A.1  [15/15/15] <A.2> Use the following code fragment:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1,0(R2)</td>
<td>1</td>
</tr>
<tr>
<td>DADDI R1,R1,#1</td>
<td>1</td>
</tr>
<tr>
<td>SD 0(R2),R1</td>
<td>1</td>
</tr>
<tr>
<td>DADDI R2,R2,#4</td>
<td>1</td>
</tr>
<tr>
<td>DSUB R4,R3,R2</td>
<td>1</td>
</tr>
<tr>
<td>BNEZ R4,Loop</td>
<td>1</td>
</tr>
</tbody>
</table>

; load R1 from address 0+R2
; R1=R1+1
; store R1 at address 0+R2
; R2=R2+4
; R4=R3-R2
; branch to Loop if R4!=0

Assume that the initial value of R3 is R2 + 396.

Throughout this exercise use the classic RISC five-stage integer pipeline (see Figure A.1) and assume all memory accesses take 1 clock cycle.

(a)  [15] <A.2> Show the timing of this instruction sequence for the RISC pipeline without any forwarding or bypassing hardware but assuming a register read and a write in the same clock cycle “forwards” through the register file, as in Figure A.6. Use a pipeline timing chart like Figure A.6. Assume that the branch is handled by flushing the pipeline. If all memory references take 1 cycle, how many cycles does this loop take to execute?

(b)  [15] <A.2> Show the timing of this instruction sequence for the RISC pipeline with normal forwarding and bypassing hardware. Use a pipeline timing chart like Figure A.6. Assume that the branch is handled by predicting it as not taken. If all memory references take 1 cycle, how many cycles does this loop take to execute?

(c)  [15] <A.2> Assume the RISC pipeline with a single-cycle delayed branch and normal forwarding and bypassing hardware. Schedule the instructions in the loop including the branch delay slot. You may reorder instructions and modify the individual instruction operands, but do not undertake other loop transformations that change the number or opcode of the instructions in the loop (that’s for Chapter 4!). Show a pipeline timing diagram and compute the number of cycles needed to execute the entire loop.

A.5  [12/13/20/20/15/15] <A.2, A.3> For these problems, we will explore a pipeline for a register-memory architecture. The architecture has two instruction formats: a register-register format and a register-memory format. There is a single-memory addressing mode (offset + base register).

There is a set of ALU operations with format:

\[ \text{ALUop Rdest, Rsrt, Rsrt} \]

or

\[ \text{ALUop Rdest, Rsrt, MEM} \]
where the ALUOp is one of the following: Add, Subtract, And, Or, Load (Rsrc\textsubscript{1} ignored), Store. Rs or Rdest are registers. MEM is a base register and offset pair.

Branches use a full compare of two registers and are PC-relative. Assume that this machine is pipelined so that a new instruction is started every clock cycle. The following pipeline structure—similar to that used in the VAX 8700 micropipeline [Clark 1987]—is

<table>
<thead>
<tr>
<th>IF</th>
<th>RF</th>
<th>ALU1</th>
<th>MEM</th>
<th>ALU2</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>RF</td>
<td>ALU1</td>
<td>MEM</td>
<td>ALU2</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>RF</td>
<td>ALU1</td>
<td>MEM</td>
<td>ALU2</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>RF</td>
<td>ALU1</td>
<td>MEM</td>
<td>ALU2</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>RF</td>
<td>ALU1</td>
<td>MEM</td>
<td>ALU2</td>
<td>WB</td>
</tr>
</tbody>
</table>

The first ALU stage is used for effective address calculation for memory references and branches. The second ALU cycle is used for operations and branch comparison. RF is both a decode and register-fetch cycle. Assume that when a register read and a register write of the same register occur in the same clock the write data is forwarded.

a. [12] <A.2> Find the number of adders needed, counting any adder or incrementer; show a combination of instructions and pipe stages that justify this answer. You need only give one combination that maximizes the adder count.

b. [13] <A.2> Find the number of register read and write ports and memory read and write ports required. Show that your answer is correct by showing a combination of instructions and pipeline stage indicating the instruction and the number of read ports and write ports required for that instruction.

c. [20] <A.4> Determine any data forwarding for any ALUs that will be needed. Assume that there are separate ALUs for the ALU1 and ALU2 pipe stages. Put in all forwarding among ALUs needed to avoid or reduce stalls. Show the relationship between the two instructions involved in forwarding using the format of the table in Figure A.22 but ignoring the last two columns. Be careful to consider forwarding across an intervening instruction, for example,

```
ADD    R1, ...
any instruction
ADD    ..., R1, ...
```

d. [20] <A.3> Show all data forwarding requirements needed to avoid or reduce stalls when either the source or destination unit is not an ALU. Use the same format as Figure A.22, again ignoring the last two columns. Remember to forward to and from memory references.

e. [15] <A.3> Show all the remaining hazards that involve at least one unit other than an ALU as the source or destination unit. Use a table like that in Figure A.21, but listing the length of hazard in place of the last column.
f. [15] <A.2> Show all control hazard types by example and state the length of the stall. Use a format like Figure A.11, labeling each example.

A.6 [12/13/13/15/15] <A.1, A.2, A.3> We will now add support for register-memory ALU operations to the classic five-stage RISC pipeline. To offset this increase in complexity, all memory addressing will be restricted to register indirect (i.e., all addresses are simply a value held in a register: no offset or displacement may be added to the register value). For example, the register-memory instruction ADD R4, R5, (R1) means add the contents of register R5 to the contents of the memory location with address equal to the value in register R1 and put the sum in register R4. Register-register ALU operations are unchanged. Answer the following for the integer RISC pipeline.

a. [12] <A.1> List a rearranged order of the five traditional stages of the RISC pipeline that will support register-memory operations implemented exclusively by register indirect addressing.

b. [13] <A.2, A.3> Describe what new forwarding paths are needed for the rearranged pipeline by stating the source, destination, and information transferred on each needed new path.

c. [13] <A.2, A.3> For the reordered stages of the RISC pipeline, what new data hazards are created by this addressing mode? Give an instruction sequence illustrating each new hazard.

d. [15] <A.3> List all ways that the RISC pipeline with register-memory ALU operations can have a different instruction count for a given program than the original RISC pipeline. Give a pair of specific instruction sequences, one for the original pipeline and one for the rearranged pipeline, to illustrate each way.

e. [15] <A.3> Assume all instructions take 1 clock cycle per stage. List all ways that the register-memory RISC can have a different CPI for a given program as compared to the original RISC pipeline.

A.7 [10/15/15] <A.2> Scheduling branch delay slots (see the three ways in Figure A.14) can improve performance. Assume a single branch delay slot and an instruction execution pipeline that determines branch outcome in the second stage.

a. [10] <A.2> For a delayed branch instruction, what is the penalty for each branch delay slot scheduling scheme if the branch is taken and if it is not taken, and what condition, if any, must be satisfied to ensure correct execution?

b. [15] <A.2> A cancel-if-not-taken branch instruction (also called branch likely and implemented in MIPS) does not execute the instruction in the delay slot if the branch is not taken. Thus, a compiler need not be as conservative when filling the delay slot. For each branch delay slot scheduling scheme, what is the penalty if the branch is taken and if it is not taken, and what condition, if any, must be satisfied to ensure correct execution?

c. [15] <A.2> Assume that an instruction set has a delayed branch and a cancel-if-not-taken branch. When should a compiler use each branch instruction and from where should the slot be filled?
A.13 [25] <A.8> It is critical that the scoreboard be able to distinguish RAW and WAR hazards, because a WAR hazard requires stalling the instruction doing the writing until the instruction reading an operand initiates execution, but a RAW hazard requires delaying the reading instruction until the writing instruction finishes—just the opposite. For example, consider the sequence

\[
\begin{align*}
\text{MUL.D} & : F0,F6,F4 \\
\text{DSUB.D} & : F8,F0,F2 \\
\text{ADD.D} & : F2,F10,F2
\end{align*}
\]

The DSUB.D depends on the MUL.D (a RAW hazard) and thus the MUL.D must be allowed to complete before the DSUB.D; if the MUL.D were stalled for the DSUB.D due to the inability to distinguish between RAW and WAR hazards, the processor will deadlock. This sequence contains a WAR hazard between the ADD.D and the DSUB.D, and the ADD.D cannot be allowed to complete until the DSUB.D begins execution. The difficulty lies in distinguishing the RAW hazard between MUL.D and DSUB.D, and the WAR hazard between DSUB.D and ADD.D.

To see just why the three-instruction scenario is important, trace the handling of each instruction stage by stage through Issue, Read Operands, Execute, and Write.

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP multiply</td>
<td>FP ALU op</td>
<td>6</td>
</tr>
<tr>
<td>FP add</td>
<td>FP ALU op</td>
<td>4</td>
</tr>
<tr>
<td>FP multiply</td>
<td>FP store</td>
<td>5</td>
</tr>
<tr>
<td>FP add</td>
<td>FP store</td>
<td>3</td>
</tr>
<tr>
<td>Integer operation (including load)</td>
<td>Any</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure A.56 Pipeline latencies where latency is number of cycles between producing and consuming instruction.

Result. Assume that each scoreboard stage other than Execute takes 1 clock cycle. Assume that the MUL.D instruction requires 3 clock cycles to execute and that the DSUB.D and ADD.D instructions each take 1 cycle to execute. Finally, assume the processor has two multiply function units and two add function units. Present the trace as follows.

Step 1. Make a table with the column headings Instruction, Issue, Read Operands, Execute, Write Result, and Comment. In the first column, list the instructions in program order (be generous with space between instructions; larger table cells will better hold the results of your analysis). Start the table by writing a 1 in the Issue column of the MUL.D instruction row to show that MUL.D completes the Issue stage in clock cycle 1. Now, fill in the stage columns of the table through the cycle at which the scoreboard first stalls an instruction.

Step 2. For a stalled instruction write the words “waiting at clock cycle X,” where X is the number of the current clock cycle, in the appropriate table column to show that the scoreboard is resolving an RAW or WAR hazard by stalling that stage. In the Comment column, state what type of hazard and what dependent instruction is causing the wait.

Step 3. Adding the words “completes with clock cycle Y” to a “waiting” table entry, fill in the rest of the table through the time when all instructions are complete. For an instruction that stalled, add a description in the Comments column telling why the wait ended when it did and how deadlock was avoided. (Hint: Think about how WAW hazards are prevented and what this implies about active instruction sequences.) Note the completion order of the three instructions as compared to their program order.