4.16

(1) Pipelined: 350; non-pipelined: 1250

(2) Pipelined: 1750; non-pipelined: 1250

(3) Split the ID stage. This reduces the clock-cycle time to 300ps.

(4) 35%

(5) 65%

4.27

(1)
```
add $s3, $s1, $s0
nop
nop
ld $s2, 4($s3)
ld $s1, 0($s4)
nop
or $s2, $s3, $s2
nop
nop
sw $s2, 0($s3)
```

(2) It is not possible to reduce the number of NOPs.

(3) The code executes correctly. We need hazard detection only to insert a stall when the instruction following a load uses the result of the load. That does not happen in this case.

(4) Because there are no stalls in this code, PCWrite and IF/IDWrite are always 1 and the mux before ID/EX is always set to pass the control values through.

Cycle	1	2	3	4	5	6	7	8	
add	IF	ID	EX	ME	WB				
ld		IF	ID	EX	ME	WB			
ld			IF	ID	EX	ME	WB		
or				IF	ID	EX	ME	WB	
sd					IF	ID	EX	ME	WB

1. ForwardA = X; ForwardB = X (no instruction in EX stage yet)
2. ForwardA = X; ForwardB = X (no instruction in EX stage yet)
3. ForwardA = 0; ForwardB = 0 (no forwarding; values taken from registers)
4. ForwardA = 2; ForwardB = 0 (base register $s3 taken from result of previous instruction)
5. ForwardA = 0; ForwardB = 0 (no forwarding; values taken from register and immediate)
6. ForwardA = 0; ForwardB = 1 ($s3 taken from register; $s2 from two instructions ago)
7. ForwardA = 0; ForwardB = 2 ($s3 taken from register file. $s2 taken from previous instruction)

(5) The hazard detection unit additionally needs the values of rd that comes out of the MEM/WB register. The instruction that is currently in the ID stage needs to be stalled if it depends on a value produced by (or forwarded from) the instruction in the EX or the instruction in the MEM stage. So we need to check the destination register of these two instructions. The Hazard unit already has the value of rd from the EX/MEM register as inputs, so we need only add the value from the MEM/WB register.

No additional outputs are needed. We can stall the pipeline using the three output signals that we already have. The value of rd from EX/MEM is needed to detect the data hazard between the add and the following ld. The value of rd form MEM/WB is needed to detect the data hazard between the first ld instruction and the or instruction.

(6)

1. PCWrite = 1; IF/IDWrite = 1; control mux = 0
2. PCWrite = 1; IF/IDWrite = 1; control mux = 0
3. PCWrite = 1; IF/IDWrite = 1; control mux = 0
4. PCWrite = 0; IF/IDWrite = 0; control mux = 1
5. PCWrite = 0; IF/IDWrite = 0; control mux = 1

4.28

(1) The CPI increases from 1 to 1. 1375. 55% of the branches will result in the flushing of 1 instruction, giving us a CPI of 1 + (1 − 0.45)(0.25)1 = 1.1375.

(2) The CPI increases from 1 to 1.1125. (1 + (.25)(1 − .55)1 = 1.1125)

(3) The CPI increases from 1 to 1.0375. (1 + (.25)(1 − .85)1 = 1.0375)

(4) The speedup is approximately 1.019. Changing half of the branch instructions to an ALU instruction reduces the percentage of instructions that are branches from 25% to 12.5%. Because predicted and mispredicted branches are replaced equally, the misprediction rate remains 15%. Thus, the new CPU is 1 + (.125)(1 − .85)(1) = 1.01875. This represents a speedup of 1.0375 / 1.01875 = 1.019

(5) The “speedup” is .91. There are two ways to look at this problem. One way is to look at the two ADD instruction as a branch with an “extra” cycle. Thus, half of the branches have 1 extra cycle; 15% of the other half have 1 extra cycles (the pipeline flush); and the remaining branches (those correctly redicted) have no extra cycles. This gives us a CPI of 1 + (.5)(.25)*1 + (.5)(.25)(.15)*1 = 1.14375 and a speedup of 1.0375 / 1.14375 = .91. We can also treat the ADD instructions as separate instructions. The modified program now has 1.125n instructions (half of 25% produce an extra instruction). .125n of these 1.125n instructions (or 11.1%) are branches. The CPI for this new program is 1 + (.111)(.15)*1 = 1.01665. When we factor in the 12.5% increase in instructions, we get a speedup of 1.0375 / (1.125 * 1.01665) = .91.

(6) The predictor is 25% accurate on the remaining branches. We know that 80% of branches are always predicted correctly and the overall accuracy is 0.85. Thus, 0.8*1 + 0.2*x = 0.85. Solving for x shows that x = 0.25.