Selected Exercise for Chapter 4. 4.1, 4.2, 4.3, 4.5, 4.8.1~3, 4.9, 4.10, 4.12, 4.16.1~3, 4.19.1~3

Exercise 4.1

Different instructions utilize different hardware blocks in the basic single-cycle implementation. The next three problems in this exercise refer to the following instruction:

Instruction	Interpretation
add Rd ,Rs ,Rt	Reg[Rd] = Reg[Rs] + Reg[Rt]

4.1.1 [5] <4.1> what are the values of control signals generated by the control in Figure 4.2 for this instruction?

4.1.1 Solution: The values of the signals are as follows:

RegWrite	MemRead	ALUSrc	MemWrite	ALUOp	MemToReg	Branch
1	0	0 (Reg)	0	Add	0(ALU)	0

ALUSrc 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. MemToReg 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 [5] <4.1> which resources (blocks) perform a useful function for this instruction?

4.1.2 Solution: Resources performing a useful function for this instruction are the following: All except Data Memory and branch Add unit.

4.1.3 [10] <4.1> which resources (blocks) produce outputs, but their outputs are not used for this instruction? Which resources produce no outputs for this instruction?

4.1.3 Solution: Outputs that are not used : Branch Add. No output: Data Memory

4.2.1 Solution: This instruction uses instruction memory, both register read ports, the ALU to add Rd and Rs together, data memory, and write port in Registers.

4.2.2 Solution: None. This instruction can be implemented using existing blocks.

4.2.3 Solution: None. This instruction can be implemented without adding new control signals. It only requires changes in the Control logic.

4.3.1 Solution: 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 400 ps + 200 ps + 30 ps + 120 ps + 350 ps + 30 ps = 1130 ps. 1430 ps (1130 ps + 300 ps, ALU is on the critical path).

4.3.2 Solution: The speedup comes from changes in clock cycle time and changes to the number of clock cycles we need for the program: We need 5% fewer cycles for a program, but cycle time is 1430 instead of

1130, so we have a speedup of (1/0.95)*(1130/1430) = 0.83, which means we actually have a slowdown.

4.3.3 Solution: 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 speedup we previously computed, and our cost/ performance relative to the baseline is as follows:
New Cost: 3890 + 600 = 4490
Relative Cost: 4490/3890 = 1.15
Cost/Performance: 1.15/0.83 = 1.39. We are paying significantly more for significantly worse performance; the cost/performance is a lot worse than with the unmodified processor.

4.5.1 Solution: The data memory is used by LW and SW instructions, so the answer is:
25% + 10% = 35%

4.5.2 The input of the sign-extend circuit is needed for ADDI (to provide the immediate ALU operand), BEQ (to provide the PC-relative off set), and LW and SW (to provide the offset used in addressing memory) so the answer is: 20% + 25% + 25% + 10% = 80%

The sign-extend circuit is actually computing a result in every cycle, but its output is ignored for ADD and NOT instructions.

4.8.1 Solution: Pipeline: 350 ps. Single-cycle: 250+ 350+150+300+200=1250ps

4.8.2 Solution: Pipeline: 350*5= 1750ps. Single-cycle: 1250ps

4.8.3 Solution: Stage to split is ID because it has the largest latency. New clock cycle time is 300 ps

4.9.1

Instruction sequence	Dependences
I1: OR R1,R2,R3 I2: OR R2,R1,R4 I3: OR R1,R1,R2	RAW on R1 from I1 to I2 and from I1 to I3 RAW on R2 from I2 to I3 WAR on R2 from I1 to I2 WAR on R1 from I2 to I3 WAW on R1 from I1 to I3

4.9.2 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	
OR R1,R2,R3 NOP NOP OR R2,R1,R4 NOP NOP OR R1,R1,R2	Delay I2 to avoid RAW hazard on R1 from I1 Delay I3 to avoid RAW hazard on R2 from I2

4.9.3 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	
OR R1,R2,R3	
OR R2,R1,R4	No RAW hazard on R1 from I1 (forwarded)
OR R1,R1,R2	No RAW hazard on R2 from I2 (forwarded)

4.9.4 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 4.9.2, and execution forwarding must add a stall cycle for every NOP we had in 4.9.3. Overall, we get:
No forwarding: (7 + 4)*250 ps = 2750 ps
With forwarding: 7*300 ps = 2100 ps. Speedup due to forwarding= 1.31

4.9.5 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	
OR R1,R2,R3	
OR R2,R1,R4	ALU-ALU forwarding of R1 from I1
OR R1,R1,R2	ALU-ALU forwarding of R2 from I2

4.9.6 solution
No forwarding: (7 + 4)*250 ps = 2750 ps
With ALU-ALU forwarding only: 7*290 ps = 2030 ps
Speedup with ALU-ALU forwarding: 1.355

4.10.1 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. X represents this pipeline stage is not

used. The Cycles are represented from left to right, and for each instruction we show the pipeline stage it is in during that cycle:

SW	IF	ID	EX	ME	X								
LW		IF	ID	EX	ME	WB							
BEQ			IF	ID	EX	X	WB						
ADD				***	***	IF	ID	EX	ME	WB			
SLT							IF	ID	EX	ME	WB		

Instruction	Pipeline Stage	Cycles

SW R16,12(R6)
LW R16,8(R6)
BEQ R5,R4,Lbl
ADD R5,R1,R4
SLT R5,R15,R4
	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
		11

We cannot add NOPs to the code to eliminate this hazard because NOPs need to be fetched from the memory just like any other instructions. Therefore, this hazard must be addressed with a hardware hazard detection unit in the processor.

4.10.2 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 instructions, the change would help eliminate some stall cycles.

Instructions Executed	Cycles with 5 stages	Cycles with 4 stages	Speedup
5	4+5=9	3+5=8	9/8=1.13

4.10.3 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	Speedup
5	1	4+5+1*2=11	4+5+1*1=10	11/10=1.10

A more detailed diagram
Determined in EXE stage

SW	IF	ID	EX	ME	X								

LW		IF	ID	EX	ME	WB							
BEQ			IF	ID	EX	X	WB						
ADD				***	***	IF	ID	EX	ME	WB			
SLT							IF	ID	EX	ME	WB		

A more detailed diagram

Determined in ID stage

SW	IF	ID	EX	ME	X								
LW		IF	ID	EX	ME	WB							
BEQ			IF	ID	EX	X	WB						
ADD				***	IF	ID	EX	X	WB				
SLT						IF	ID	EX	ME	WB			

4.10.4 The number of cycles for the (normal) 5-stage and the (combined EX/ MEM) 4-stage pipeline is already computed in 4.10.2. 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	Speedup
200 ps (IF)	210 ps (MEM + 20 ps)	(9*200)/(8*210) = 1.07

4.10.5

New ID latency	New EX latency	New cycle time	Old cycle time	Speedup
180 ps	140 ps	200ps(IF)	200ps(IF)	(11*200)/(10*200) = 1.10

4.10.6 The cycle time remains unchanged: a 20 ps 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 speedup from this change will be below 1 (a slowdown). In 4.10.3, 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)	Speedup
a.	4+5+1*2=11	11*200 ps=2200 ps	4+5+1*3=12	12*200 ps=2400 ps	11/12=0.92

Branch is determined in MEM stage

SW	IF	ID	EX	ME	X								
LW		IF	ID	EX	ME	WB							
BEQ			IF	ID	EX	X	WB						
ADD				***	***	***	IF	ID	EX	ME	WB		
SLT								IF	ID	EX	ME	WB	

4.12.1 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 Cycle
1+0.35*2+0.15*1=1.85 (EX to 1, MEM to 1, EX to 1) *2 + (EX to 2, MEM to 2)*2	46%(0.85/1.85)

4.12.2 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 Cycle
1+0.20=1.20	17%(0.20/1.20)

4.12.3 With forwarding only from the EX/MEM register, EX to 1st dependences can be satisfied without stalls but any other dependences (even when together with EX to 1st) incur a one-cycle stall. With forwarding only from the MEM/WB register, EX to 2nd dependences incur no stalls. MEM to1st dependences still incur a one-cycle stall, 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
0.2+0.05+0.1+0.1=0.45	0.05+0.2+0.1=0.35	MEM/WB

4.12.4. In 4.12.1 and 4.12.2, 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	Speedup
1.85*150 ps=277.5 ps	1.20*150 ps=180 ps	1.54

4.12.5 We already computed the time per instruction for full forwarding in 4.12.4. Now we compute time-per instruction with time-travel forwarding and the speedup over full forwarding:

With full forwarding	Time-travel forwarding	Speedup

1.20*150 ps=180 ps	1*250 ps=250 ps	0.72

4.12.6

EX/MEM	MEM/WB	Shorter time per instruction with
1.45*150 ps=217.5	1.35*150 ps = 202.5 ps	MEM/WB

Exercise 4.16

This exercise examines the accuracy of various branch predictors for the following repeating pattern of branch outcomes: T, T, NT, T

4.16.1 [5] <4.8> What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?

4.16.1 Solution: always-taken 3/4=75%, always not-taken 1/4=25/%

4.16.2 [5] <4.8> What is the accuracy of the two-bit predictor for the first four branches in this pattern, assuming that the predictor starts off in the bottom left state from Figure 4.63 (predict not taken).

4.16.2 Solution

Outcomes	Predictor value at time of prediction	Correct or Incorrect	Accuracy
T, T, NT, T	0, 1, 2, 1	I, I, I, I	0%

4.16.3 [10] <4.8> What is the accuracy of the two-bit predictor if this pattern is repeated forever?

4.16.13 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).

Repeated Pattern: T T N T T T N T T T N T T T N T T T N T

Predictor value at time of prediction: 0 1 2 1 2 3 2 3 3 3 2 3 3 3 2 3 3 3 2 3

Prediction result in steady state C C I C C C I C C C I C

Accuracy in steady state: 75%

4.16.4 [30] <4.8> Design a predictor that would achieve a perfect accuracy if this pattern is repeated forever. Your predictor should be a sequential circuit with one output that provides a prediction (1 for taken, 0 for not taken) and no inputs other than the clock and the control signal that indicates that the instruction is a conditional branch.

4.16.4 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.

4.16.5 [10] <4.8> What is the accuracy of your predictor from 4.24.4 if it is given a repeating pattern that is the exact opposite of this one?

4.16.5 Since the predictor's output is always the opposite of the actual outcome of the branch instruction, the

accuracy is zero.

4.19.1 The energy for the two designs is the same: I-Mem is read, two registers are read, and a register is written. We have: 140 pJ+2*70 ps +60 pJ =340 pJ

4.19.2 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: 140 pJ+2*70 pJ +60 pJ + 140 pJ =480 pJ

4.19.3 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
140 pJ+2*70 pJ +60 pJ +140 pJ =480 pJ	70 pJ	14.6%