next three problems, it helps to compute for each signal how much time we have to generate it before it starts affecting the critical path. We already did this for RegDst and RegWrite in 4.10.1, and in 4.10.3 we described how to do it for the remaining control signals. We have:

	RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
a.	730ps	730ps	700ps	380ps	730ps	210ps	760ps	230ps	760ps
b.	1600ps	1600ps	1500ps	600ps	1600ps	365ps	1700ps	320ps	1700ps

The difference between the allowed time and the actual time to generate the signal is called “slack”. For this problem, the allowed time will be the maximum time the signal can take without affecting clock cycle time. If slack is positive, the signal arrives before it is actually needed and it does not affect clock cycle time. If the slack is positive, the signal is late and the clock cycle time must be adjusted. We now compute the clack for each signal:

	RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
a.	10ps	0ps	100ps	–20ps	30ps	10ps	50ps	30ps	–40ps
b.	0ps	0ps	100ps	100ps	200ps	–35ps	200ps	–80ps	0ps

4.10.4 With this in mind, the clock cycle time is what we computed in 4.10.1, plus the absolute value of the most negative slack. We have:

	Control signal with the most negative slack is	Clock cycle time with ideal Control unit (from 4.10.1)	Actual clock cycle time with these signal latencies
a.	RegWrite (–40ps)	1160ps	1200ps
b.	ALUSrc (–80ps)	2200ps	2280ps

4.10.5 It only makes sense to pay to speed-up signals with negative slack, because improvements to signals with positive slack cost us without improving performance. Furthermore, for each signal with negative slack, we need to speed it up only until we eliminate all its negative slack, so we have:

	Signals with negative slack	Per-processor cost to eliminate all negative slack
a.	MemRead (–20ps) RegWrite (–40ps)	60ps at $1/5ps = $12
b.	ALUOp (–35ps) ALUSrc (–80ps)	115ps at $1/5ps = $23

4.10.6 The signal with the most negative slack determines the new clock cycle time. The new clock cycle time increases the slack of all signals until there are is no remaining negative slack. To minimize cost, we can then slow down signals that end up having some (positive) slack. Overall, the cost is minimized by slowing signals down by:

	RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
a.	50ps	40ps	140ps	20ps	70ps	50ps	90ps	70ps	0ps
b.	80ps	80ps	180ps	180ps	280ps	45ps	280ps	0ps	80ps

Solution 4.11

4.11.1

	Sign-extend	Jump's shift-left-2
a.	00000000000000000000000000010000	0001000011000000000001000000
b.	00000000000000000000000000001100	0000100011000000000000110000

4.11.2

	ALUOp[1-0]	Instruction[5-0]
a.	00	010000
b.	01	001100

4.11.3

	New PC	Path
a.	PC + 4	PC to Add (PC + 4) to branch Mux to jump Mux to PC
b.	If $1 and $3 are not equal, PC + 4 If $1 and $3 are equal, PC + 4 + 4 × 12	PC to Add (PC + 4) to branch Mux, or PC to Add (PC + 4) to Add (adds offset) to branch Mux. After the branch Mux, we go through jump Mux and into the PC

4.11.4

	WrReg Mux	ALU Mux	Mem/ALU Mux	Branch Mux	Jump Mux
a.	3	16	0	PC + 4	PC + 4
b.	3 or 0 (RegDst is X)	–3	X	PC + 4	PC + 4

4.11.5

	ALU	Add (PC + 4)	Add (Branch)
a.	2 and 16	PC and 4	PC + 4 and 16 × 4
b.	–16 and –3	PC and 4	PC + 4 and 12 × 4

4.11.6

	Read Register 1	Read Register 2	Write Register	Write Data	RegWrite
a.	2	3	3	0	1
b.	1	3	X (3 or 0)	X	0

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 4.14

4.14.1 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

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 `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.

4.14.2 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 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

4.14.3 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	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

4.14.4 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

4.14.5

	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

4.14.6 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

Solution 4.15

4.15.1

a.	This instruction behaves like a load with a zero offset until it fetches the value from memory. The pre-ALU Mux must have another input now (zero) to allow this. After the value is read from memory in the MEM stage, it must be compared against zero. This must either be done quickly in the WB stage, or we must add another stage between MEM and WB. The result of this zero-comparison must then be used to control the branch Mux, delaying the selection signal for the branch Mux until the WB stage.
b.	We need to compute the memory address using two register values, so the address computation for SWI is the same as the value computation for the ADD instruction. However, now we need to read a third register value, so Registers must be extended to support a another read register input and another read data output and a Mux must be added in EX to select the Data Memory's write data input between this value and the value for the normal SW instruction.

4.15.2

a.	We need to add one more bit to the control signal for the pre-ALU Mux. We also need a control signal similar to the existing "Branch" signal to control whether or not the new zero-compare result is allowed to change the PC.
b.	We need a control signal to control the new Mux in the EX stage.

4.15.3

a.	This instruction introduces a new control hazard. The new PC for this branch is computed only after the Mem stage. If a new stage is added after MEM, this either adds new forwarding paths (from the new stage to EX) or (if there is no forwarding) makes a stall due to a data hazard one cycle longer.
b.	This instruction does not affect hazards. It modifies no registers, so it causes no data hazards. It is not a branch instruction, so it produces no control hazards. With the added third register read port, it creates no new resource hazards, either.

4.15.4

a.	`lw Rtmp,0(Rs)`
	`beq Rt,$0,Label`
		E.g., BEZI can be used when trying to find the length of a zero-terminated array.

b.	`add Rtmp,Rs,Rt`
	`sw Rd,0(Rtmp)`
		E.g., SWI can be used to store to an array element, where the array begins at address Rt and Rs is used as an index into the array.

4.15.5 The instruction can be translated into simple MIPS-like micro-operations (see 4.15.4 for a possible translation). These micro-operations can then be executed by the processor with a "normal" pipeline.

4.15.6 We will compute the execution time for every replacement interval. The old execution time is simply the number of instruction in the replacement interval (CPI of 1). The new execution time is the number of instructions after we made the replacement, plus the number of added stall cycles. The new number of instructions is the number of instructions in the original replacement interval, plus the new instruction, minus the number of instructions it replaces:

	New execution time	Old execution time	Speed-up
a.	20 – (2 – 1) + 1 = 20	20	1.00
b.	60 – (3 – 1) + 0 = 58	60	1.03

Solution 4.16

4.16.1 For every instruction, the IF/ID register keeps the PC + 4 and the instruction word itself. The ID/EX register keeps all control signals for the EX, MEM, and WB stages, PC + 4, the two values read from Registers, the sign-extended lowermost 16 bits of the instruction word, and Rd and Rt fields of the instruction word (even for instructions whose format does not use these fields). The EX/MEM register keeps control signals for MEM and WB stages, the PC + 4 + Offset (where Offset is the sign-extended lowermost 16 bits of the instructions, even for instructions that have no offset field), the ALU result and the value of its Zero output, the value that was read from the second register in the ID stage (even for instructions that never need this value), and the number of the destination register (even for instructions that need no register writes; for these instructions the number of the destination register is simply a "random" choice between Rd or Rt). The MEM/WB register keeps the WB control signals, the value read from memory (or a "random" value if there was no memory read), the ALU result, and the number of the destination register.

4.16.2

	Need to be read	Actually read
a.	$6	$6, $1
b.	$5	$5 (twice)

4.16.3

	EX	MEM
a.	40 + $6	Load value from memory
b.	$5 + $5	Nothing

4.16.4

	Loop	

a.	2:add $5,$5,$8
	2:add $6,$6,$8
	2:sw $1,20($5)
	2:beq $1,$0,Loop
	3:lw $1,40($6)
	3:add $5,$5,$8
	3:add $6,$6,$8
	3:sw $1,20($5)
	3:beq $1,$0,Loop
		WB
		MEM WB
		EX MEM WB
		ID EX MEM WB
		IF ID EX MEM WB
		IF ID EX MEM
		IF ID EX
		IF ID
		IF

b.	sw $0,0($1)
	sw $0,4($1)
	add $2,$2,$4
	beq $2,$0,Loop
	add $1,$2,$3
	sw $0,0($1)
	sw $0,4($1)
	add $2,$2,$4
	beq $2,$0,Loop
		WB
		MEM WB
		EX MEM WB
		ID EX MEM WB
		IF ID EX MEM WB
		IF ID EX MEM
		IF ID EX
		IF ID
		IF

4.16.5 In a particular clock cycle, a pipeline stage is not doing useful work if it is stalled or if the instruction going through that stage is not doing any useful work there. In the pipeline execution diagram from 4.16.4, a stage is stalled if its name is not shown for a particular cycles, and stages in which the particular instruction is not doing useful work are marked in red. Note that a BEQ instruction is doing useful work in the MEM stage, because it is determining the correct value of the next instruction's PC in that stage. We have:

	Cycles per loop iteration	Cycles in which all stages do useful work	% of cycles in which all stages do useful work
a.	5	1	20%
b.	5	2	40%

4.16.6 The address of that first instruction of the third iteration (PC + 4 for the `beq` from the previous iteration) and the instruction word of the `beq` from the previous iteration.

