1. We are given the following latencies for each stage in our single-cycle processor.

IF	ID	ALU	MEM	WB
200ps	100ps	200ps	200ps	100ps

(a) What is the speedup obtained from pipelining?
Solution:
To determine the speedup, we need to compare the latency of instructions in the old (single-cycle) implementation vs. the latency of instructions in the new (pipelined) implementation.

Note that single-cycle instruction latency = time for a single clock cycle = time for longest possible instruction. The longest instruction is one that uses all *the given components, namely a* `lw` *(load) instruction. Hence, single-cycle instruction latency = 200 + 100 + 200 + 200 + 100 = 800ps.*

In contrast, pipelined instruction latency $\hat{=}$ time for a single clock cycle = time for longest possible stage. *The first "equality" only holds in the ideal case, i.e. when there are no stalls, we ignore the pipeline filling and draining stages, and we ignore the overhead of extra hardware needed for the pipelined datapath and control. The second equality holds because each pipeline stage must be completed within a clock cycle. The pipelined instruction latency is 200ps.*

The speedup obtained is 800ps/200ps = 4. Remember, this is in the ideal case, for the assumption we made above. The key is to remember that a pipelined machine strives to attain the same CPI as the single-cycle machine (which is 1), but achieve a much faster clock rate (faster by a factor of k, where k is the ratio of the length of a full instruction to the length of a single stage).

(b) If the time for an ALU operation can be shortened by 25% will it affect the speedup obtained from pipelining? If yes, by how much? If no, why not?
Solution:
Speeding up the ALU does not speed up the pipeline. The length of the longest stage remains the same, because IF and MEM stages still take 200 ps.

(c) What if the ALU operation now takes 25% more time?
Solution:
With the ALU modification, pipelined implementation takes:
*instr time new = (longest stage time)/cycle * 1 cycle/instr*
*= 250ps/cycle * 1 cycle/instr*
= 250ps/instr
Of course, the single cycle machine also takes longer by 50ps.
Speedup = 850/250 = 3.4
The pipeline speedup is reduced from a factor of 4 to a factor of 3.4, so lengthening the ALU stage does affect the pipelining speedup. Now the ALU is the longest stage and the cycle time must be increased to accommodate it.

2. StingyMIPS is a 5-stage pipelined implementation of MIPS *without* forwarding. Consider the following piece of code containing data hazards.
   Rewrite this code so that it does the same thing on StingyMIPS as on regular MIPS, but runs without *stalls* on StingyMIPS. A stall delays every subsequent instruction by 1 cycle.

   Initial code:

   ```
   add $1, $2, $3
   add $4, $1, $3
   add $5, $6, $3
   add $7, $8, $3
   ```

   Solution:

   ```
   add $1, $2, $3
   add $5, $6, $3
   add $7, $8, $3
   add $4, $1, $3
   ```

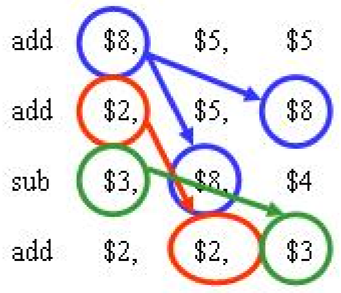
   There is a dependency between the first and second lines of code (on register $1). Without forwarding we would need to stall for two cycles. However, the trick here is to realize that the 3^{rd} and 4^{th} instructions can be brought forward, since they can be done **out of order** *and without causing any hazards.*

3. Suppose we have the following chunk of code containing only R-type instructions.

   ```
   add $8,$5,$5
   add $2,$5,$8
   sub $3,$8,$4
   add $2,$2,$3
   ```

   (a) Identify the hazards involved (draw the arrows between *dependencies* that cause data hazards).

   Figure 1. Hazards in the code above

   (b) Figure below shows the pipelined datapath with four forwarding inputs. For each dependency identified above specify which numbered forwarding path is used.
   Solution:
   The first instruction writes the results to register $8. The next instruction needs it as the second ALU input. This corresponds to the forwarding path #3. The third instruction also reads register $8 as the first ALU input. The value that it needs is already in the MEM stage. This corresponds to the forwarding path #2.
   Similarly, the forwarding of the contents of register $2 uses the forwarding path #2 and the forwarding of the contents of register $3 uses the forwarding path #3.

(c) Fill out the pipeline table assuming forwarding implemented.
Solution:
All hazards are eliminated by forwarding. Therefore, the code can be pipelined with no stalls.

Table 1. Pipeline Diagram with Forwarding

Instruction	1	2	3	4	5	6	7	8
`add $8,$5,$5`	IF	ID	EX	MEM	WB			
`add $2,$5,$8`		IF	ID	EX	MEM	WB		
`sub $3,$8,$4`			IF	ID	EX	MEM	WB	
`add $2,$2,$3`				IF	ID	EX	MEM	WB

(d) Now assume that you do not have forwarding hardware. Fill in the pipeline diagram below.
Solution:
Forwarding is not an available option. The code has to be stalled in the pipeline if it is to execute correctly. Note the fact that reads and writes of the register file can happen in the MIPS pipeline in the same cycle. Writes occur on the leading edge of the clock and due reads happen on the trailing edge. Therefore, ID and WB stages can execute in the same cycle, limiting the stall time to 2 cycles instead of 3.

Table 2. Pipeline Diagram without Forwarding

Instruction	1	2	3	4	5	6	7	8	9	10	11	12
`add $8,$5,$5`	IF	ID	EX	MEM	WB							
`add $2,$5,$8`		IF	stall	stall	ID	EX	MEM	WB				
`sub $3,$8,$4`					IF	ID	EX	MEM	WB			
`add $2,$2,$3`						IF	stall	stall	ID	EX	MEM	WB

4. A pipelined processor has k pipeline stages. Assuming no stalls, how many cycles are required to execute n instructions?
Solution:
In a k-stage pipeline, the first instruction completes after k clock cycles, and one instruction completes in every cycle after that. So, it takes k cycles to execute 1 instruction, $(k+1)$ cycles to execute two, and so on. Executing n instructions, takes $(k+n-1)$ cycles.