Problem 1

Here is an unusual loop. First, list the dependences (output, anti, and true) and then rewrite the loop so that it is parallel:

```c
for (i = 1; i < 100; i = i + 1) {
    a[i] = b[i] + c[i]; /* S1 */
    b[i] = a[i] + d[i]; /* S2 */
    a[i+1] = a[i] + e[i]; /* S3 */
}
```

Problem 2

H&P 3.3

Problem 3

H&P 3.5

Problem 4

H&P 3.6
Problem 5

Scoreboarding is a simple dynamic scheduling technique traceable to the CDC 6600. In the simplest form of scoreboarding, instructions are issued in order – i.e., instructions pass one at a time through the fetch, decode, and issue stages in program order. If one instruction gets stalled, all instructions following it get stalled as well. Once past the issue stage, however, an instruction enters one of several execution units, and is allowed to execute concurrently with instructions already executing in other units. Since functional units may have different latencies, instructions are allowed to commit (i.e., produce irreversible changes to the processor state) out of order. (In most cases, committing involves writing either to the register file or to memory). In more complex forms of scoreboarding, we allow out-of-order issue as well – i.e., instructions are still fetched in program order, but if an instruction gets stalled, other instructions following it are allowed to be issued as long as they do not have any problematic dependencies. As we shall see, out-of-order completion and out-of-order issue both create new types of possible hazards. Scoreboarding is one way to enforce proper dependence relationships between instructions despite these hazards. In this problem, we will use a very simple scoreboarding technique to handle both in-order and out-of-order issue.

We consider a simplified model of a typical floating-point pipeline, shown in Figure 1. Unlike the integer pipeline we have seen so far, this floating-point pipeline has several execution units with different functionalities. Here, we have 1 FP add unit, 2 FP multiply units, and 1 FP divide unit. Note that since these units can be used independently of one another, we can maximize performance by doing different operations in parallel. As long as two instructions use different functional units and have no data dependencies (or anti-dependencies), we can execute them in parallel to save time.

Checking for dependencies and reading the data operands from the register file are done in the issue stage. An instruction is said to be issued when it has successfully read its operands and is passed-on to the appropriate functional unit. Data hazards, such as RAW hazards, may cause
an instruction to be stalled (i.e., not issued right away). A stalled instruction can only be issued when all hazards involving it are cleared.

The functional units perform different functions and have different latencies. For this problem, we assume that the functional unit latencies are unknown and variable. We also assume that the functional units are not pipelined. We maintain a bit-vector, busy, with one bit corresponding to each functional unit. Whenever functional unit $X$ is in use, then busy[$X$] is set to 1. When the functional unit is finished with its computation, then busy[$X$] is reset to 0. An instruction using functional unit $X$ can only be issued if busy[$X$] is 0 (i.e., when there is no structural hazard).

Scoreboarding is used to determine whether an instruction is issued or stalled. For this problem, our scoreboard consists of the busy vector, and a 32-bit vector, WP (Write Pending) which has 1 bit corresponding to each of the 32 FPRs. WP[i] is set to 1 if and only if there is some instruction that has been issued but has not completed yet or is stalled waiting to issue that has FPR $i$ as its destination register. As we shall see below, the vector WP is used in the issue stage to determine whether or not an instruction can be issued safely without causing a hazard. We assume for now that this scoreboard is maintained by a “magic” bookkeeper which guarantees that the scoreboard bits are always correct and up-to-date.

**Part A: In-Order Issue**

With in-order instruction issue, when an instruction flows from the decode stage to the issue stage, we first check the scoreboard to decide if there is any data or structural hazard. If there is no hazard, the instruction is issued to the appropriate FP functional unit. However, if a hazard is detected, the instruction is stalled, and all instructions following it are also stalled.

1. A Write-after-Write (WAW) hazard happens when an earlier instruction can overwrite a register that already has been updated by a later instruction. Can a WAW hazard happen in this FP datapath? Explain briefly.

2. A Write-after-Read (WAR) hazard happens when a later instruction can modify a register before an earlier instruction has read the original register contents as its operand. Can a WAR hazard happen in this FP datapath? Explain briefly.

3. Suppose the issue stage gets a floating-point multiply instruction from the decode stage:

   $$\text{MULTF Rd, Rs1, Rs2}$$

   where Rd is the destination FP register, and Rs1 and Rs2 are source FP registers.

   Write a boolean equation (in terms of appropriate scoreboard bits) to specify the STALL signal. Note that with in-order instruction issue, the STALL signal stalls not only this MULTF instruction to be issued, but all following instructions as well.

4. Show how imprecise interrupts can occur when a FP arithmetic exception happens.

**Part B: Out-of-Order Issue**
With in-order instruction issue, when an instruction is stalled, all the following instructions are also stalled – even if some of them may have neither data nor structure hazard with the instructions that are currently executing. For example, consider the following code sequence:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>MULTF</td>
<td>F1, F2, F3</td>
</tr>
<tr>
<td>I2</td>
<td>ADDF</td>
<td>F4, F1, F5</td>
</tr>
<tr>
<td>I3</td>
<td>DIVF</td>
<td>F6, F7, F8</td>
</tr>
</tbody>
</table>

Suppose instruction I2 is stalled due to a RAW hazard with instruction I1. With in-order issue, instruction I3 will not be issued even though it can be executed safely.

Out-of-order issue tries to avoid this inefficiency by allowing instructions to be issued in an order different from the order in which they are fetched. In the above example, instruction I3 can be executed while I2 is stalled. To hold stalled instructions, one possible method is to use a buffer called *instruction window* in the issue stage. Whenever necessary (e.g. the scoreboard is modified), the magic bookkeeper checks the instruction window, and may decide to extract and issue certain stalled instructions if it is safe to do so.

1. With out-of-order issue, you may have to worry about certain kinds of data hazards that do not exist with in-order issue. Identify these new kinds of hazards and demonstrate how they can arise.

2. Suppose we expand the scoreboard with another 32-bit vector, \( RP \) (Read Pending). \( RP[i] \) is set to 1 if a stalled instruction has FPR \( i \) as a source register. Our magic bookkeeper is responsible for maintaining this RP vector.

   Again suppose the issue stage gets the following instruction from the decode stage:

   \[
   \text{MULTF Rd, Rs1, Rs2}
   \]

   Write the boolean equation to specify the \( \text{STALL} \) signal. Note that with out-of-order instruction issue, the \( \text{STALL} \) signal, if asserted, only stalls this MULTF instruction.