Solution 4.17

4.17.1 Of all these instructions, the value produced by this adder is actually used only by a `beq` instruction when the branch is taken. We have:

a.	15% (60% of 25%)
b.	9% (60% of 15%)

4.17.2 Of these instructions, only `add` needs all three register ports (reads two registers and write one). `beq` and `sw` does not write any register, and `lw` only uses one register value. We have:

a.	50%
b.	30%

4.17.3 Of these instructions, only `lw` and `sw` use the data memory. We have:

a.	25% (15% + 10%)
b.	55% (35% + 20%)

4.17.4 The clock cycle time of a single-cycle is the sum of all latencies for the logic of all five stages. The clock cycle time of a pipelined datapath is the maximum latency of the five stage logic latencies, plus the latency of a pipeline register that keeps the results of each stage for the next stage. We have:

	Single-cycle	Pipelined	Speed-up
a.	500ps	140ps	3.57
b.	730ps	230ps	3.17

4.17.5 The latency of the pipelined datapath is unchanged (the maximum stage latency does not change). The clock cycle time of the single-cycle datapath is the sum of logic latencies for the four stages (IF, ID, WB, and the combined EX + MEM stage). We have:

	Single-cycle	Pipelined
a.	410ps	140ps
b.	560ps	230ps

4.17.6 The clock cycle time of the two pipelines (5-stage and 4-stage) as explained for 4.17.5. The number of instructions increases for the 4-stage pipeline, so the speed-up is below 1 (there is a slowdown):

	Instructions with 5-stage	Instructions with 4-stage	Speed-up
a.	1.00 × I	1.00 × I + 0.5 × (0.15 + 0.10) × I = 1.125 × I	0.89
b.	1.00 × I	1.00 × I + 0.5 × (0.35 + 0.20) × I = 1.275 × I	0.78

Solution 4.18

4.18.1 No signals are asserted in IF and ID stages. For the remaining three stages we have:

	EX	MEM	WB
a.	ALUSrc = 0, ALUOp = 10, RegDst = 1	Branch = 0, MemWrite = 0, MemRead = 0	MemtoReg = 1, RegWrite = 1
b.	ALUSrc = 0, ALUOp = 10, RegDst = 1	Branch = 0, MemWrite = 0, MemRead = 0	MemtoReg = 1, RegWrite = 1

4.18.2 One clock cycle.

4.18.3 The PCSrc signal is 0 for this instruction. The reason against generating the PCSrc signal in the EX stage is that the `and` must be done after the ALU computes its Zero output. If the EX stage is the longest-latency stage and the ALU output is on its critical path, the additional latency of an AND gate would increase the clock cycle time of the processor. The reason in favor of generating this signal in the EX stage is that the correct next-PC for a conditional branch can be computed one cycle earlier, so we can avoid one stall cycle when we have a control hazard.

4.18.4

	Control signal 1	Control signal 2
a.	Generated in ID, used in EX	Generated in ID, used in WB
b.	Generated in ID, used in MEM	Generated in ID, used in WB

4.18.5

a.	R-type instructions
b.	Loads.

4.18.6 Signal 2 goes back though the pipeline. It affects execution of instructions that execute after the one for which the signal is generated, so it is not a time-travel paradox.

Solution 4.19

4.19.1 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
a.	1 + 0.45 × 2 + 0.05 × 1 = 1.95	49% (0.95/1.95)
b.	1 + 0.40 × 2 + 0.10 × 1 = 1.9	47% (0.9/1.9)

4.19.2 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
a.	1 + 0.25 = 1.25	20% (0.25/1.25)
b.	1 + 0.20 = 1.20	17% (0.20/1.20)

4.19.3 With forwarding only from the EX/MEM register, EX to 1st dependences can be satisfied without stalls but EX to 2nd and MEM to 1st dependences incur a one-cycle stall. With forwarding only from the MEM/WB register, EX to 2nd dependences incur no stalls. MEM to 1st dependences still incur a one-cycle stall (no time travel), and EX to 1st dependences now incur one stall cycle because we must wait for the instruction to complete the MEM stage to be able to forward to the next instruction. We compute stall cycles per instructions for each case as follows:

	EX/MEM	MEM/WB	Fewer stall cycles with
a.	0.10 + 0.05 + 0.25 = 0.40	0.10 + 0.10 + 0.25 = 0.45	EX/MEM
b.	0.05 + 0.10 + 0.20 = 0.35	0.15 + 0.05 + 0.20 = 0.40	EX/MEM

4.19.4 In 4.19.1 and 4.19.2 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	Speed-up
a.	1.95 × 100ps = 195ps	1.25 × 110ps = 137.5ps	1.42
b.	1.90 × 300ps = 570ps	1.20 × 350ps = 420ps	1.36

4.19.5 We already computed the time per instruction for full forwarding in 4.19.4. Now we compute time-per instruction with time-travel forwarding and the speed-up over full forwarding:

	With full forwarding	Time-travel forwarding	Speed-up
a.	1.25 × 110ps = 137.5ps	1 × 210ps = 210ps	0.65
b.	1.20 × 350ps = 420ps	1 × 450ps = 450ps	0.93

4.19.6

	EX/MEM	MEM/WB	Shorter time per instruction with
a.	1.40 × 100ps = 140ps	1.45 × 100ps = 145ps	EX/MEM
b.	1.35 × 320ps = 432ps	1.40 × 310ps = 434ps	EX/MEM

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)

4.20.5 A register modification becomes “visible” to the EX stage of the following instructions only two cycles after the instruction that produces the register value leaves the EX stage. Our forwarding-assuming hazard detection unit only adds a 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					Signals
		1	2	3	4	5	
a.	`lw $1,40($6)`	IF	ID	EX	MEM	WB	1: PCWrite = 1, ALUin1 = X, ALUin2 = X
	`add $2,$3,$1`		IF	ID	***	EX	2: PCWrite = 1, ALUin1 = X, ALUin2 = X
	`add $1,$6,$4`			IF	***	ID	3: PCWrite = 1, ALUin1 = 0, ALUin2 = 0
	`sw $2,20($4)`					IF	4: PCWrite = 0, ALUin1 = X, ALUin2 = X
	`and $1,$1,$4`						5: PCWrite = 1, ALUin1 = 0, ALUin2 = 2
b.	`add $1,$5,$3`	IF	ID	EX	MEM	WB	1: PCWrite = 1, ALUin1 = X, ALUin2 = X
	`sw $1,0($2)`		IF	ID	EX	MEM	2: PCWrite = 1, ALUin1 = X, ALUin2 = X
	`lw $1,4($2)`			IF	ID	EX	3: PCWrite = 1, ALUin1 = 0, ALUin2 = 0
	`add $5,$5,$1`				IF	ID	4: PCWrite = 1, ALUin1 = 0, ALUin2 = 1
	`sw $1,0($2)`					IF	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:

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

Solution 4.22

4.22.1

		Pipeline Cycles													
	Executed Instructions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
a.	`lw  $1,40($6)`	IF	ID	EX	MEM	WB									
	`beq $2,$3,Label2 (T)`		IF	ID	EX	MEM	WB								
	`beq $1,$2,Label1 (NT)`			IF	ID	EX	MEM	WB							
	`sw  $2,20($4)`						IF	ID	EX	MEM	WB				
	`and $1,$1,$4`							IF	ID	EX	MEM	WB			
b.	`add $1,$5,$3`	IF	ID	EX	MEM	WB									
	`sw  $1,0($2)`		IF	ID	EX	MEM	WB								
	`add $2,$2,$3`			IF	ID	EX	MEM	WB							
	`beq $2,$4,Label1 (NT)`				IF	ID	EX	MEM	WB						
	`add $5,$5,$1`							IF	ID	EX	MEM	WB			
	`sw  $1,0($2)`								IF	ID	EX	MEM	WB		

4.22.2

	Executed Instructions	Pipeline Cycles													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
a.	`lw  $1,40($6)`	IF	ID	EX	MEM	WB									
	`beq $2,$3,Label2 (T)`		IF	ID	EX	MEM	WB								
	`add $1,$6,$4`			IF	ID	EX	MEM	WB							
	`beq $1,$2,Label1 (NT)`				IF	ID	***	EX	MEM	WB					
	`sw  $2,20($4)`					IF	***	ID	EX	MEM	WB				
	`and $1,$1,$4`								IF	ID	EX	MEM	WB		
b.	`add $1,$5,$3`	IF	ID	EX	MEM	WB									
	`sw  $1,0($2)`		IF	ID	EX	MEM	WB								
	`add $2,$2,$3`			IF	ID	EX	MEM	WB							
	`beq $2,$4,Label1 (NT)`				IF	ID	EX	MEM	WB						
	`add $5,$5,$1`					IF	ID	EX	MEM	WB					
	`sw  $1,0($2)`							IF	ID	EX	MEM	WB			

4.22.3

a.

```
Label1: lw  $1,40($6)
        seq $8,$2,$3
        bnez $8,Label2 ; Taken
        add $1,$6,$4
Label2: seq $8,$1,$2
        bnez $8,Label1 ; Not taken
        sw  $2,20($4)
        and $1,$1,$4
```

b.

```
        add $1,$5,$3
Label1: sw  $1,0($2)
        add $2,$2,$3
        bez $8,$2,$4
        bnez $8,Label1 ; Not taken
        add $5,$5,$1
        sw  $1,0($2)
```

4.22.4 The hazard detection logic must detect situations when the branch depends on the result of the previous R-type instruction, or on the result of two previous loads. When the branch uses the values of its register operands in its ID stage, the R-type instruction's result is still being generated in the EX stage. Thus we must stall the processor and repeat the ID stage of the branch in the next cycle. Similarly, if the branch depends on a load that immediately precedes it, the result of the load is only generated two cycles after the branch enters the ID stage, so we must stall the branch for two cycles. Finally, if the branch depends on a load that is the second-previous instruction, the load is completing its MEM stage when the branch is in its ID stage, so we must stall the branch for one cycle. In all three cases, the hazard is a data hazard.

Note that in all three cases we assume that the values of preceding instructions are forwarded to the ID stage of the branch if possible.

4.22.5 For 4.22.1 we have already shows the pipeline execution diagram for the case when branches are executed in the EX stage. The following is the pipeline diagram when branches are executed in the ID stage, including new stalls due to data dependences described for 4.22.4:

		Pipeline Cycles													
	Executed Instructions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
a.	`lw  $1,40($6)`	IF	ID	EX	MEM	WB									
	`beq $2,$3,Label2 (T)`		IF	ID	EX	MEM	WB								
	`beq $1,$2,Label1 (NT)`			IF	***	ID	EX	MEM	WB						
	`sw  $2,20($4)`					IF	ID	EX	MEM	WB					
	`and $1,$1,$4`						IF	ID	EX	MEM	WB				
b.	`add $1,$5,$3`	IF	ID	EX	MEM	WB									
	`sw  $1,0($2)`		IF	ID	EX	MEM	WB								
	`add $2,$2,$3`			IF	ID	EX	MEM	WB							
	`beq $2,$4,Label1 (NT)`				IF	***	ID	EX	MEM	WB					
	`add $5,$5,$1`							IF	ID	EX	MEM	WB			
	`sw  $1,0($2)`								IF	ID	EX	MEM	WB		

Now the speed-up can be computed as:

a.	11/10 = 1.1
b.	12/12 = 1

4.22.6 Branch instructions are now executed in the ID stage. If the branch instruction is using a register value produced by the immediately preceding instruction, as we described for 4.22.4 the branch must be stalled because the preceding instruction is in the EX stage when the branch is already using the stale register values in the ID stage. If the branch in the ID stage depends on an R-type instruction that is in the MEM stage, we need forwarding to ensure correct execution of the branch. Similarly, if the branch in the ID stage depends on an R-type of load instruction in the WB stage, we need forwarding to ensure correct execution of the branch. Overall, we need another forwarding unit that takes the same inputs as the one that forwards to the EX stage. The new forwarding unit should control two Muxes placed right before the branch comparator. Each Mux selects between the value read from Registers, the ALU output from the EX/MEM pipeline register, and the data value from the MEM/WB pipeline register. The complexity of the new forwarding unit is the same as the complexity of the existing one.

