Question 1.
This question concerns the latency of memory accesses in the classical 5-stage MIPS pipeline described in your textbook. One important design concept of pipeline architectures is the balance of pipe stages, as the cycle time is limited by the longest pipeline stage. Because the accesses to the instruction and data memory typically require more time than the function of decode, execute, and write back, your design team is deciding between the following two pipeline implementations:

1. A 5-stage pipeline IF ID EXE MEM WB with a cycle time of 1.5ns.
2. A 7-stage pipeline IF1 IF2 ID EXE MEM1 MEM2 WB with a cycle time of 0.9ns.

Part A. Assume both the implementations have data forwarding. For the following pairs of instructions with true data dependences (either through a register or a memory location), what is the number of stalls needed to be inserted in between (assume initially each pair of instruction is scheduled back to back)? Please analyze your answers for both the 5-stage and the 7-stage pipeline.

i) Arithmetic, arithmetic
ii) Load, arithmetic
iii) Arithmetic, store
iv) Store, load
v) Load, load

Part B. Your design team has decided to go with the 7-stage pipeline implementation. However, in order to save hardware, the team also decided to cut down on additional forwarding paths. Consequently, only the following forwarding paths exist in the 7-stage pipeline:

EXE/MEM1 to ID/EXE
MEM2/WB to ID/EXE

The following loop is going to be executed on the 7-stage pipeline.

```
loop:  lw R1, 100(R2)
       addi R1, R1, #1
       sw R1, 0(R2)
       addi R2, R2, #4
       sub R4, R3, R2
       bnez R4, loop
```

Assume the loop is composed of 100 iterations. The loop branch is resolved at the end of the ID
stage. Assume the branch is predicted to be always taken. For the code presented above, you are asked to identify the position of all stalls, the reason for each stall (i.e., the dependent instruction that causes the stall), as well as the steady-state CPI of the code.

**Part C.** Assume you have a branch delay slot. Can you reorder the code in Part B so as to minimize the stalls? If so, write the reordered code and compute the new CPI.

**Question 2-5**
You will be comparing uni-scalar (1-way issue) with the UltraSPARC (static superscalar) architecture.
Assumptions:
1. The branch instruction is in the BTB and the branch is always predicted taken.
2. Memory accesses are pipelined.
3. Assume an ideal fetch unit with no fetch-related penalties and assume versions of these machines that have no delay slots.
4. Assume all machines have forwarding.
5. Unless otherwise mentioned, all data accesses are L1 hits.
6. The following table lists the functional units and the latency associated with each.

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Execution Time (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>1</td>
</tr>
<tr>
<td>FP Add</td>
<td>3</td>
</tr>
<tr>
<td>FP Mult/Div</td>
<td>3</td>
</tr>
<tr>
<td>Branch</td>
<td>1</td>
</tr>
<tr>
<td>Memory Access (ld/sd)</td>
<td>2</td>
</tr>
</tbody>
</table>

7. Some instructions and the corresponding pipeline stages it goes through:
   - FP add: Fetch Decode Fadd1 Fadd2 Fadd3 WB
   - Branch: Fetch Decode Br WB
   - ALU: Fetch Decode Ex WB
   - ld: Fetch Decode M1 M2 WB
Question 2.
Your company has a vector-operation centric benchmark that is considered representative of its typical application. The company wants to choose the machine that gives them the best throughput on their applications. They wish to evaluate the machines' performance for the following piece of C-code.

```c
for(i=256;i>0;i--)
    x[i] += s;
```

For the initial evaluation, the team decides to profile the machines using the code compiled for the scalar version of the processor. The assembly code looks like the following:

**Code-1**

```assembly
loop:
    ld f0, 0(r1)    // F0 <-- x[i]
    add.f f4, f0, f2 // F4 <-- x[i] + s
    sd f4, 0(r1)    // x[i] <-- F4
    add r1, r1,#-8  // R1 <-- R1 - 8
    bnez r1, loop
```

**Part A.** Please list all true dependences (RAW), anti-dependences (WAR) and output dependences (WAW) in the first two iterations of the loop.

**Part B.** With the help of pipeline diagrams, evaluate the performance of Uni-scalar and UltraSPARC for the given code. Compute the steady-state CPI for both cases.

