Question 1
Consider the following piece of code
The C-code would be
```c
int *p;
for (; p != pEnd; p++) {
    *p = 2(*p+12);
}
```
1) Loop:

1)     lw  $t1, 0(a0)
2)     addi $t1, $t1,#12  // t1 = t1+12;
3)     add $t1, $t1,$t1  // t1 = t1+t1;
4)     sw  $t1, 0($a0)
5)     add $a0, $a0,4
6)     bne $a0, $a1, Loop

Assume you have a processor shown below, with 5 stages of pipelining. Assume that the First Instruction
IF stage is done on clock cycle 1 and the pipeline is empty at the beginning. Assume branching decisions
can be made immediately after the ID stage at the end of clock cycle 2.

![Pipeline Diagram]

1.a: [1 point] Assume $a0 = 12$; and $a1 = 44$ at the beginning. How many times do you expect to go
through the loop?

$$\frac{44-12}{4} = 8$$

1.b: [2 points] Assuming no forwarding of any kind, at the end of which clock cycle does the register $t1
contain *(12) (that is, the data in the memory location 12)?

At the end of 5th cycle, which is the WB stage

1.c: [4 points] List the various data hazards in this loop.

1)—2) : $t1$
2)—3) : $t1$
3)—4) : $t1$
5)—6) : $a0$

1.d: [2 points] Which of these data hazards can you fix by forwarding?

2)—3) : $t1$
3)—4) : $t1$
5)—6) : $a0$

1.e: [1 point] Which data hazards not fixable by forwarding?

1)—2) : $t1$ (load)
**Question 2**

Consider the following piece of code

The c-version is

```c
For(i=0; i<100; i++) { b[i] = a[i]; }
```

In assembly this was created by your compiler as

(assume $a0 contains a, and $a1 contains b)

```assembly
addi $t5, $a0, #100
Loop:
    lw $t1, 0($a0)
    sw $t1, 0($a1)
    add $a0, $a0,4
    add $a1, $a1,4
    bne $a0, $t5, Loop
```

Assume you have a processor shown below, with 5 stages of pipelining. Assume that the First Instruction IF stage is done on clock cycle 1 and the pipeline is empty at the beginning. Assume branching decisions can be made immediately after the ID stage at the end of clock cycle 2.

Assume data forwarding has been implemented.

Assume no branch prediction.

Assume no speculative execution and a branch penalty of 1 cycle.

![Processor Diagram]

2.a: [2 points] Describe the data hazards in the code, as generated.

Between lw and sw on $t1

2.b: [2] Describe the control hazards due to the branch decision by bne

2.c: [4] Fix the above data and control hazards by inserting no-ops in the code. How many no-ops. Write your code and reasons for the no-ops clearly.

   1. Add one NOP between lw and sw
   2. Add another NOP after bne

2.d: [4 points] Describe the operations that go on, in each clock cycle (and for each instruction) in the following table. Note that the last line is the lw from the next round of the loop.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.e: [3] How many clock cycles does it take to go from the “IF” of the load word, to the “IF” of the next lw executed?

7 cycles

2.f[5] Can you do better by reordering the code? (Note that the goal is to remove all no-ops). Write your new code and explain it carefully.

Addi
Loop:  
Lw $t1 = $a0
Add $a0 += 4
Sw $a1 = $t1
Bne
Add $a1 += 4

2.g[2] Explain the difference between static and dynamic branch prediction?

Static branch prediction makes fixed prediction of a branch to either taken or not taken;
Dynamic branch prediction makes prediction based on previous history situation, by looking at history table.

2.h[3] If you knew a-priori that you need to run the code above, what would your preferred static branch prediction policy look like? Explain in words only, we do not need code.

Since the loop will iterate 100 times, it’s preferred to predict the branch always taken, which occupies most of the cases.

Consider the situation that you have a superscalar machine with multiple (in this case two) pipelines, each with 5 pipeline stages, as shown in the figure below.

Both pipelines are running in parallel. And they are all using the same memory.

Assume that groups of two instructions are being executed every cycle, the top one in the top pipeline and the second one in the lower pipeline. (This is called “dispatching two instructions per cycle”)

2.i[7] Restructure the code above to use all two pipes? (you can use either the original code in the question, or your modified one from 2.f). Make sure you mark each line of the code with “what pipe is it being executed in”.

Note: call the 2 pipelines A & B, the 2 instructions executed in A & B at the same time cannot have dependency. In order to make fill A & B as much as possible, we unroll the loop once.
The unrolled loop:

```
addi $t5, $a0, #100
Loop:
    lw $t1, 0($a0)
    sw $t1, 0($a1)
    lw $t2, 4($a0)
    sw $t2, 4($a1)
    add $a0, $a0,8
    add $a1, $a1,8
    bne $a0, $t5, Loop
```

Then we fill the above instructions into the two pipeline:

<table>
<thead>
<tr>
<th>Pipeline A</th>
<th>Pipeline B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
</tr>
<tr>
<td>addi $t5, $a0, #100</td>
<td>lw $t2, 4($a0)</td>
</tr>
<tr>
<td>lw $t1, 0($a0)</td>
<td>sw $t1, -4($a1)</td>
</tr>
<tr>
<td>add $a0, $a0,8</td>
<td>add $a1, $a1,8</td>
</tr>
<tr>
<td>bne $a0, $t5, Loop</td>
<td>sw $t1, -8($a1)</td>
</tr>
<tr>
<td>sw $t2, -4($a1)</td>
<td>nop</td>
</tr>
</tbody>
</table>

2. Explain what kind of performance improvement do you get? (This can be in “number of cycles per loop” or similar criteria, but make sure you explain your criteria and meaning carefully, )

Without superscalar, one iteration of the unrolled loop needs 8 cycles (with an additional nop and rearranged order). With superscalar implementation, one iteration takes 4 cycles.