Solution 4.23

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

	Extra CPI
a.	3 × (1 – 0.40) × 0.15 = 0.27
b.	3 × (1 – 0.60) × 0.10 = 0.12

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

	Extra CPI
a.	3 × (1 – 0.60) × 0.15 = 0.18
b.	3 × (1 – 0.40) × 0.10 = 0.18

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

	Extra CPI
a.	3 × (1 – 0.80) × 0.15 = 0.090
b.	3 × (1 – 0.95) × 0.10 = 0.015

4.23.4 Correctly predicted branches had CPI of 1 and now they become ALU instructions whose CPI is also 1. Incorrectly predicted instructions that are converted also become ALU instructions with a CPI of 1, so we have:

	CPI without conversion	CPI with conversion	Speed-up from conversion
a.	1 + 3 × (1 – 0.80) × 0.15 = 1.090	1 + 3 × (1 – 0.80) × 0.15 × 0.5 = 1.045	1.090/1.045 = 1.043
b.	1 + 3 × (1 – 0.95) × 0.10 = 1.015	1 + 3 × (1 – 0.95) × 0.10 × 0.5 = 1.008	1.015/1.008 = 1.007

4.23.5 Every converted branch instruction now takes an extra cycle to execute, so we have:

	CPI without conversion	Cycles per original instruction with conversion	Speed-up from conversion
a.	1.090	1 + (1 + 3 × (1 – 0.80)) × 0.15 × 0.5 = 1.120	1.090/1.120 = 0.97
b.	1.015	1 + (1 + 3 × (1 – 0.95)) × 0.10 × 0.5 = 1.058	1.015/1.058 = 0.96

4.23.6 Let the total number of branch instructions executed in the program be B. Then we have:

	Correctly predicted	Correctly predicted non-loop-back	Accuracy on non-loop-back branches
a.	B × 0.80	B × 0.00	(B × 0.00)/(B × 0.20) = 0.00 (00%)
b.	B × 0.95	B × 0.15	(B × 0.15)/(B × 0.20) = 0.75 (75%)

Solution 4.24

4.24.1

	Always-taken	Always not-taken
a.	3/4 = 75%	1/4 = 25%
b.	3/5 = 60%	2/5 = 40%

4.24.2

	Outcomes	Predictor value at time of prediction	Correct or Incorrect	Accuracy
a.	T, T, NT, T	0, 1, 2, 1	I, I, I, I	0%
b.	T, T, T, NT	0, 1, 2, 3	I, I, C, I	25%

4.24.3 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
a.	T, T, NT, T	1st occurrence: 0, 1, 2, 1 2nd occurrence: 2, 3, 2, 3 3rd occurrence: 3, 3, 3, 2 4th occurrence: 3, 3, 3, 2	C, C, I, C	75%
b.	T, T, T, NT, NT	1st occurrence: 0, 1, 2, 3, 2 2nd occurrence: 1, 2, 3, 3, 2 3rd occurrence: 1, 2, 3, 3, 2	C, C, C, I, I	60%

4.24.4 The predictor should be an N-bit shift register, where N is the number of branch outcomes in the target pattern. The shift register should be initialized with the pattern itself (0 for NT, 1 for T), and the prediction is always the value in the leftmost bit of the shift register. The register should be shifted after each predicted branch.

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

4.24.6 The predictor is the same as in 4.24.4, 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.

Solution 4.25

4.25.1

	Instruction 1	Instruction 2
a.	Overflow (EX)	Invalid target address (EX)
b.	Invalid data address (MEM)	No exceptions

4.25.2 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.

4.25.3 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.

4.25.4

	Handler address
a.	0xFFFFF000
b.	0x00000010

The first instruction word from the handler address is fetched in the cycle after the one in which the original exception is detected. When this instruction is decoded in the next cycle, the processor detects that the instruction is invalid. This exception is treated just like a normal exception—it converts the instruction being fetched in that cycle into a `nop` and puts the address of the Invalid Instruction handler into the PC at the end of the cycle in which the Invalid Instruction exception is detected.

4.25.5 This approach requires us to fetch the address of the handler from memory. We must add the code of the exception to the address of the exception vector table, read the handler's address from memory, and jump to that address. One way of doing this is to handle it like a special instruction that computer the address in EX, loads the handler's address in MEM, and sets the PC in WB.

4.25.6 We need a special instruction that allows us to move a value from the (exception) Cause register to a general-purpose register. We must first save the general-purpose register (so we can restore it later), load the Cause register into it, add the address of the vector table to it, use the result as an address for a load that gets the address of the right exception handler from memory, and finally jump to that handler.

Solution 4.26

4.26.1 All exception-related signals are 0 in all stages, except the one in which the exception is detected. For that stage, we show values of Flush signals for various stages, and also the value of the signal that controls the Mux that supplies the PC value.

	Stage	Signals
a.	EX	IF.Flush = ID.Flush = EX.Flush = 1, PCSel = Exc
b.	MEM	IF.Flush = ID.Flush = EX.Flush = MEM.Flush = 1, PCSel = Exc This exception is detected in MEM, so we added MEM.Flush

4.26.2 The signals stored in the ID/EX stage are needed to execute the instruction if there are no exceptions. Figure 4.66 does not show it, but exception conditions from various stages are also supplied as inputs to the Control unit. The signal that goes directly to EX is EX.Flush and it is based on these exception condition inputs, not on the opcode of the instruction that is in the ID stage. In particular, the EX.Flush signal becomes 1 when the instruction in the EX stage triggers an exception and must be prevented from completing.

4.26.3 The disadvantage is that the exception handler begins executing one cycle later. Also, an exception condition normally checked in MEM cannot be delayed into WB, because at that time the instruction is updating registers and cannot be prevented from doing so.

4.26.4 When overflow is detected in EX, each exception results in a 3-cycle delay (IF, ID, and EX are cancelled). By moving overflow into MEM, we add one more cycle to this delay. To compute the speed-up, we compute execution time per 100,000 instructions:

	Old clock cycle time	New clock cycle time	Old time per 100,000 instructions	New time per 100,000 instructions	Speed-up
a.	350ps	350ps	350ps × 100,003	350ps × 100,004	0.99999
b.	210ps	210ps	210ps × 100,003	210ps × 100,004	0.99999

4.26.5 Exception control (Flush) signals are not really generated in the EX stage. They are generated by the Control unit, which is drawn as part of the ID stage, but we could have a separate "Exception Control" unit to generate Flush signals and this unit is not really a part of any stage.

4.26.6 Flush signals must be generated one Mux time before the end of the cycle. However, their generation can only begin after exception conditions are generated. For example, arithmetic overflow is only generated after the ALU operation in EX is complete, which is usually in the later part of the clock cycle. As a result, the Control unit actually has very little time to generate these signals, and they can easily be on the critical path that determines the clock cycle time.

Solution 4.27

4.27.1 When the invalid instruction (I3) is decoded, IF.Flush and ID.Flush signals are used to convert I3 and I4 into `nops` (marked with *). In the next cycle, in IF we fetch the first instruction of the exception handler, in ID we have a `nop` (instead of I4, marked), in EX we have a `nop` (instead of I3), and I1 and I2 still continue through the pipeline normally:

	Branch and delay slot	Pipeline

a.	`I1: beq  $1,$0,Label`
	`I2: sw   $6,50($1)`
	`I3: Invalid`
	`I4: Something`
	`I5: Handler`
		`IF   ID   EX   MEM  WB`
		`     IF   ID   EX   MEM`
		`          IF   ID   *EX`
		`               IF   *ID`
		`                    IF`

b.	`I1: beq  $5,$0,Label`
	`I2: nor $5,$4,$3`
	`I3: Invalid`
	`I4: Something`
	`I5: Handler`
		`IF   ID   EX   MEM  WB`
		`     IF   ID   EX   MEM`
		`          IF   ID   *EX`
		`               IF   *ID`
		`                    IF`

4.27.2 When I2 is in the MEM stage, it triggers an exception condition that results in converting I2 and I5 into `nops` (I3 and I4 are already `nops` by then). In the next cycle, we fetch I6, which is the first instruction of the exception handler for the exception triggered by I2.

	Branch and delay slot	Branch and delay slot

a.	`I1: beq  $1,$0,Label`
	`I2: sw   $6,50($1)`
	`I3: Invalid`
	`I4: Something`
	`I5: Handler 1`
	`I6: Handler 2`
		`IF  ID  EX  MEM WB`
		`    IF  ID  EX  MEM *WB`
		`        IF  ID  *EX *ME`
		`            IF  *ID *EX`
		`                IF  *ID`
		`                     IF`

b.	`I1: beq  $5,$0,Label`
	`I2: nor $5,$4,$3`
	`I3: Invalid`
	`I4: Something`
	`I5: Handler 1`
	`I6: Handler 2`
		`IF  ID  EX  MEM WB`
		`    IF  ID  EX  MEM *WB`
		`        IF  ID  *EX *ME`
		`            IF  *ID *EX`
		`                IF  *ID`
		`                     IF`

4.27.3 The EPC is the PC + 4 of the delay slot instruction. As described in Section 4.9, the exception handler subtracts 4 from the EPC, so it gets the address of the instruction that generated the exception (I2, the delay slot instruction). If the exception handler decides to resume execution of the application, it will jump to the I2. Unfortunately, this causes the program to continue as if the branch was not taken, even if it was taken.