**Part C.** Between Uni-scalar and UltraSPARC, which machine performs better and why?
Question 3.
After profiling Code-1 on both the machines, the team observes that using code compiled for the scalar version of the processor does not give the expected throughput. The team is now considering code-reordering such that the number of dependent instructions within an iteration are minimal. Since the body of each iteration has only 4-5 instructions, they decide on unrolling the loop. They expect the wide issue width of the UltraSPARC machine to keep feeding the pipeline with instruction “groups” that are independent and can be scheduled effectively on the pipeline.

Code-2

```
loop:  ld  f0, 0(r1)
       add.f f4, f0, f2
       sd  f4, 0(r1)
       ld  f6, -8(r1)
       add.f f8, f6, f2
       sd  f8, -8(r1)
       ld  f10, -16(r1)
       add.f f12, f10, f2
       sd  f12, -16(r1)
       ld  f14, -24(r1)
       add.f f16, f14, f2
       sd  f16, -24(r1)
       add  r1, r1, #-32
       bnez r1, loop
```

Part A. Please list all true dependences (RAW), anti-dependences (WAR) and output dependences (WAW). How have they changed from Code-1?

Question 4.
Based on the findings from Q3, the team observes that though simple loop unrolling reduces the # of dependences, the distance between the producer instruction and the consumer does not increase. (The greater the distance between the producer and consumer instructions, the better throughput we achieve. The independent instructions in between can be executed while the consumer is waiting on the producer instruction). This limited the performance throughput of the in-order superscalar machine. The team now decides to re-order the code such that dependent instructions are placed as far as possible from each other. The rearranged code is as follows:

```
Code-3
loop:   ld   f0, 0(r1)  
     ld   f6, -8(r1) 
     ld   f10, -16(r1)  
     ld   f14, -24(r1)  
     add.f f4, f0, f2  
     add.f f8, f6, f2  
     add.f f12, f10, f2 
     add.f f16, f14, f2 
     sd   f4, 0(r1)  
     sd   f8, -8(r1) 
     sd   f12, -16(r1)  
     sd   f16, -24(r1)  
     add   r1, r1, #32 
     bnez  r1, loop
```

Part A. Please list all true dependences (RAW), anti-dependences (WAR) and output dependences (WAW) in the first two iterations of the loop. How have they changed from Code-2?

Part B. With the help of pipeline diagrams, evaluate the performance of Uni-scalar and UltraSPARC for Code-3. Quantify the performance in terms of CPI.

Part C. The code-reordering was done in favor of the in-order superscalar machine. How was the performance of the uniscalor machine affected by this (that is, in moving from Code-2 to Code-3)?
Question 5.
Based on the previous experiments, the team notices a trend. As the number of dependences reduces, the program throughput increases. The team decides to re-structure Code-3 such that intra-iteration dependence is minimal. The team transforms the code using the software-pipelining technique. (Note: Software pipelining is an alternative to loop unrolling, which can attain higher performance with fewer instructions. It merges instructions from different loop iterations into the loop body, so as to space them out and eliminate stalls. In software pipelining many values are live across the loop edge, while in unrolling, values typically spill to memory. Logically, software pipelined execution resembles a hardware pipeline).

Code-4

```c
//*** ramp up code
ld f3, 0(r1)
add.f f4,f3,f2
ld f5, -8(r1)
add r1,r1,-16
beqz r1, ramp_down

//*** steady state loop
loop:
  ld f3, 0(r1)
  add.f f6,f5,f2
  sd f4, 16 (r1)
  add.f f4,f3,f2
  ld f5, -8(r1)
  add r1,r1,-16
  sd f6, 24 (r1)
  bnez r1, loop

//*** ramp down code
ramp_down:
  add.f f6,f5,f2
  sd f4, 16 (r1)
  sd f6, 8(r1) // it’s 8 because we did not add -16 this time around
```

Part A. Please list all true dependences (RAW), anti-dependences (WAR) and output dependences (WAW) in the first two iterations of the loop. How have they changed from Code-3?

Part B. Explain how the performance on the two machines is affected going from Code-3 to Code-4. Give reasons why.