Exercises for Week 7
Textbook Sections 4.5 and 4.6
Sample Solutions

Question 1

Based on Textbook Exercise 4.16 <§4.5>.

In this Exercise, we examine how pipelining affect the clock cycle time of the processor. Problems in this exercise assume that individual stages of the datapath have the following latencies:

IF	ID	EX	MEM	WB
250ps	350ps	150ps	300ps	200ps

Assume also that instructions executed by the processor are broken down as follows:

ALU/Logic	Jump/Branch	Load	Store
45%	20%	20%	15%

a. What is the clock cycle time in a pipelined and non-pipelined processor?

Pipelined: 350ps (longest time of any stage).

Single-cycle: 1250ps (sum of the stage times).

b. What is the total latency of an `ld` instruction in a pipelined and non-pipelined processor?

Pipelined: 1750ps (1 cycle for each stage, 5 stages).

Single-cycle: 1250ps (1 cycle).

c. If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor?

Stage to split: ID.

New clock cycle time: 300ps (MEM stage now has the longest time).

d. Assuming there are no stalls or hazards, what is the utilization (% of cycles it is used) of the data memory?

The data memory is used for Load and Store instructions: 35%.

e. Assuming there are no stalls or hazards, what is the utilization of the write-register port of the "Registers" unit?

The write-register port is used for ALU/Logic and Load instructions: 65%.

f. Instead of a single-cycle organization, we can use a multi-cycle organization where each instruction takes multiple cycles but one instruction finishes before another is fetched. In this organization, an instruction only goes through stages it actually needs (e.g. `sd` only takes 4 cycles because it does not need the WB stage). Compare clock cycle times and execution times of single-cycle and multi-cycle organizations with a pipelined organization.

We already computed clock cycle times for pipelined and single-cycle organizations. The multi-cycle organization has the same clock cycle time as the pipelined organization, for the same reason: the cycle time is determined by the stage with the longest latency.

In a pipelined organization, a long-running program with no pipeline stalls completes one instruction in every cycle. We will compute execution times of the other organizations relative to this.

In a single-cycle organization, every instruction takes one clock cycle, but the clock cycle is longer. Execution time is 1250ps/350ps = 3.57 times pipelined execution time.

A multi-cycle organization completes a Load in 5 cycles, a Store in 4 cycles (no WB), an ALU/Logic instruction in 4 cycles (no MEM), and a Jump/Branch in 4 cycles (no WB). Execution time is 0.20 × 5 + 0.80 × 4 = 4.2 times pipelined execution time.

Question 2

Textbook Exercises 4.19 <§4.5>.

Assume that x11 is initialized to 11 and x12 is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section 4.5 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). What would the final value of register x15 be? Assume the register file is written at the beginning of the cycle and read at the end of a cycle. Therefore, an ID stage will return the results of a WB state occurring during the same cycle. See Section 4.7 and Figure 4.51 for details.

```
addi  x11, x12, 5
add   x13, x11, x12
addi  x14, x11, 15
add   x15, x11, x11
```

We can trace execution of the instruction sequence cycle-by-cycle:

Instruction	1	2	3	4	5	6	7	8
`addi x11, x12, 5`	IF	ID	EX	MEM	WB			
`add x13, x11, x12`		IF	ID	EX	MEM	WB		
`addi x14, x11, 15`			IF	ID	EX	MEM	WB	
`add x15, x11, x11`				IF	ID	EX	MEM	WB

The first instruction reads 22 from x12 at the end of cycle 2, adds 5 to get 27, and writes 27 to x11 at the beginning of cycle 5.

The second instruction reads 11 from x11 and 22 from x12 at the end of cycle 3, adds them to get 33, and writes 33 to x13 at the beginning of cycle 6.

The third instruction reads 11 from x11 at the end of cycle 4, adds 15 to get 26, and writes 26 to x15 at the beginning of cycle 7.

The fourth instruction reads 27 from x11 twice at the end of cycle 5, adds the values to get 54, and writes 54 to x15 at the beginning of cycle 8. This is the final value of x15.

