(c) [5 points] What is the *Cycles Per Instruction (CPI)* of the program when executed on the pipelined processor provided in part (b)?

$CPI \approx 1.5$

Explanation.
Since the code is an infinite loop, the number of cycles to fill the pipeline becomes negligible after a large number of iterations. Thus, we can consider that the throughput is one instruction every cycle. We count the number of cycles for one loop iteration. It is 15 for 10 instructions. This way, $CPI \approx \frac{15}{10} = 1.5$.

(d) [10 points] Now, assume a processor with a *multi-cycle* datapath. In this multi-cycle datapath, each instruction type is executed in the following number of cycles: 4 cycles for R-type, 5 cycles for load, 4 cycles for store, and 3 cycles for jump. What is the CPI of the program in part (a) when executed on this multi-cycle datapath? Assuming the multi-cycle datapath runs at the same clock frequency as the pipelined datapath in part (b), how much speedup does pipelining provide?

CPI:

$CPI = 4.3$

Explanation.
For the multi-cycle datapath, we have to take into account the number of cycles for each instruction type: 4 cycles for R-type, 5 cycles for load, 4 cycles for store, and 3 cycles for jump.
Thus, $CPI = \frac{4\times5+5\times4+3\times1}{10} = 4.3$.

Speedup:

Pipelining provides 287% speedup.

Explanation.
We calculate the speedup as follows:
$Speedup = \frac{CPI_{multi-cycle}}{CPI_{pipelined}} = \frac{4.3}{1.5} = 2.87$.

MIPS Instruction Summary

Opcode	Example Assembly	Semantics
add	`add $1, $2, $3`	`$1 = $2 + $3`
sub	`sub $1, $2, $3`	`$1 = $2 - $3`
add immediate	`addi $1, $2, 100`	`$1 = $2 + 100`
add unsigned	`addu $1, $2, $3`	`$1 = $2 + $3`
subtract unsigned	`subu $1, $2, $3`	`$1 = $2 - $3`
add immediate unsigned	`addiu $1, $2, 100`	`$1 = $2 + 100`
multiply	`mult $2, $3`	`hi, lo = $2 * $3`
multiply unsigned	`multu $2, $3`	`hi, lo = $2 * $3`
divide	`div $2, $3`	`lo = $2/$3, hi = $2 mod $3`
divide unsigned	`divu $2, $3`	`lo = $2/$3, hi = $2 mod $3`
move from hi	`mfhi $1`	`$1 = hi`
move from low	`mflo $1`	`$1 = lo`
and	`and $1, $2, $3`	`$1 = $2 & $3`
or	`or $1, $2, $3`	`$1 = $2 | $3`
and immediate	`andi $1, $2, 100`	`$1 = $2 & 100`
or immediate	`ori $1, $2, 100`	`$1 = $2 | 100`
shift left logical	`sll $1, $2, 10`	`$1 = $2 « 10`
shift right logical	`srl $1, $2, 10`	`$1 = $2 » 10`
load word	`lw $1, 100($2)`	`$1 = memory[$2 + 100]`
store word	`sw $1, 100($2)`	`memory[$2 + 100] = $1`
load upper immediate	`lui $1, 100`	`$1 = 100 « 16`
branch on equal	`beq $1, $2, label`	`if ($1 == $2) goto label`
branch on not equal	`bne $1, $2, label`	`if ($1 != $2) goto label`
set on less than	`slt $1, $2, $3`	`if ($2 < $3) $1 = 1 else $1 = 0`
set on less than immediate	`slti $1, $2, 100`	`if ($2 < 100) $1 = 1 else $1 = 0`
set on less than unsigned	`sltu $1, $2, $3`	`if ($2 < $3) $1 = 1 else $1 = 0`
set on less than immediate	`sltui $1, $2, 100`	`if ($2 < 100) $1 = 1 else $1 = 0`
jump	`j label`	`goto label`
jump register	`jr $31`	`goto $31`
jump and link	`jal label`	`$31 = PC + 4; goto label`

6 Pipelining [35 points]

Consider two pipelined machines implementing MIPS ISA, Machine I and Machine II:

Both machines have the following *five pipeline stages*, very similarly to the basic 5-stage pipelined MIPS processor we discussed in lectures, and *one ALU*:

1. Fetch (one clock cycle)
2. Decode (one clock cycle)
3. Execute (one clock cycle)
4. Memory (one clock cycle)
5. Write-back (one clock cycle).

Machine I does *not* implement interlocking in hardware. It assumes all instructions are independent and relies on the compiler to order instructions such that there is sufficient distance between dependent instructions. The compiler either moves other independent instructions between two dependent instructions, if it can find such instructions, or otherwise, inserts `nops`. Assume internal register file forwarding (an instruction writes into a register in the first half of a cycle and another instruction can correctly access the updated value of the same register in the next half of the cycle). Assume that the processor predicts all branches as *always-taken*.

Machine II implements data forwarding in hardware. On detection of a flow dependence, it can forward an operand from the memory stage or from the write-back stage to the execute stage. The load instruction (`lw`) can *only* be forwarded from the write-back stage because data becomes available in the memory stage but *not* in the execute stage like for the other instructions. Assume internal register file forwarding (an instruction writes into a register in the first half of a cycle and another instruction can access the updated value of the same register in the next half of the cycle). The compiler does *not* reorder instructions. Assume that the processor predicts all branches as *always-taken*.

Consider the following code segment:

```
Copy: lw   $2, 100($5)
      sw   $2, 200($6)
      addi $1, $1, 1
      bne  $1, $25, Copy
```

Initially, `$5` = 0, `$6` = 0, `$1` = 0, and `$25` = 25.

(a) [10 points] When the given code segment is executed on Machine I, the compiler has to reorder instructions and insert `nops` if needed. Write the resulting code that has *minimal modifications* from the original.

```
Copy:  lw $2, 100($5)
       addi $1, $1, 1
       nop
       sw $2, 200($6)
       bne $1, $25, Copy
```

(b) [10 points] When the given code segment is executed on Machine II, dependencies between instructions are resolved in hardware. Explain **when** data is forwarded and **which instructions** are stalled and **when** they are stalled.

In every iteration, data are forwarded for `sw` and for `bne`. The instruction `sw` is dependent on `lw`, so it is stalled one cycle in every iteration

(c) [5 points] Calculate the *machine code size* of the code segments executed on Machine I (part (a)) and Machine II (part (b)).

Machine I: Machine I - 20 bytes (because of the additional `nop`)

Machine II: Machine II - 16 bytes

(d) [7 points] Calculate the number of cycles it takes to execute the code segment on Machine I and Machine II.

Machine I:

The compiler reorders instructions and places one `nop`. This is the execution timeline of the first iteration:

1	2	3	4	5	6	7	8	9
F	D	E	M	W				
	F	D	E	M	W			
		N	N	N	N	N		
			F	D	E	M	W	
				F	D	E	M	W

9 cycles for one iteration. As there are 5 instructions in each iteration and 25 iterations, the total number of cycles is 129 cycles.

Machine II:

The machine stalls `sw` one cycle in the decode stage. This is the execution timeline of the first iteration:

1	2	3	4	5	6	7	8	9
F	D	E	M	W				
	F	D	D	E	M	W		
		F	F	D	E	M	W	
				F	D	E	M	W

9 cycles for one iteration. As there are 4 instructions in each iteration and 25 iterations, and one stall cycle in each iteration, the total number of cycles is 129 cycles.