Exercises

Solutions to the "starred" exercises appear in Appendix B.

3.1 [12/12/12/15] <3.1, 3.2, 3.3, 3.6> For the following code fragment, assume that all data references are shown, that all values are defined before use, and that only b and c are used again after this segment. You may ignore any possible exceptions. The individual statements are numbered to provide an easy reference.

```
1. if (a > c) {
2.       d = d + 5;
3.       a = b + d + e;}
  } else {
4.       e = e + 2;
5.       f = f + 2;
6.       c = c + f;
  }
7. b = a + f;
```

a. [12] <3.1> List the control dependences. For each control dependence, tell whether the dependent statement can be scheduled before the if statement based on the data references.

b. [12] <3.1, 3.2, 3.3, 3.6> Assume a dynamically scheduled, multiple-issue processor without speculation and with a window that is holding the entire code fragment. Find the data dependences and use this information to make a list of the successive groups of statements that are issued together.

c. [12] <3.1> It is given that only the values b and c are live after the code segment. If it is known that a value is not live at some point in the code, then the statement that defines that value can be deleted without changing the program meaning. Find any values that are not live within the given code fragment, and list the statement(s) that a compiler with this information could delete.

d. [15] <3.1, 3.2, 3.3, 3.6> How does the result for part (c) affect the ILP achieved by the processor of part (b)? What does this illustrate about measuring computer performance factors such as ILP, making hardware design choices, and compiler technology?

✪ 3.2 [15/15] <3.1, 3.2> This chapter examines hardware approaches for exploiting instruction-level parallelism. This exercise asks how well hardware can find and exploit ILP.

Consider the following four MIPS code fragments each containing two instructions:

i.
```
DADDI R1,R1,#4
LD    R2,7(R1)
```

ii.
```
DADD  R3,R1,R2
SD    R2,7(R1)
```

iii.
```
SD   R2,7(R1)
S.D  F2,200(R7)
```

iv.
```
BEZ  R1,place
SD   R1,7(R1)
```

a. [15] <3.1> For each code fragment (i) to (iv) identify each type of dependence that exists or that may exist (a fragment may have no dependences) and describe what data flow, name reuse, or control structure causes or would cause the dependence. For a dependence that may exist, describe the source of the ambiguity and identify the time at which that uncertainty is resolved.

b. [15] <3.2> For each code fragment, discuss whether dynamic scheduling is, may be, or is not sufficient to allow out-of-order execution of the fragment.

3.3 [12/15] <3.1, A> Consider the following MIPS assembly code.

```
LD   R1,45(R2)
DADD R7,R1,R5
DSUB R8,R1,R6
OR   R9,R5,R1
BNEZ R7,target
DADD R10,R8,R5
XOR  R2,R3,R4
```

a. [12] <3.1> Identify each dependence by type; list the two instructions involved; identify which instruction is dependent; and, if there is one, name the storage location involved.

b. [15] <3.1, A> Use information about the MIPS five-stage pipeline from Appendix A and assume a register file that writes in the first half of the clock cycle and reads in the second half-cycle forwarding. Which of the dependences that you found in part (a) become hazards and which do not? Why?

3.4 [20/15/20/15/15] <1, 2, 3.1> Stack and accumulator instruction set architectures (see Figures 2.1 and 2.2) were common in the early days of electronic computers. Processors for both can be implemented with a small number of logic gates. With the technology of the time, vacuum tubes, only small circuits had reasonable cost and acceptable reliability. Solid-state integration of extremely inexpensive, highly reliable transistors has made very large circuits feasible. Current processors use the register-register (load-store) instruction set architecture, which requires more gates to build. Increased feasible circuit size could explain growing use of the load-store ISA, but is that sufficient to explain the absence of new generations of stack and accumulator processors? The following explores other possible reasons: cost, clock speed, dependences, and opportunity for instruction scheduling.

a. [20] <1, 2> Chapter 1 presents a model for integrated circuit die cost that shows cost escalates rapidly with increasing die area. The essential core circuit of a stack processor (ALU, two stack positions, control logic, and data paths) or of an accumulator processor (ALU, accumulator, control logic, and

data paths) is smaller than the essential core circuit of a register-register processor (ALU, set of at least three registers for operands and result, control logic, and data paths). For current processors, investigate how much die area is devoted to functions other than the essential core execution logic and what reliability is achieved. Discuss how much cost and reliability advantage stack or accumulator designs would have with respect to current load-store processor designs if they could cut essential core execution logic area by 10%, by 50%, and by more than 99%.

b. [15] <2> Consider the block diagram circuits shown in Figure 2.1 for the stack, accumulator, and load-store ISAs. How might achievable clock speeds differ among the processor circuits?