4.27.4 The processor cancels the store instruction and other instructions (from the "Invalid instruction" exception handler) fetched after it, and then begins fetching instructions from the invalid data address handler. A major problem here is that the new exception sets the EPC to the instruction address in the "Invalid instruction" handler, overwriting the EPC value that was already there (address for continuing the program). If the invalid data address handler repairs the problem and attempts to continue the program, the "Invalid instruction" handler will be executed. However, if it manages to repair the problem and wants to continue the program, the EPC it incorrect (it was overwritten before it could be saved). This is the reason why exception handlers must be written carefully to avoid triggering exceptions themselves, at least until they have safely saved the EPC.

4.27.5 Not for store instructions. If we check for the address overflow in MEM, the store is already writing data to memory in that cycle and we can no longer "cancel" it. As a result, when the exception handler is called the memory is already changed by the store instruction, and the handler can not observe the state of the machine that existed before the store instruction.

4.27.6 We must add two comparators to the EX stage, one that compares the ALU result to WADDR, and another that compares the data value from Rt to WVAL. If one of these comparators detects equality and the instruction is a store, this triggers a "Watchpoint" exception. As discussed for 4.27.5, we cannot delay the comparisons until the MEM stage because at that time the store is already done and we need to stop the application at the point before the store happens.

Solution 4.28

4.28.1

a.

```
       add  $1,$0,$0
Again: beq  $1,$2,End
       add  $6,$3,$1
       lw   $7,0($6)
       add  $8,$4,$1
       sw   $7,0($8)
       addi $1,$1,1
       beq  $0,$0,Again
End:
```

b.

```
       add  $4,$0,$0
Again: add  $1,$4,$6
       lw   $2,0($1)
       lw   $3,1($1)
       beq  $2,$3,End
       sw   $0,0($1)
       addi $4,$4,1
       beq  $0,$0,Again
End:
```

4.28.2

	Instructions	Pipeline																							
a.	add $1,$0,$0	IF	ID	EX	ME	WB																			
	beq $1,$2,End	IF	ID	**	EX	ME	WB																		
	add $6,$3,$1		IF	**	ID	EX	ME	WB																	
	lw $7,0($6)		IF	**	ID	**	EX	ME	WB																
	add $8,$4,$1				IF	**	ID	EX	ME	WB															
	sw $7,0($8)				IF	**	ID	**	EX	ME	WB														
	addi $1,$1,1						IF	**	ID	EX	ME	WB													
	beq $0,$0,Again						IF	**	ID	**	EX	ME	WB												
	beq $1,$2,End								IF	**	ID	EX	ME	WB											
	add $6,$3,$1								IF	**	ID	**	EX	ME	WB										
	lw $7,0($6)										IF	**	ID	EX	ME	WB									
	add $8,$4,$1										IF	**	ID	EX	ME	WB									
	sw $7,0($8)												IF	ID	EX	ME	WB								
	addi $1,$1,1												IF	ID	EX	ME	WB								
	beq $0,$0,Again													IF	ID	EX	ME	WB							
	beq $1,$2,End													IF	ID	**	EX	ME	WB						
b.	add $4,$0,$0	IF	ID	EX	ME	WB																			
	add $1,$4,$6	IF	ID	**	EX	ME	WB																		
	lw $2,0($1)		IF	**	ID	EX	ME	WB																	
	lw $3,1($1)		IF	**	ID	**	EX	ME	WB																
	beq $2,$3,End				IF	**	ID	**	EX	ME	WB														
	sw $0,0($1)				IF	**	ID	**	EX	ME	WB														
	addi $4,$4,1						IF	**	ID	EX	ME	WB													
	bew $0,$0,Again						IF	**	ID	**	EX	ME	WB												
	add $1,$4,$6								IF	**	ID	EX	ME	WB											
	lw $2,0($1)								IF	**	ID	**	EX	ME	WB										
	lw $3,1($1)										IF	**	ID	EX	ME	WB									
	beq $2,$3,End										IF	**	ID	**	**	EX	ME	WB							
	sw $0,0($1)												IF	**	**	ID	EX	ME	WB						
	addi $4,$4,1												IF	**	**	ID	EX	ME	WB						
	bew $0,$0,Again															IF	ID	EX	ME	WB					
	add $1,$4,$6															IF	ID	**	EX	ME	WB				
	lw $2,0($1)																IF	**	ID	EX	ME	WB			
	lw $3,1($1)																IF	**	ID	**	EX	ME	WB		
	beq $2,$3,End																		IF	**	ID	**	EX	ME	WB

4.28.3 The only way to execute 2 instructions fully in parallel is for a load/store to execute together with another instruction. To achieve this, around each load/store instruction we will try to put non-load/store instructions that have no dependences with the load/store.

a.	`       add  $1,$0,$0`
	`Again: beq  $1,$2,End`
	`       add  $6,$3,$1`
	`       add  $8,$4,$1`
	`       lw   $7,0($6)`
	`       addi $1,$1,1`
	`       sw   $7,0($8)`
	`       beq  $0,$0,Again`
	`End:`	

b.	`       add  $4,$0,$0`
	`Again: add  $1,$4,$6`
	`       lw   $2,0($1)`
	`       lw   $3,1($1)`
	`       beq  $2,$3,End`
	`       sw   $0,0($1)`
	`       addi $4,$4,1`
	`       beq  $0,$0,Again`
	`End:`
		We have not changed anything. Note that the only instruction without dependences to or from the two loads is ADDI, and it cannot be moved above the branch (then the loop would exit with the wrong value for i).

4.28.4

	Instructions	Pipeline
a.	`add  $1,$0,$0`	`IF ID EX ME WB`
	`beq  $1,$2,End`	`IF ID ** EX ME WB`
	`add  $6,$3,$1`	`   IF ** ID EX ME WB`
	`add  $8,$4,$1`	`   IF ** ID ** EX ME WB`
	`lw   $7,0($6)`	`         IF ** ID EX ME WB`
	`addi $1,$1,1`	`         IF ** ID EX ME WB`
	`sw   $7,0($8)`	`               IF ID EX ME WB`
	`beq  $0,$0,Again`	`               IF ID EX ME WB`
	`beq  $1,$2,End`	`                  IF ID EX ME WB`
	`add  $6,$3,$1`	`                  IF ID ** EX ME WB`
	`add  $8,$4,$1`	`                     IF ** ID EX ME WB`
	`lw   $7,0($6)`	`                     IF ** ID EX ME WB`
	`addi $1,$1,1`	`                           IF ID EX ME WB`
	`sw   $7,0($8)`	`                           IF ID EX ME WB`
	`beq  $0,$0,Again`	`                              IF ID EX ME WB`
	`beq  $1,$2,End`	`                              IF ID ** EX ME WB`
b.	`add  $4,$0,$0`	`IF ID EX ME WB`
	`add  $1,$4,$6`	`IF ID ** EX ME WB`
	`lw   $2,0($1)`	`   IF ** ID EX ME WB`
	`lw   $3,1($1)`	`   IF ** ID ** EX ME WB`
	`beq  $2,$3,End`	`         IF ** ID ** EX ME WB`
	`sw   $0,0($1)`	`         IF ** ID ** EX ME WB`
	`addi $4,$4,1`	`               IF ** ID EX ME WB`
	`bew  $0,$0,Again`	`               IF ** ID ** EX ME WB`
	`add  $1,$4,$6`	`                     IF ** ID EX ME WB`
	`lw   $2,0($1)`	`                     IF ** ID ** EX ME WB`
	`lw   $3,1($1)`	`                           IF ** ID EX ME WB`
	`beq  $2,$3,End`	`                           IF ** ID ** ** EX ME WB`
	`sw   $0,0($1)`	`                                 IF ** ** ID EX ME WB`
	`addi $4,$4,1`	`                                 IF ** ** ID EX ME WB`
	`bew  $0,$0,Again`	`                                          IF ID EX ME WB`
	`add  $1,$4,$6`	`                                          IF ID ** EX ME WB`
	`lw   $2,0($1)`	`                                             IF ** ID EX ME WB`
	`lw   $3,1($1)`	`                                             IF ** ID ** EX ME WB`
	`beq  $2,$3,End`	`                                                   IF ** ID ** EX ME WB`

4.28.5

	CPI for 1-issue	CPI for 2-issue	Speed-up
a.	1 (no data hazards)	0.86 (12 cycles for 14 instructions). In even-numbered iterations the LW and the SW can execute in parallel with the next instruction.	1.16
b.	1.14 (8 cycles per 7 instructions). There is 1 stall cycle in each iteration due to a data hazard between LW and the next instruction (BEQ).	1 (14 cycles for 14 instruction). Neither LW instruction can execute in parallel with another instruction, and the BEQ after the second LW is stalled because it depends on the load. However, SW always executes in parallel with another instruction (alternating between BEQ and ADDI).	1.14

4.28.6

	CPI for 1-issue	CPI for 2-issue	Speed-up
a.	1	0.64 (9 cycles for 14 instructions). In odd-numbered iterations ADD and LW cannot execute in the same cycle because of a data dependence, and then ADD and SW have the same problem. The rest of the instructions can execute in pairs.	1.56
b.	1.14	0.86 (12 cycles for 14 instructions). In all iterations BEQ is stalled because it depends on the second LW. In odd-numbered BEQ and SW execute together, and so do ADDI and the last BEQ. In even-numbered iterations SW and ADDI execute together, and so do the last BEQ and the first ADD of the next iteration.	1.33

Solution 4.29

4.29.1 Note that all register read ports are active in every cycle, so 4 register reads (2 instructions with 2 reads each) happen in every cycle. We determine the number of cycles it takes to execute an iteration of the loop and the number of useful reads, then divide the two. The number of useful register reads for an instruction is the number of source register parameters minus the number of registers that are forwarded from prior instructions. We assume that register writes happen in the first half of the cycle and the register reads happen in the second half.

	Loop	Pipeline stages	Useful reads	% Useful

a.	addi $5,$5,-4
	beq $5,$0,Loop
	lw $1,40($6)
	add $5,$5,$1
	sw $1,20($5)
	addi $6,$6,4
	addi $5,$5,-4
	beq $5,$0,Loop
		ID EX ME WB
		ID ** EX ME WB
		IF ** ID EX ME WB
		IF ** ID ** ** EX ME WB
		      IF ** ** ID EX ME WB
		      IF ** ** ID EX ME WB
		               IF ID EX ME WB
		               IF ID ** EX ME WB
			
			1
			0 ($1, $5 fw)
			1 ($5 fw)
			1
			0 ($5 fw)
			1 ($5 fw)
				17%
				(4/(6 × 4))

b.	addi $2,$2,4
	beq $2,$0,Loop
	add $1,$2,$3
	sw $0,0($1)
	addi $2,$2,4
	beq $2,$0,Loop
		ID EX ME WB
		ID ** EX ME WB
		IF ** ID EX ME WB
		IF ** ID ** EX ME WB
		      IF ** ID EX ME WB
		      IF ** ID ** EX ME WB
			
			1 ($2 fw)
			1 ($1 fw)
			1
			1 ($2 fw)
				25%
				(4/(4 × 4))

4.29.2 The utilization of read ports is lower with a wider-issue processor:

	Loop	Pipeline stages	Useful reads	% Useful

a.	addi $6,$6,4
	addi $5,$5,-4
	beq $5,$0,Loop
	lw $1,40($6)
	add $5,$5,$1
	sw $1,20($5)
	addi $6,$6,4
	addi $5,$5,-4
	beq $5,$0,Loop
		ID EX ME WB
		ID EX ME WB
		ID ** EX ME WB
		IF ** ID EX ME WB
		IF ** ID ** ** EX ME WB
		IF ** ID ** ** ** EX ME WB
		      IF ** ** ** ID EX ME WB
		      IF ** ** ** ID EX ME WB
		      IF ** ** ** ID ** EX ME WB
			
			0 ($6 fw)
			0 ($1, $5 fw)
			0 ($1, $5 fw)
			1
			0 ($5 fw)
			1 ($5 fw)
				5.6%
				(2/(6 × 6))

b.	sw $0,0($1)
	addi $2,$2,4
	beq $2,$0,Loop
	add $1,$2,$3
	sw $0,0($1)
	addi $2,$2,4
	beq $2,$0,Loop
	add $1,$2,$3
	sw $0,0($1)
	addi $2,$2,4
	beq $2,$0,Loop
	add $1,$2,$3
	sw $0,0($1)
	addi $2,$2,4
	beq $2,$0,Loop
		ID EX ME WB
		ID EX ME WB
		ID ** EX ME WB
		IF ** ID EX ME WB
		IF ** ID ** EX ME WB
		IF ** ID ** EX ME WB
		      IF ** ID EX ME WB
		      IF ** ID EX ME WB
		      IF ** ID ** EX ME WB
		            IF ** ID EX ME WB
		            IF ** ID ** EX ME WB
		            IF ** ID ** EX ME WB
		                  IF ** ID EX ME WB
		                  IF ** ID EX ME WB
		                  IF ** ID ** EX ME WB
			
			1 ($2 fw)
			1 ($1 fw)
			0 ($2 fw)
			1 ($2 fw)
			1 ($2 fw)
			1 ($1 fw)
			1
			1 ($2 fw)
			1 ($2 fw)
			1 ($1 fw)
			0 ($2 fw)
			1 ($2 fw)
				21%
				(10/(8 × 6))

4.29.3

	2 ports used	3 ports used
a.	1 cycle out of 6 (16.7%)	Never (0%)
b.	4 cycles out of 8 (50%)	Never (0%)

4.29.4

	Unrolled and scheduled loop	Comment

a.	`Loop: lw $10,40($6)`
	`lw $1,44($6)`
	`addi $5,$5,-8`
	`addi $6,$6,8`
	`add $11,$5,$10`
	`add $5,$11,$1`
	`sw $10,28($11)`
	`sw $1,24($5)`
	`beq $5,$0,Loop`
		The only time this code is unable to execute two instructions in the same cycle is in even-numbered iterations of the unrolled loop when the two ADD instruction are fetched together but must execute in different cycles.

b.	`Loop: add $1,$2,$3`
	`addi $2,$2,8`
	`sw $0,-8($1)`
	`sw $0,-4($1)`
	`beq $2,$0,Loop`
		We are able to execute two instructions per cycle in every iteration of this loop, so we execute two iterations of the unrolled loop every 5 cycles.

4.29.5 We determine the number of cycles needed to execute two iterations of the original loop (one iteration of the unrolled loop). Note that we cannot use CPI in our speed-up computation because the two versions of the loop do not execute the same instructions.

	Original loop	Unrolled loop	Speed-up
a.	6 × 2 = 12	5	2.4
b.	4 × 2 = 8	2.5 (5/2)	3.2

4.29.6 On a pipelined processor the number of cycles per iteration is easily computed by adding together the number of instructions and the number of stalls. The only stalls occur when a `lw` instruction is followed immediately with a RAW-dependent instruction, so we have:

	Original loop	Unrolled loop	Speed-up
a.	(6 + 1) × 2 = 14	9	1.6
b.	4 × 2 = 8	5	1.6

Solution 4.30

4.30.1 Let p be the probability of having a mispredicted branch. Whenever we have an incorrectly predicted `beq` as the first of the two instructions in a cycle (the probability of this event is p), we waste one issue slot (half a cycle) and another two entire cycles. If the first instruction in a cycle is not a mispredicted `beq` but the second one is (the probability of this is (1 – p) × p), we waste two cycles. Without these mispredictions, we would be able to execute 2 instructions per cycle. We have:

	CPI
a.	0.5 + 0.02 × 2.5 + 0.98 × 0.02 × 2 = 0.589
b.	0.5 + 0.05 × 2.5 + 0.95 × 0.05 × 2 = 0.720

4.30.2 Inability to predict a branch results in the same penalty as a mispredicted branch. We compute the CPI like in 4.30.1, but this time we also have a 2-cycle penalty if we have a correctly predicted branch in the first issue slot and another branch that would be correctly predicted in the second slot. We have:

	CPI with 2 predicted branches per cycle	CPI with 1 predicted branch per cycle	Speed-up
a.	0.589	0.5 + 0.02 × 2.5 + 0.98 × 0.02 × 2 + 0.18 × 0.18 × 2 = 0.654	1.11
b.	0.720	0.5 + 0.05 × 2.5 + 0.95 × 0.05 × 2 + 0.10 × 0.10 × 2 = 0.740	1.03

4.30.3 We have a one-cycle penalty whenever we have a cycle with two instructions that both need a register write. Such instructions are ALU and `lw` instructions. Note that `beq` does not write registers, so stalls due to register writes and due to branch mispredictions are independent events. We have:

	CPI with 2 register writes per cycle	CPI with 1 register write per cycle	Speed-up
a.	0.589	0.5 + 0.02 × 2.5 + 0.98 × 0.02 × 2 + 0.70 × 0.70 × 1 = 1.079	1.83
b.	0.720	0.5 + 0.05 × 2.5 + 0.95 × 0.05 × 2 + 0.75 × 0.75 × 1 = 1.283	1.78

4.30.4 We have already computed the CPI with the given branch prediction accuracy, and we know that the CPI with ideal branch prediction is 0.5, so:

	CPI with given branch prediction	CPI with perfect branch prediction	Speed-up
a.	0.589	0.5	1.18
b.	0.720	0.5	1.44

4.30.5 The CPI with perfect branch prediction is now 0.25 (four instructions per cycle). A branch misprediction in the first issue slot of a cycle results in 2.75 penalty cycles (remaining issue slots in the same cycle plus 2 entire cycles), in the second issue slot 2.5 penalty cycles, in the third slot 2.25 penalty cycles, and in the last (fourth) slot 2 penalty cycles. We have:

	CPI with given branch prediction	CPI with perfect branch prediction	Speed-up
a.	$0.25 + 0.02 \times 2.75 + 0.98 \times 0.02 \times 2.5 + 0.98^2 \times 0.02 \times 2.25 + 0.98^3 \times 0.02 \times 2 = 0.435$	0.25	1.74
b.	$0.25 + 0.05 \times 2.75 + 0.95 \times 0.05 \times 2.5 + 0.95^2 \times 0.05 \times 2.25 + 0.95^3 \times 0.05 \times 2 = 0.694$	0.25	2.77

The speed-up from improved branch prediction is much larger in a 4-issue processor than in a 2-issue processor. In general, processors that issue more instructions per cycle gain more from improved branch prediction because each branch misprediction costs them more instruction execution opportunities (e.g., 4 per cycle in 4-issue versus 2 per cycle in 2-issue).

4.30.6 With this pipeline, the penalty for a mispredicted branch is 20 cycles plus the fraction of a cycle due to discarding instructions that follow the branch in the same cycle. We have:

	CPI with given branch prediction	CPI with perfect branch prediction	Speed-up
a.	$0.25 + 0.02 \times 20.75 + 0.98 \times 0.02 \times 20.5 + 0.98^2 \times 0.02 \times 20.25 + 0.98^3 \times 0.02 \times 20 = 1.832$	0.25	7.33
b.	$0.25 + 0.05 \times 20.75 + 0.95 \times 0.05 \times 20.5 + 0.95^2 \times 0.05 \times 20.25 + 0.95^3 \times 0.05 \times 20 = 4.032$	0.25	16.13

We observe huge speed-ups when branch prediction is improved in a processor with a very deep pipeline. In general, processors with deeper pipelines benefit more from improved branch prediction because these processors cancel more instructions (e.g., 20 stages worth of instructions in a 50-stage pipeline versus 2 stages worth of instructions in a 5-stage pipeline) on each misprediction.

Solution 4.31

4.31.1 The number of cycles is equal to the number of instructions (one instruction is executed per cycle) plus one additional cycle for each data hazard which occurs when a `lw` instruction is immediately followed by a dependent instruction. We have:

	CPI
a.	$(8 + 1)/8 = 1.13$
b.	$(7 + 1)/7 = 1.14$

4.31.2 The number of cycles is equal to the number of instructions (one instruction is executed per cycle), plus the stall cycles due to data hazards. Data hazards occur when the memory address used by the instruction depends on the result of a previous instruction (EXE to ARD, 2 stall cycles) or the instruction after that (1 stall cycle), or when an instruction writes a value to memory and one of the next two instructions reads a value from the same address (2 or 1 stall cycles). All other data dependences can use forwarding to avoid stalls. We have:

	Instructions	Stall Cycles	CPI
a.	I1: mov -4(esp), eax I2: add (edx), eax I3: mov eax, -4(esp) I4: add 1, ecx I5: add 4, edx I6: cmp esi, ecx I7: jl Label	No stalls.	7/7 = 1
b.	I1: add eax, (edx) I2: mov eax, edx I3: add 1, eax I4: jl Label	No stalls.	4/4 = 1

4.31.3 The number of instructions here is that from the x86 code, but the number of cycles per iteration is that from the MIPS code (we fetch x86 instructions, but after instructions are decoded we end up executing the MIPS version of the loop):

	CPI
a.	9/7 = 1.29
b.	8/4 = 2

4.31.4 Dynamic scheduling allows us to execute an independent "future" instruction when the one we should be executing stalls. We have:

	Instructions	Reordering	CPI
a.	I1: lw $2,-4($sp) I2: lw $3,0($4) I3: add $2,$2,$3 I4: sw $2,-4($sp) I5: addi $6,$6,1 I6: addi $4,$4,4 I7: slt $1,$6,$5 I8: bne $1,$0,Label	I3 stalls, but we do I5 instead.	1 (no stalls)
b.	I1: lw $2,0($4) I2: add $2,$2,$5 I3: sw $2,0($4) I4: add $4,$5,$0 I5: addi $5,$5,1 I6: slt $1,$5,$0 I7: bne $1,$0,Label	I2 stalls, and all subsequent instructions have dependences so this stall remains.	(7 + 1)/7 = 1.14

4.31.5 We use t0, t1, etc. as names for new registers in our renaming. We have:

	Instructions	Stalls	CPI

a.	`I1: lw    t1,-4($sp)`
	`I2: lw    $3,0($4)`
	`I3: add   $2,t1,$3`
	`I4: sw    $2,-4($sp)`
	`I5: addi  $6,$6,1`
	`I6: addi  $4,$4,4`
	`I7: slt   $1,$6,$5`
	`I8: bne   $1,$0,Label`
		I3 would stall, but I5 is executed instead.	1 (no stalls)

b.	`I1: lw    t1,0($4)`
	`I2: add   $2,t1,$5`
	`I3: sw    $2,0($4)`
	`I4: add   $4,$5,$0`
	`I5: addi  $5,$5,1`
	`I6: slt   $1,$5,$0`
	`I7: bne   $1,$0,Label`
		I2 stalls, and all subsequent instructions have dependences so this stall remains. Note that I4 or I5 cannot be done instead of I2 because of WAR dependences that are not eliminated. Renaming $4 in I4 or $5 in I5 does not eliminate any WAR dependences. This is a problem when renaming is done on the code (e.g., by the compiler). If the processor was renaming registers at runtime each instance of I4 would get a new name for the $4 it produces and we would be able to “cover” the I2 stall.	(7 + 1)/7 = 1.14

4.31.6 Note that now every time we execute an instruction it can be renamed differently. We have:

	Instructions	Reordering	CPI

a.	`I1: lw    t1,-4($sp)`
	`I2: lw    t2,0($4)`
	`I3: add   t3,t1,t2`
	`I4: sw    t3,-4($sp)`
	`I5: addi  t4,$6,1`
	`I6: addi  t5,$4,4`
	`I7: slt   t6,t4,$5`
	`I8: bne   t6,$0,Label`
	In next iteration uses of $6 renamed to t4, $4 renamed to t5.
		No stalls remain. I3 would stall stalls, but we can do I5 instead.	1 (no stalls)

b.	`I1: lw    t1,0($4)`
	`I2: add   t2,t1,$5`
	`I3: sw    t2,0($4)`
	`I4: add   t3,$5,$0`
	`I5: addi  t4,$5,1`
	`I6: slt   t5,t4,$0`
	`I7: bne   t5,$0,Label`
	In next iteration uses of $4 renamed to t3, $5 renamed to t4.
		No stalls remain. I2 would stall, but we can do I4 instead.	7/7 = 1

Solution 4.32

4.32.1 The expected number of mispredictions per instruction is the probability that a given instruction is a branch that is mispredicted. The number of instructions between mispredictions is one divided by the number of mispredictions per instruction. We get:

	Mispredictions per instruction	Instructions between mispredictions
a.	0.2 × (1 – 0.9)	50
b.	0.20 × (1 – 0.995)	1000

4.32.2 The number of in-progress instructions is equal to the pipeline depth times the issue width. The number of in-progress branches can then be easily computed because we know what percentage of all instructions are branches. We have:

	In-progress branches
a.	12 × 4 × 0.20 = 9.6
b.	25 × 4 × 0.20 = 20

4.32.3 We keep fetching from the wrong path until the branch outcome is known, fetching 4 instructions per cycle. If the branch outcome is known in stage N of the pipeline, all instructions are from the wrong path in N – 1 stages. In the Nth stage, all instructions after the branch are from the wrong path. Assuming that the branch is just as likely to be the 1st, 2nd, 3rd or 4th instruction fetched in its cycle, we have on average 1.5 instructions from the wrong path in the Nth stage (3 is branch is 1st, 2 is branch is 2nd, 1 is branch is 3rd, and 0 if branch is last). We have:

	Wrong-path instructions
a.	(10 – 1) × 4 × 1.5 = 37.5
b.	(18 – 1) × 4 × 1.5 = 69.5

4.32.4 We can compute the CPI for each processor, then compute the speed-up. To compute the CPI, we note that we have determined the number of useful instructions between branch mispredictions (for 4.32.1) and the number of misfetched instructions per branch misprediction (for 4.32.3), and we know how many instructions in total are fetched per cycle (4 or 8). From that we can determine the number of cycles between branch mispredictions, and then the CPI (cycles per useful instruction). We have:

	4-issue		8-issue			
	Cycles	CPI	Mis-fetched	Cycles	CPI	Speed-up
a.	(37.5 + 50)/4 = 21.9	21.9/50 = 0.438	(10 – 1) × 8 × 3.5 = 75.5	(75.5 + 50)/8 = 15.7	15.7/50 = 0.314	1.39
b.	(69.5 + 1000)/4 = 267.4	267.4/1000 = 0.267	(18 – 1) × 8 × 3.5 = 139.5	(139.5 + 1000)/8 = 142.4	142.4/1000 = 0.142	1.88

4.32.5 When branches are executed one cycle earlier, there is one less cycle needed to execute instructions between two branch mispredivctions. We have:

	"Normal" CPI	"Improved" CPI	Speed-up
a.	21.9/50 = 0.438	20.9/50 = 0.418	1.048
b.	267.4/1000 = 0.267	266.4/1000 = 0.266	1.004

4.32.6

	"Normal" CPI	"Improved" CPI	Speed-up
a.	15.7/50 = 0.314	14.7/50 = 0.294	1.068
b.	142.4/1000 = 0.142	141.4/1000 = 0.141	1.007

Speed-ups from this improvement are larger for the 8-issue processor than with the 4-issue processor. This is because the 8-issue processor needs fewer cycles to execute the same number of instructions, so the same 1-cycle improvement represents a large relative improvement (speed-up).

Solution 4.33

4.33.1 We need two register reads for each instruction issued per cycle:

	Read ports
a.	4 × 2 = 8
b.	2 × 2 = 4

4.33.2 We compute the time-per-instruction as CPI times the clock cycle time. For the 1-issue 5-stage processor we have a CPI of 1 and a clock cycle time of T. For an N-issue K-stage processor we have a CPI of 1/N and a clock cycle of T × 5/K. Overall, we get a speed-up of:

	Speed-up
a.	10/5 × 4 = 8
b.	25/5 × 2 = 10

4.33.3 We are unable to benefit from a wider issue width (CPI is 1), so we have:

	Speed-up
a.	10/5 = 2
b.	25/5 = 5

4.33.4 We first compute the number of instructions executed between mispredicted branches. Then we compute the number of cycles needed to execute these instructions if there were no misprediction stalls, and the number of stall cycles due to a misprediction. Note that the number of cycles spent on a misprediction in is the number of entire cycles (one less than the stage in which branches are executed) and a fraction of the cycle in which the mispredicted branch instruction is. The fraction of a cycle is determined by averaging over all possibilities. In an N-issue processor, we can have the branch as the first instruction of the cycle, in which case we waste (N − 1) Nths of a cycle, or the branch can be the second instruction in the cycle, in which case we waste (N − 2) Nths of a cycle, …, or the branch can be the last instruction in the cycle, in which case none of that cycle is wasted. With all of this data we can compute what percentage of all cycles are misprediction stall cycles:

	Instructions between branch mispredictions	Cycles between branch mispredictions	Stall Cycles	% Stalls
a.	1/(0.30 × 0.05) = 66.7	66.7/4 = 16.7	6.4	6/(16.7 + 6.4) = 26%
b.	1/(0.15 × 0.03) = 222.2	222.2/2 = 111.1	7.3	7/(111.1 + 7.3) = 5.9%

4.33.5 We have already computed the number of stall cycles due to a branch misprediction, and we know how to compute the number of non-stall cycles between mispredictions (this is where the misprediction rate has an effect). We have:

	Stall cycles between mispredictions	Need # of instructions between mispredictions	Allowed branch misprediction rate
a.	6.4	6.4 × 4/0.10 = 255	1/(255 × 0.30) = 1.31%
b.	7.3	7.3 × 2/0.02 = 725	1/(725 × 0.15) = 0.92%

The needed accuracy is 100% minus the allowed misprediction rate.

4.33.6 This problem is very similar to We have already computed the number of stall cycles due to a branch misprediction, and we know how to compute the number of non-stall cycles between mispredictions (this is where the misprediction rate has an effect). We have:, except that we are aiming to have as many stall cycles as we have non-stall cycles. We get:

	Stall cycles between mispredictions	Need # of instructions between mispredictions	Allowed branch misprediction rate
a.	6.4	6.4 × 4 = 25.5	1/(25.5 × 0.30) = 13.1%
b.	7.3	7.3 × 2 = 14.5	1/(14.5 × 0.15) = 46.0%

The needed accuracy is 100% minus the allowed misprediction rate.

Solution 4.34

4.34.1 We need an IF pipeline stage to fetch the instruction. Since we will only execute one kind of instruction, we do not need to decode the instruction but we still need to read registers. As a result, we will need an ID pipeline stage although it would be misnamed. After that, we have an EXE stage, but this stage is simpler because we know exactly which operation should be executed so there is no need for an ALU that supports different operations. Also, we need no Mux to select which values to use in the operation because we know exactly which value it will be. We have:

a.	In the ID stage we read two registers and we do not need a sign-extend unit. In the EXE stage we need an Add unit whose inputs are the two register values read in the ID stage. After the EXE stage we have a WB stage which writes the result from the Add unit into Rd (again, no Mux). Note that there is no MEM stage, so this is a 4-stage pipeline. Also note that the PC is always incremented by 4, so we do not need the other Add and Mux units that compute the new PC for branches and jumps.
b.	We only read one register in the ID stage so there is no need for the second read port in the Registers unit. We do need a sign-extend unit for the Offs field in the instruction word. In the EXE stage we need an Add unit whose inputs are the register value and the sign-extended offset from the ID stage. After the EXE stage we use the output of the Add unit as a memory address in the MEM stage, and then we have a WB stage which writes the value we read in the MEM stage into Rt (again, no Mux). Also note that the PC is always incremented by 4, so we do not need the other Add and Mux units that compute the new PC for branches and jumps.

4.34.2

a.	Assuming that the register write in WB happens in the first half of the cycle and the register reads in ID happen in the second half, we only need to forward the Add result from the EX/WB pipeline register to the inputs of the Add unit in the EXE stage of the next instruction (if that next instruction depends on the previous one). No hazard detection unit is needed because forwarding eliminates all hazards.
b.	Assuming that the register write in WB happens in the first half of the cycle and the register read in ID happens in the second half, we only need to forward the memory value from the MEM/WB pipeline register to the first (register) input of the Add unit in the EXE stage of the next or second-next instruction (if one of those two instructions is dependent on the one that has just read the value). We also need a hazard detection unit that stalls any instruction whose Rs register field is equal to the Rt field of the previous instruction.

4.34.3 We need to add some decoding logic to our ID stage. The decoding logic must simply check whether the opcode and funct filed (if there is a funct field) match this instruction. If there is no match, we must put the address of the exception handler into the PC (this adds a Mux before the PC) and flush (convert to `nops`) the undefined instruction (write zeros to the ID/EX pipeline register) and the following instruction which has already been fetched (write zeros to the IF/ID pipeline register).

4.34.4

a.	We need to add the logic that computes the branch address (sign-extend, shift-left–2, Add, and Mux to select the PC). We also need to replace the Add unit in EXE with an ALU that supports either an ADD or a comparison. The ALUOp signal to select between these operations must be supplied by the Control unit.
b.	We need to add back the second register read port (AND reads two registers), add the Mux that selects the value supplied to the second ALU input (register for AND, Offs for LW), add an ALUOp signal to select between two ALU operations, and replace the Add unit in EXE with an ALU that supports either an Add or an And operation. Finally, we must add to the WB stage the Mux that select whether the value to write to the register is the value from the ALU of from memory, and the Mux in the EX stage that selects which register to write to (Rd for AND, Rt for LW).

4.34.5

a.	The same forwarding logic used for forwarding from one ADD to another can also be used to forward from ADD to BEQ. We still need no hazard detection for data hazards, but we must add detection of control hazards. Assuming there is no branch prediction, whenever a BEQ is taken we must flush (convert to NOPs) all instructions that were fetched after that branch.
b.	We need to add forwarding from the EX/MEM pipeline register to the ALU inputs in the EXE stage (so AND can forward to the next instruction), and we need to extend our forwarding from the MEM/WB pipeline register to the second input of the ALU unit (so LW can forward to an AND whose Rt (input) register is the same as the Rt (result) register of the LW instruction. We also need to extend the hazard detection unit to also stall any AND instruction whose Rs or Rt register field is equal to the Rt field of the previous LW instruction.

4.34.6 The decoding logic must now check if the instruction matches either of the two instructions. After that, the exception handling is the same as for 4.34.3.

Solution 4.35

4.35.1 The worst case for control hazards is if the mispredicted branch instruction is the last one in its cycle and we have been fetching the maximum number of instructions in each cycle. Then the control hazard affects the remaining instructions in the branch's own pipeline stage and all instructions in stages between fetch and branch execution stage. We have:

	Delay slots needed
a.	7 × 4 – 1 = 27
b.	17 × 2 – 1 = 33

4.35.2 If branches are executed in stage X, the number of stall cycles due to a misprediction is (N – 1). These cycles are reduced by filling them with delay slot instructions. We compute the number of execution (non-stall) cycles between mispredictions, and the speed-up as follows:

	Non-stall cycles between mispredictions	Stall cycles without delay slots	Stall cycles with 4 delay slots	Speed-up due to delay slots
a.	1/(020 × (1 – 0.80) × 4) = 6.25	6	5	(6.25 + 6)/(6.25 + 5) = 1.089
b.	1/(025 × (1 – 0.92) × 2) = 25	16	14	(25 + 16)/(25 + 14) = 1.051

4.35.3 For 20% of branches, we add an extra instruction, for 30% of the branches we add two extra instructions, and for 40% of branches, we add three extra instructions. Overall, an average branch instruction is now accompanied by 0.20 + 0.30 × 2 + 0.40 × 3 = 2 `nop` instructions. Note that these `nops` are added for every branch, not just mispredicted ones. These `nop` instructions add to the execution time of the program, so we have:

	Total cycles between mispredictions without delay slots	Stall cycles with 4 delay slots	Extra cycles spent on NOPs	Speed-up due to delay slots
a.	6.25 + 6 = 12.25	5	0.5 × 6.25 × 0.20 = 0.625	12.5/(6.25 + 5 + 0.625) = 1.032
b.	25 + 16 = 41	14	1 × 25 × 0.25 = 6.25	41/(25 + 14 + 6.25) = 0.906

4.35.4

a.

```
      add $2,$0,$0        ;  $1=0
Loop: beq $2,$3,End
      lb  $10,1000($2)    ;  Delay slot
      sb  $10,2000($2)
      beq $0,$0,Loop
      addi $2,$2,1        ;  Delay slot
Exit:
```

b.

```
      add $2,$0,$0        ;  $1=0
Loop: lb  $10,1000($2)
      lb  $11,1001($2)
      beq $10,$11,End
      addi $1,$1,1        ;  Delay slot
      beq $0,$0,Loop
      addi $2,$2,1        ;  Delay slot
Exit: addi $1,$1,-1       ;  Undo c++ from delay slot
```

4.35.5

a.

```
      add $2,$0,$0        ;  $1=0
Loop: beq $2,$3,End
      lb  $10,1000($2)    ;  Delay slot
      nop                 ;  2nd delay slot
      beq $0,$0,Loop
      sb  $10,2000($2)    ;  Delay slot
      addi $2,$2,1        ;  2nd delay slot
Exit:
```

b.

```
      add $2,$0,$0        ;  $1=0
      lb  $10,1000($2)    ;  Prepare for first iteration
      lb  $11,1001($2)    ;  Prepare for first iteration
Loop: beq $10,$11,End
      addi $1,$1,1        ;  Delay slot
      addi $2,$2,1        ;  2nd delay slot
      beq $0,$),Loop
      lb  $10,1000($2)    ;  Delay slot, prepare for next iteration
      lb  $11,1001($2)    ;  2nd delay slot, prepare for next iteration
Exit: addi $1,$1,-1       ;  Undo c++ from delay slot
      addi $2,$2,-1       ;  Undo i++ from 2nd delay slot
```

4.35.6 The maximum number of in-flight instructions is equal to the pipeline depth times the issue width. We have:

	Instructions in flight	Instructions per iteration	Iterations in flight
a.	10 × 4 = 40	5	40/5 + 1 = 9
b.	25 × 2 = 50	6	roundUp(50/6) + 1 = 10

Note that an iteration is in-flight when even one of its instructions is in-flight. This is why we add one to the number we compute from the number of instructions in flight (instead of having an iteration entirely in flight, we can begin another one and still have the "trailing" one partially in-flight) and round up.

Solution 4.36

4.36.1

	Instruction	Translation

a.	`lwinc Rt,Offset(Rs)`	`lw Rt,Offset(Rs)`
		`addi Rs,Rs,4`

b.	`addr  Rt,Offset(Rs)`	`lw tmp,Offset(Rs)`
		`add Rt,Rt,tmp`

4.36.2 The ID stage of the pipeline would now have a lookup table and a micro-PC, where the opcode of the fetched instruction would be used to index into the lookup table. Micro-operations would then be placed into the ID/EX pipeline register, one per cycle, using the micro-PC to keep track of which micro-op is the next one to be output. In the cycle in which we are placing the last micro-op of an instruction into the ID/EX register, we can allow the IF/ID register to accept the next instruction. Note that this results in executing up to one micro-op per cycle, but we actually fetching instructions less often than that.

4.36.3

	Instruction
a.	We need to add an incrementer in the MEM stage. This incrementer would increment the value read from Rs while memory is being accessed. We also need to change the Registers unit to allow two writes to happen in the same cycle, so we can write the value from memory into Rt and the incremented value of Rs back into Rs.
b.	We need another EX stage after the MEM stage to perform the addition. The result can then be stored into Rt in the WB stage.

4.36.4 Not often enough to justify the changes we need to make to the pipeline. Note that these changes slow down all the other instructions, so we are speeding up a relatively small fraction of the execution while slowing down everything else.

4.36.5 Each original `addm` instruction now results in executing two more instructions, and also adds a stall cycle (the `add` depends on the `lw`). As a result, each cycle in which we executed an addm instruction now adds three more cycles to the execution. We have:

	Speed-up from addm translation
a.	1/(1 + 0.05 × 3) = 0.87
b.	1/(1 + 0.10 × 3) = 0.77

4.36.6 Each translated addm adds the 3 stall cycles, but now half of the existing stalls are eliminated. We have:

	Speed-up from addm translation
a.	1/(1 + 0.05 × 3 – 0.05/2) = 0.89
b.	1/(1 + 0.10 × 3 – 0.10/2) = 0.8

Solution 4.37

4.37.1 All of the instructions use the instruction memory, the PC + 4 adder, the control unit (to decode the instruction), and the ALU. For the least utilized unit, we have:

a.	The result of the branch adder (add offset to PC + 4) is only used by the BEQ instruction, the data memory read port is only used by the LW instruction, and the write port is only used by the last SW instruction (the first SW is not executed because the BEW is taken).
b.	The result of the branch adder (add offset to PC + 4) is never used.

Note that the branch adder performs its operation in every cycle, but its result is actually used only when a branch is taken.

4.37.2 The read port is only used by lw and the write port by sw instructions. We have:

	Data memory read	Data memory write
a.	25% (1 out of 4)	25% (1 out of 4)
b.	40% (2 out of 5)	20% (1 out of 5)

4.37.3 In the IF/ID pipeline register, we need 32 bits for the instruction word and 32 bits for PC + 4 for a total of 64 bits. In the ID/EX register, we need 32 bits for each of the two register values, the sign-extended offset/immediate value, and PC + 4 (for exception handling). We also need 5 bits for each of the three register fields from the instruction word (Rs, Rt, Rd), and 10 bits for all the control signals output by the Control unit. The total for the ID/EX register is 153 bits.

In the EX/MEM register, we need 32 bits each for the value of register Rt and for the ALU result. We also need 5 bits for the number of the destination register and 4 bits for control signals. The total for the EX/MEM register is 73 bits. Finally, for the MEM/WB register we need 32 bits each for the ALU result and value from memory, 5 bits for the number of the destination register, and 2 bits for control signals. The total for MEM/WB is 71 bits. The grand total for all pipeline registers is 361 bits.

4.37.4 In the IF stage, the critical path is the I-Mem latency. In the ID stage, the critical path is the latency to read Regs. In the EXE stage, we have a Mux and then ALU latency. In the MEM stage we have the D-Mem latency, and in the WB stage we have a Mux latency and setup time to write Regs (which we assume is zero). For a single-cycle design, the clock cycle time is the sum of these per-stage latencies (for a load instruction). For a pipelined design, the clock cycle time is the longest of the per-stage latencies. To compare these clock cycle times, we compute a speed-up based on clock cycle time alone (assuming the number of clock cycles is the same in single-cycle and pipelined designs, which is not true). We have:

	IF	ID	EX	MEM	WB	Single-cycle	Pipelined	"Speed-up"
a.	400ps	200ps	150ps	350ps	30ps	1130ps	400ps	2.83
b.	500ps	220ps	280ps	1000ps	100ps	2100ps	1000ps	2.10

Note that this speed-up is significantly lower than 5, which is the "ideal" speed-up of 5-stage pipelining.

4.37.5 If we only support `add` instructions, we do not need the MUX in the WB stage, and we do not need the entire MEM stage. We still need Muxes before the ALU for forwarding. We have:

	IF	ID	EX	WB	Single-cycle	Pipelined	"Speed-up"
a.	400ps	200ps	150ps	0ps	750ps	400ps	1.88
b.	500ps	220ps	280ps	0ps	1000ps	500ps	2.00

Note that the "ideal" speed-up from pipelining is now 4 (we removed the MEM stage), and the actual speed-up is about half of that.

4.37.6 For the single cycle design, we can reduce the clock cycle time by 1ps by reducing the latency of any component on the critical path by 1ps (if there is only one critical path). For a pipelined design, we must reduce latencies of all stages that have longer latencies than the target latency. We have:

	Single-cycle	Needed cycle time for pipelined	Cost for Pipelined
a.	0.2 × 1130 = $226	0.8 × 400ps = 320ps	$80 + $30 = $130 (IF and MEM)
b.	0.2 × 2100 = $420	0.8 × 1000ps = 800ps	$200 (MEM)

Note that the cost of improving the pipelined design by 20% is lower. This is because its clock cycle time is already lower, so a 20% improvement represents fewer picoseconds (and fewer dollars in our problem).

Solution 4.38

4.38.1 The energy for the two designs is the same: I-Mem is read, two registers are read, and a register is written. We have:

a.	100pJ + 2 × 60pJ + 70pJ = 290pJ
b.	200pJ + 2 × 90pJ + 80pJ = 460pJ

4.38.2 The instruction memory is read for all instructions. Every instruction also results in two register reads (even if only one of those values is actually used). A load instruction results in a memory read and a register write, a store instruction results in a memory write, and all other instructions result in either no register write (e.g., `beq`) or a register write. Because the sum of memory read and register write energy is larger than memory write energy, the worst-case instruction is a load instruction. For the energy spent by a load, we have:

a.	100pJ + 2 × 60pJ + 70pJ + 120pJ = 410pJ
b.	200pJ + 2 × 90pJ + 80pJ + 300pJ = 760pJ

4.38.3 Instruction memory must be read for every instruction. However, we can avoid reading registers whose values are not going to be used. To do this, we must add RegRead1 and RegRead2 control inputs to the Registers unit to enable or disable each register read. We must generate these control signals quickly to avoid lengthening the clock cycle time. With these new control signals, a `lw` instruction results in only one register read (we still must read the register used to generate the address), so we have:

	Energy before change	Energy saved by change	% Savings
a.	100pJ + 2 × 60pJ + 70pJ + 120pJ = 410pJ	60pJ	14.6%
b.	200pJ + 2 × 90pJ + 80pJ + 300pJ = 760pJ	90pJ	11.8%

4.38.4 Before the change, the Control unit decodes the instruction while register reads are happening. After the change, the latencies of Control and Register Read cannot be overlapped. This increases the latency of the ID stage and could affect the processor's clock cycle time if the ID stage becomes the longest-latency stage. We have:

	Clock cycle time before change	Clock cycle time after change
a.	400ps (I-Mem in IF stage)	500ps (Ctl then Regs in ID stage)
b.	1000ps (D-Mem in MEM stage)	No change (400ps + 220ps < 1000ps).

4.38.5 If memory is read in every cycle, the value is either needed (for a load instruction), or it does not get past the WB Mux (or a non-load instruction that writes to a register), or it does not get written to any register (all other instructions, including stall). This change does not affect clock cycle time because the clock cycle time must already allow enough time for memory to be read in the MEM stage. It does affect energy: a memory read occurs in every cycle instead of only in cycles when a load instructions is in the MEM stage.

4.38.6

	I-Mem active energy	I-Mem latency	Clock cycle time	Total I-Mem Energy	Idle energy %
a.	100pJ	400ps	400ps	100pJ	0%
b.	200pJ	500ps	1000ps	200pJ + 500ps × 0.1 × 200pJ/500ps = 220pJ	20pJ/220pJ = 9.1%

Solution 4.39

4.39.1 The number of instructions executed per second is equal to the number of instructions executed per cycle (IPC, which is 1/CPI) times the number of cycles per second (clock frequency, which is 1/T where T is the clock cycle time). The IPC is he percentage of cycle in which we complete an instruction (and not a stall), and the clock cycle time is the latency of the maximum-latency pipeline stage. We have:

	IPC	Clock cycle time	Clock frequency	Instructions per second
a.	0.85	500ps	2.00 GHz	1.70×10^9
b.	0.70	200ps	5.00 GHz	3.50×10^9

4.39.2 Power is equal to the product of energy per cycle times the clock frequency (cycles per second). The energy per cycle is the total of the energy expenditures in all five stages. We have:

	Clock Frequency	Energy per cycle (in pJ)	Power (W)
a.	2.00 GHz	120 + 60 + 75 + 0.30 × 120 + 0.55 × 20 = 305	0.61
b.	5.00 GHz	150 + 60 + 50 + 0.35 × 150 + 0.50 × 20 = 322.5	1.61

4.39.3 The time that remains in the clock cycle after a circuit completes its work is often called slack. We determine the clock cycle time and then the slack for each pipeline stage:

	Clock cycle time	IF slack	ID slack	EX slack	MEM slack	WB slack
a.	500ps	200ps	100ps	150ps	0ps	400ps
b.	200ps	0ps	50ps	80ps	10ps	60ps

4.39.4 All stages now have latencies equal to the clock cycle time. For each stage, we can compute the factor X for it by dividing the new latency (clock cycle time) by the original latency. We then compute the new per-cycle energy consumption for each stage by dividing its energy by its factor X. Finally, we re-compute the power dissipation:

	X for IF	X for ID	X for EX	X for MEM	X for WB	New Power (W)
a.	500/300	500/400	500/350	500/500	500/100	0.43
b.	200/200	200/150	200/120	200/190	200/140	1.41

4.39.5 This changes the clock cycle time to 1.1 of the original, which changes the factor X for each stage and the clock frequency. After that this problem is solved in the same way as all stages now have latencies equal to the clock cycle time. For each stage, we can compute the factor X for it by dividing the new latency (clock cycle time) by the original latency. We then compute the new per-cycle energy consumption for each stage by dividing its energy by its factor X. Finally, we re-compute the power dissipation:. We get:

	X for IF	X for ID	X for EX	X for MEM	X for WB	New Power (W)
a.	550/300	550/400	550/350	550/500	550/100	0.35
b.	220/200	220/150	220/120	220/190	220/140	1.16

4.39.6 The X factor for each stage is the same as in this changes the clock cycle time to 1.1 of the original, which changes the factor X for each stage and the clock frequency. After that this problem is solved in the same way as all stages now have latencies equal to the clock cycle time. For each stage, we can compute the factor X for it by dividing the new latency (clock cycle time) by the original latency. We then compute the new per-cycle energy consumption for each stage by dividing its energy by its factor X. Finally, we re-compute the power dissipation:. We get:, but this time in our power computation we divide the per-cycle energy of each stage by X^2 instead of x. We get:

	New Power (W)	Old Power (W)	Saved
a.	0.24	0.61	60.7%
b.	0.95	1.61	41.0%