Question 3

Textbook Exercises 4.20 <§4.5>.

Add NOP instructions to the code below so that it will run correctly on a pipeline that does not handle data hazards.

```
addi  x11, x12, 5
add   x13, x11, x12
addi  x14, x11, 15
add   x15, x13, x12
```

The dependencies are:

- Second instruction reads x11, which must occur after first instruction writes x11.

- Third instruction reads x11, which must occur after first instruction writes x11.
- Fourth instruction reads x13, which must occur after the second instruction writes x13.

We need to insert NOP instructions to ensure that each read occurs in the same cycle in which the corresponding write occurs.

Instruction	1	2	3	4	5	6	7	8	9	10	11
`i1: addi x11, x12, 5`	IF	ID	EX	MEM	WB						
`nop`		IF	ID	EX	MEM	WB					
`nop`			IF	ID	EX	MEM	WB				
`i2: add x13, x11, x12`				IF	ID	EX	MEM	WB			
`i3: addi x14, x11, 15`					IF	ID	EX	MEM	WB		
`nop`						IF	ID	EX	MEM	WB	
`i4: add x15, x13, x12`							IF	ID	EX	MEM	WB

- Inserting 2 nop instructions before i2 ensures that the read of x11 in i2 happens in cycle 5, the same cycle as the write in i1.
- As a consequence of the inserted nop instructions, the read of x11 in i3 happens in cycle 6, after the write in i1.
- Inserting a nop before i4 ensures that the read of x13 in i4 happens in cycle 8, the same cycle as the write in i2.

Question 4

Textbook Exercises 4.22 <§4.5>.

Consider the fragment of RISC-V assembly below:

```
sd    x29, 12(x16)
ld    x29, 8(x16)
sub   x17, 12(x16)
beqz  x17, label
add   x15, x11, x14
sub   x15, x30, x14
```

Suppose we modify the pipeline so that it has only one memory (that handles both instructions and data). In this case, there will be a structural hazard every time a program needs to fetch an instruction during the same cycle in which another instruction accesses data.

a. Draw a pipeline diagram to show were the code above will stall.

In any cycle where the memory is required for a data access, the IF stage must stall, resulting in a bubble in the pipeline.

Instruction	1	2	3	4	5	6	7	8	9	10	11	12
`sd x29, 12(x16)`	IF	ID	EX	MEM	WB							
`ld x29, 8(x16)`		IF	ID	EX	MEM	WB						
`sub x17, 12(x16)`			IF	ID	EX	MEM	WB					
				☁	☁	☁	☁	☁				
					☁	☁	☁	☁	☁			
`beqz x17, label`						IF	ID	EX	MEM	WB		
`add x15, x11, x14`							IF	ID	EX	MEM	WB	
`sub x15, x30, x14`								IF	ID	EX	MEM	WB

b. In general, is it possible to reduce the number of stalls/NOPs resulting from this structural hazard by reordering code?

No. However you reorder the code, the IF stage cannot fetch an instruction when the `sd` or the `ld` is in the MEM stage. So there will be a stall for each `sd` or `ld` instruction executed.

c. Must this structural hazard be handled in hardware? We have seen that data hazards can be eliminated by adding NOPs to the code. Can you do the same with this structural hazard? If so, explain how. If not, explain why not.

Adding NOPs doesn't help, since NOPs are instructions and so must be fetched by the IF stage. If anything, adding NOPs makes things worse by making the program code larger, requiring more memory for no performance gain.

d. Approximately how many stalls would you expect this structural hazard to generate in a typical program? (Use the instruction mix from Exercise 4.8.)

The instruction mix in Exercise 4.8 is:

R-type/I-type (non-ld)	ld	sd	beq
52%	25%	11%	12%

So loads and stores together account for 36% of instructions, and there is an IF stall corresponding to each. Assuming there are no other stalls (a brave assumption!), the number of cycles for the program would be 1.36 times as many as on a machine without the structure hazard. That's why most processors have separate memories (or separate caches) for instructions and data.