c. [20] <2, 3.1> Consider the following simple computation:

```
C = A + B
E = D - C
```

Using Figure 2.2 as a start, write instruction sequences to perform the computation on stack, accumulator, and load-store architectures. Assume that the stack architecture subtracts the next-to-top-of-stack from the top-of-stack. Assume that the accumulator architecture subtracts a memory location from the accumulator. Use distinct registers for all data values in the load-store sequence. Carefully examine your code to find the dependences. For each dependence, list its type, the two instructions involved, which instruction is dependent, and the storage location involved. Rank the architectures by their potential to exploit instruction-level parallelism.

d. [15] <2, 3.1> For your code in part (c), describe the different instruction schedules that a compiler could produce for each architecture.

e. [15] <1, 2, 3.1> Which of the factors examined in parts (a) through (d)—cost, clock speed, dependences, and opportunity for instruction scheduling—explain the dominance of today's processor designs by the register-register (load-store) ISA and why are these factors critical?

✪ 3.5 [15/15/15/15/12] <3.3> This exercise examines the basic Tomasulo algorithm. Answer the following questions based upon the tabular description of the algorithm given in Figure 3.5.

a. [15] <3.3> For each row of the table, state (1) whether that row could apply simultaneously to more than one program instruction (ILP); (2) whether the Tomasulo algorithm allows ILP for that row; (3) if ILP is not allowed, how that restriction is enforced; and (4) if ILP is allowed, what if anything could prevent achieving the maximum ILP present in the program.

b. [15] <3.3> Which one of `rs` and `rt` holds the name of the base address register for a load or store instruction? Explain your answer in sufficient detail to be a proof.

c. [15] <3.3> In the terminology of the table, write the function(s) performed by the Address unit in Figure 3.2.

d. [15] <3.3> Write the table entries required to support integer ALU operation instructions.

e. [12] <3.3> Consider how branch instructions affect the instruction processing described in the table. Show the modifications to the table necessary to take the presence of branch instructions in the program into account.

3.6 [20/25/25] <3.2, 3.3, 3.6, 3.7> In this exercise, we will look at how variations on Tomasulo's algorithm perform when running a common vector loop. The loop is the so-called DAXPY loop (*d*ouble-precision *aX* *p*lus *Y*) and is the central operation in Gaussian elimination. The following code implements the operation Y = aX + Y for a vector of length 100. Initially, R1 = 0 and F0 contains a.

```
foo:    L.D       F2,0(R1)      ;load X(i)
        MUL.D     F4,F2,F0      ;multiply a*X(i)
        L.D       F6,0(R2)      ;load Y(i)
        ADD.D     F6,F4,F6      ;add a*X(i) + Y(i)
        S.D       F6,0(R2)      ;store Y(i)
        DADDUI    R1,R1,#8      ;increment X index
        DADDUI    R2,R2,#8      ;increment Y index
        DSGTUI    R3,R1,#800    ;test if done
        BEQZ      R3,foo        ;loop if not done
```

The pipeline function units are described in Figure 3.62.

Assume the following:

- Function units are not pipelined.
- There is no forwarding between function units; results are communicated by the CDB.
- The execution stage (EX) does both the effective address calculation and the memory access for loads and stores. Thus the pipeline is IF / ID / IS / EX / WB.
- Loads take 1 clock cycle.
- The issue (IS) and write result (WB) stages each take 1 clock cycle.

FU type	Cycles in EX	Number of FUs	Number of reservation stations
Integer	1	1	5
FP adder	4	1	3
FP multiplier	15	1	2

Figure 3.62 Information about pipeline function units.

- There are 5 load buffer slots and 5 store buffer slots.
- Assume that the BNEQZ instruction takes 0 clock cycles.

a. [20] <3.2, 3.3> For this problem use the single-issue Tomasulo MIPS pipeline of Figure 3.2 with the pipeline latencies from Figure 3.63. Show the number of stall cycles for each instruction and what clock cycle each instruction begins execution (i.e., enters its first EX cycle) for three iterations of the loop. How many clock cycles does each loop iteration take? Report your answer in the form of a table like that in Figure 3.25.

b. [25] <3.6> Use the MIPS code for DAXPY above and a fully pipelined FPU with the latencies of Figure 3.63. Assume a two-issue Tomasulo's algorithm for the hardware with one integer unit taking one execution cycle (a latency of 0 cycles to use) for all integer operations. Show the number of stall cycles for each instruction and what clock cycle each instruction begins execution (i.e., enters its first EX cycle) for three iterations of the loop. Show your answer in the form of a table like that in Figure 3.25.

c. [25] <3.7> Using the MIPS code for DAXPY above, assume Tomasulo's algorithm with speculation as shown in Figure 3.29. Assume the latencies shown in Figure 3.63. Assume that there are separate integer function units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table as in Figure 3.34 for the first three iterations of this loop.

3.7 [15/15] <3.2, 3.3> Tomasulo's algorithm has a disadvantage: Only one result can complete per clock, per CDB.

a. [15] <3.2> Use the hardware configuration from Figure 3.2 and the FP latencies from Figure 3.63. Find a code sequence of no more than 10 instructions where Tomasulo's algorithm must stall due to CDB contention. Indicate where this occurs in your sequence.

b. [15] <3.2, 3.3> Generalize your result from part (a) by describing the characteristic of any code sequence that will eventually experience structural hazard stall given n CDBs.

Instruction producing result	Instruction using result	Latency in clock cycles
FP multiply	FP ALU op	6
FP add	FP ALU op	4
FP multiply	FP store	5
FP add	FP store	3
Integer operation (including load)	Any	0

Figure 3.63 Pipeline latencies where latency is number of cycles between producing and consuming instruction.

3.8 [20] <3.4> Branch-prediction buffers are indexed using the low-order address bits of the branch instruction. Assume now that some other subset of address bits is chosen. Discuss the effects on buffer operation.

✪ 3.9 [15/15/15] <3.4> Increasing the size of a branch-prediction buffer means that it is less likely that two branches in a program will share the same predictor. A single predictor predicting a single branch instruction is generally more accurate than is that same predictor serving more than one branch instruction.

a. [15] <3.4> List a sequence of branch taken and not taken actions to show a simple example of 1-bit predictor sharing that reduces misprediction rate.

b. [15] <3.4> List a sequence of branch taken and not taken actions that show a simple example of how sharing a 1-bit predictor increases misprediction.

c. [15] <3.4> Discuss why the sharing of branch predictors can be expected to increase mispredictions for the long instruction execution sequences of actual programs.

3.10 [15] <3.4> Construct a version of the table in Figure 3.13 on page 203 assuming the 1-bit predictors are initialized to NT, the correlation bit is initialized to T, and the value of **d** (leftmost column of the table) alternates 1, 2, 1, 2. Also, note and count the instances of misprediction.

3.11 [20/15/15/15] <3.4> Figure 3.15 on page 206 and Figure 3.18 on page 208 show that the prediction accuracy of a local 2-bit predictor improves very slowly with increasing branch-prediction buffer size, once size exceeds some amount. Because prediction buffers must be of finite size, two or more branches may be mapped to the same buffer entry. While it is possible for sharing to improve branch prediction by accidentally allowing sharing of information between related branches, typically the sharing results in destructive interference. For the following, assume that prediction accuracy is always worse when branches share a predictor.

a. [20] <3.4> For a branch-prediction buffer implementing a given type of predictor, what characteristic of any program guarantees that prediction accuracy as a function of increasing buffer size must eventually become constant (i.e., independent of buffer size)? *Hint:* examine the contents of and compare the prediction accuracy of branch-prediction buffers of different sizes on a simple code fragment such as the following:

```
Loop:   DSUBI R1,R1,#1
        BNEZ  R1,Loop
        LD    R10,0(R3)
```

b. [15] <3.4> Figure 3.15 shows that, to within the precision of the measurements, the SPEC89 benchmarks nasa7, tomcatv, and gcc were the only programs to have less branch misprediction when buffer size increased from 4096 entries to infinite. Based on the answer to part (a), what quantitative measure can you infer about the machine instruction count of the executable codes for these four benchmarks?

c. [15] <3.4> Can you infer anything similar to the result in part (b) about the instruction counts of the other seven benchmarks?

d. [15] <3.4> How might an optimizing compiler improve prediction accuracy for the other seven benchmarks in part (c) and when would this be and not be possible?

3.12 [30] <3.4> Implement a simulator to evaluate the performance of a branch-prediction buffer that does not store branches that are predicted as untaken. Consider the following prediction schemes: a 1-bit predictor storing only predicted-taken branches, a 2-bit predictor storing all the branches, a scheme with a target buffer that stores only predicted-taken branches, and a 2-bit prediction buffer. Explore different sizes for the buffers, keeping the total number of bits (assuming 32-bit addresses) the same for all schemes. Determine what the branch penalties are, using Figure 3.21 as a guideline. How do the different schemes compare both in prediction accuracy and in branch cost?

3.13 [30] <3.4> Implement a simulator to evaluate various branch-prediction schemes. You can use the instruction portion of a set of cache traces to simulate the branch-prediction buffer. Pick a set of table sizes (e.g., 1K bits, 2K bits, 8K bits, and 16K bits). Determine the performance of both (0,2) and (2,2) predictors for the various table sizes. Also compare the performance of the degenerate predictor that uses no branch address information for these table sizes. Determine how large the table must be for the degenerate predictor to perform as well as a (0,2) predictor with 256 entries.

✪ 3.14 [15] <3.5> Suppose we have a deeply pipelined processor, for which we implement a branch-target buffer for the conditional branches only. Assume that the misprediction penalty is always 4 cycles and the buffer miss penalty is always 3 cycles. Assume 90% hit rate and 90% accuracy, and 15% branch frequency. How much faster is the processor with the branch-target buffer versus a processor that has a fixed 2-cycle branch penalty? Assume a base CPI without branch stalls of 1.

3.15 [10/15] <3.5> Consider a branch-target buffer that has penalties of 0, 2, and 2 clock cycles for correct conditional branch prediction, incorrect prediction, and a buffer miss, respectively. Consider a branch-target buffer design that distinguishes conditional and unconditional branches, storing the target address for a conditional branch and the target instruction for an unconditional branch.

a. [10] <3.5> What is the penalty in clock cycles when an unconditional branch is found in the buffer?

b. [15] <3.5> Determine the improvement from branch folding for unconditional branches. Assume a 90% hit rate, an unconditional branch frequency of 5%, and a 2-cycle penalty for a buffer miss. How much improvement is gained by this enhancement? How high must the hit rate be for this enhancement to provide a performance gain?

3.16 [10/20/20/20/20/15] <3.1, 3.6> This exercise explores variations on the theme of the example on page 221. Each part of this exercise lists changes to make to the set of assumptions given in the example. For parts (b) through (d) do the follow-

ing: (i) produce a new version of the table in Figure 3.25 covering enough iterations to reach a steady-state condition, (ii) compute a value for the sustained instruction completion rate, and (iii) provide any other information requested in that part.

Make the following two assumptions also:

- There is only one memory access port.
- If there is contention for a resource, then the earliest instruction in program order is given access to that resource.

a. [10] <3.1, 3.6> What structural hazards occur in the example?

b. [20] <3.6> Assume that there are two integer functional units. Describe any structural hazards and compare them to the case for the original example.

c. [20] <3.6> Assume that three instructions may issue simultaneously (but the BNE still issues separately). Describe any structural hazards.

d. [20] <3.6> Assume branches are speculated and issue with another instruction. Assume that branch prediction is perfect. Count and describe the structural hazards.

e. [20] <3.6> Assume branches are speculated and issue with another instruction. Assume that branch prediction is perfect. Assume that there are two integer functional units.

f. [15] <3.6> Discuss the relative magnitude of performance benefit derived from adding an integer functional unit, increasing issue width, and supporting speculation. Comment on the potential for synergy between various enhancements.

✪ 3.17 [10/20/15] <3.6> To keep the issue step of the statically scheduled superscalar MIPS processor quite simple, an issue limit of one integer and one floating-point instruction per clock was imposed. Let's remove this restriction and see how issue step workload grows with increasing multiple-issue capability. Assume a five-stage superscalar pipeline (IF, ID, EX, MEM, WB) with no issue restrictions and no structural hazards. Also, regardless of instruction, each stage always takes just 1 clock cycle to complete its task, and an instruction may have up to two operands and one result.

a. [10] <3.6> The ID stage must check for what type(s) of data dependences?

b. [20] <3.6> For a two-issue design with 32 integer registers and 32 floating-point registers, how many bits must be brought to comparators in the ID stage and how many comparisons must be performed during each clock cycle to check just for data hazards? How many if the issue width is doubled?

c. [15] <3.6> Let the issue limit be n instructions, and assume the total number of registers is unbounded. How many comparisons, as a function of n, must be performed to check just for data hazards?

3.18 [25] <3.7> Consider the execution of the following loop, which searches an array, on a single-issue processor, first with dynamic scheduling and then with speculation:

```
Loop:   LD      R2,0(R1)    ;R2=array element
        DADDI   R2,R2,#1    ;increment R2
        SD      R2,0(R1)    ;store result
        DADDI   R1,R1,#4    ;decrement pointer
        BNEZ    R2,LOOP     ;branch if last element!=0
```

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table as in Figure 3.27 for the first three iterations of this loop for both machines. Assume that one instruction can commit per clock.

3.19 [15] <3.7> Use of a speculative technique may decrease performance. This is certainly true in particular when a speculative guess is wrong; performance at that point in the program is less than it would have been without speculation. However, it can also be true in general, that is, for a substantial workload such as an entire program. For a speculative technique to improve performance in general, what mathematical condition must be true? *Hint:* Construct a model in terms of speculation costs, benefits, and frequencies.

✪ 3.20 [15] <1.6, 3.7> When an instruction is correctly speculated, what is the effect on the three factors comprising the CPU time formula (from Chapter 1): dynamic instruction count, average clocks per instruction, and clock cycle time? When speculation is incorrect, it is possible for CPU time to increase. Which factor(s) of the CPU time formula best model this increase and why?

✪ 3.21 [15] <3.7> Consider the speculative Tomasulo processor shown in Figure 3.28 on page 225. Assume that the ROB has three buffer entries, named 0, 1, and 2. For the following code fragment, assume that `ADD.D`, `SUB.D`, and `ADDI` instructions execute for 1 cycle, and `MUL.D` executes for 10 cycles. Assume that the processor has sufficient function units to avoid stalling instruction issue.

```
ADD.D    F0, F8, F8
MUL.D    F2, F8, F8
SUB.D    F4, F0, F2
DADDI    R10,R12,R12
```

Fill in the table below to show ROB contents and history as it would exist on the cycle that the ADDI instruction writes its result. Assume that F8 and R12 are initialized and that the ROB is initially empty. (One table entry is already filled in to provide a fixed starting point for your answer.) Because a ROB is implemented as a circular queue, the entry number labels repeat modulo 3 reading down the table. If a ROB entry would be reallocated during the simulated execution time, write the details of the new allocation in the next available correspondingly numbered table row. Use the rightmost column to indicate if the instruction has been committed.

	ROB fields			Committed?
Entry	Instruction	Destination	Value	Yes/no
0	ADD.D			
1				
2				
0				
1				
2				

3.22 [20/15] <3.5, 3.7> Consider our speculative processor from Section 3.7. Since the reorder buffer contains a value field, you might think that the value field of the reservation stations could be eliminated.

a. [20] <3.5, 3.7> Show an example where this is the case and an example where the value field of the reservation stations is still needed. Use the speculative machine shown in Figure 3.29. Show MIPS code for both examples. How many value fields are needed in each reservation station?

b. [15] <3.5, 3.7> Find a modification to the rules for instruction commit that allows elimination of the value fields in the reservation station. What are the negative side effects of such a change?

3.23 [25] <3.7> Our implementation of speculation uses a reorder buffer and introduces the concept of instruction commit, delaying commit and the irrevocable updating of the registers until we know an instruction will complete. There are two other possible implementation techniques, both originally developed as a method for preserving precise interrupts when issuing out of order. One idea introduces a future file that keeps future values of a register; this idea is similar to the reorder buffer. An alternative is to keep a history buffer that records values of registers that have been speculatively overwritten.

Design a speculative processor like the one in Section 3.7 but using a history buffer. Show the state of the processor, including the contents of the history buffer, for the example in Figure 3.31. Show the changes needed to Figure 3.32 for a history buffer implementation. Describe exactly how and when entries in the history buffer are read and written, including what happens on an incorrect speculation.

3.24 [15/25] <3.8, 3.9> Aggressive hardware support for ILP is detailed at the beginning of Section 3.9. Keeping such a processor from stalling due to lack of work requires an average instruction fetch rate, f, that equals the average instruction completion rate, c. Achieving a high fetch rate is challenging in the presence of cache misses. Branches add to the difficulty and are ignored in this exercise. To explore just how challenging, model the average instruction memory access time as $h + mp$, where h is the time in clock cycles for a successful cache access, m is the rate of unsuccessful cache access, and p is the extra time, or penalty, in clock cycles to fetch from main memory instead of the cache.

a. [15] <3.8, 3.9> Write an equation for the number of instructions that the processor must attempt to fetch each clock cycle to achieve on average fetch rate $f = c$.

b. [25] <3.8, 3.9> Using a program with suitable graphing capability, such as a spreadsheet, plot the equation from part (a) for $0.01 \le m \le 0.1$, $10 \le p \le 100$, $1 \le h \le 2$ and a completion rate of 4 instructions per clock cycle. Comment on the importance of low average memory access time to the feasibility of achieving even modest average fetch rates.

3.25 [45] <3.2, 3.3> One benefit of a dynamically scheduled processor is its ability to tolerate changes in latency or issue capability without requiring recompilation. This capability was a primary motivation behind the 360/91 implementation of Tomasulo's algorithm. The purpose of this programming assignment is to evaluate this effect. