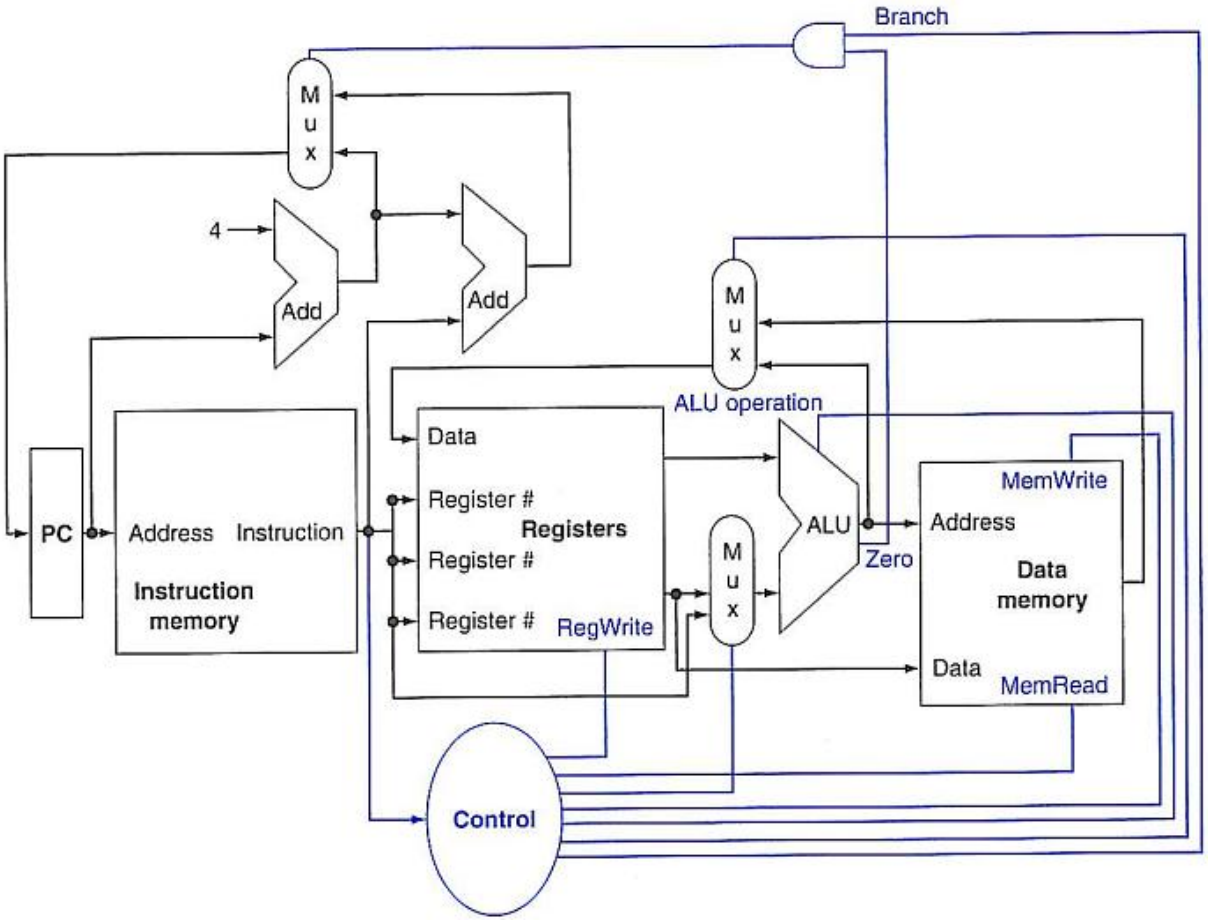
Chapter 4 (Solution)

I. Datapath

Exercise 4.1



4.1.1

	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

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

Registers sử dụng dù 1 cổng đầu vào (port) cũng được xem là sử dụng toàn khối nên ko cần ghi rõ là port nào được sử dụng ở đây

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

a. I-Mem, Regs, Mux, ALU, Mux, Regs

b. I-Mem, Regs, Mux, ALU, Mux, Regs

4.1.5

a. I-Mem, Regs, Mux, ALU, D-Mem, Mux, Regs

b. I-Mem, Regs, Mux, ALU, D-Mem, Mux, Regs

4.1.6

a. I-Mem, Regs, Mux, ALU, Mux

b. 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

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.

Exercise 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.

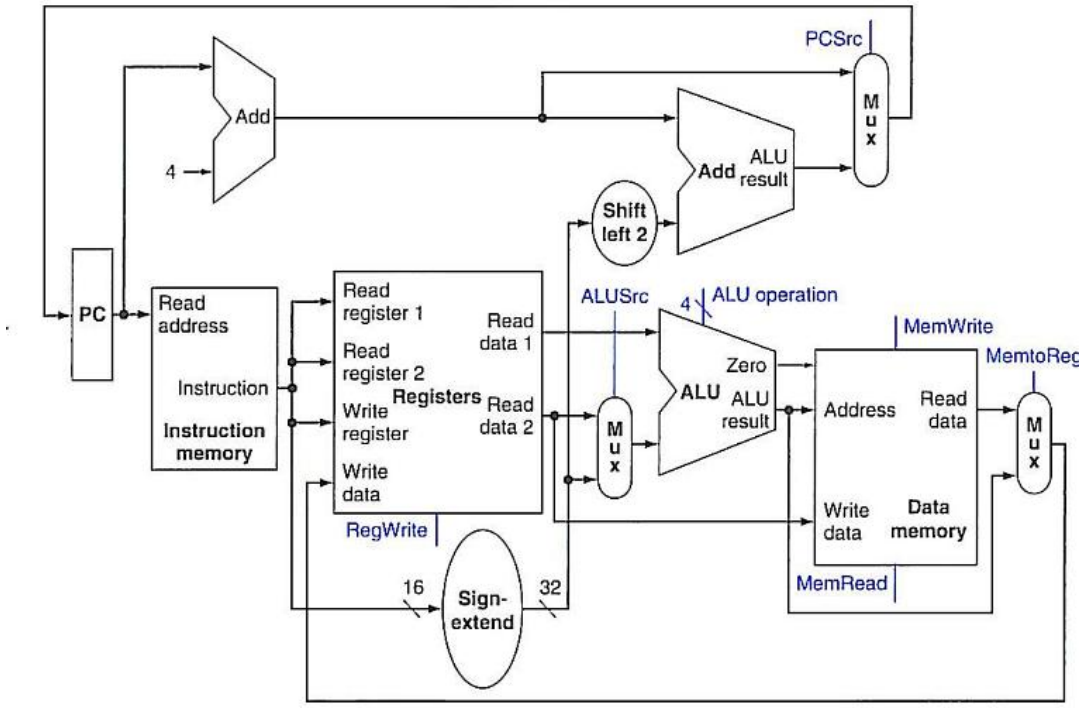
4.2.2 New functional blocks needed for this instruction are:

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.

Exercise 4.6



4.6.1

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.

Exercise 4.7

4.7.1

Critical path: I-Mem, Regs, Mux, ALU, Mux, Regs

a. 400ps + 200ps + 30ps + 120ps + 30ps + 200ps = 980ps
b. 500ps + 220ps + 100ps + 180ps + 100ps + 220ps = 1320ps

4.7.2

Critical path: I-Mem, Regs, Mux, ALU, D-Mem, Mux, Regs

c. 400ps + 200ps + 30ps + 120ps + 350ps + 30ps + 200ps = 1330ps
d. 500ps + 220ps + 100ps + 180ps + 1000ps + 30ps + 220ps = 2320ps

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, Mux, Regs

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 1330ps/1290ps = 1.031
   1330
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 2320ps/2220ps = 1,045

Exercise 4.9

4.9.1

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

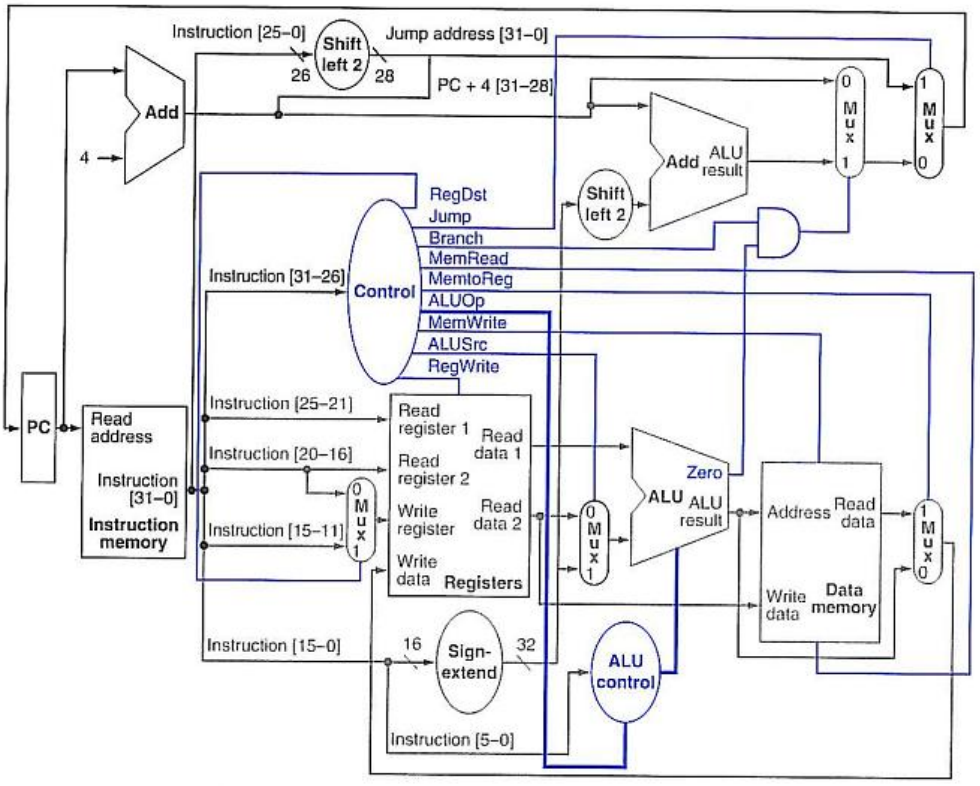
4.9.2

	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

Exercise 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, Mux, Regs 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 + 200ps = 1360ps	1360 – 400 = 960ps
b.	500ps + 100ps + 220ps + 100ps + 180ps + 1000ps + 100ps + 220ps = 2420ps	2420 – 500 = 1920ps

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.

II. Pipeline

Exercise 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.	$0.15 \times 5 + 0.85 \times 4 = 4.15$	1650ps/500ps = 3.30
b.	$0.30 \times 5 + 0.70 \times 4 = 4.30$	800ps/200ps = 4.00

Exercise 4.13

4.13.1 (Don’t care)

4.13.2

	Instruction sequence

a.	`lw $1,40($6)`
	`add $6,$2,$2`
	`nop`
	`sw $6,50($1)`

b.	`lw $5,-16($5)`
	`nop`
	`nop`
	`sw $5,-16($5)`
	`add $5,$5,$5`

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)`

b.	`lw $5,-16($5)`
	`nop`
	`sw $5,-16($5)`
	`add $5,$5,$5`

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)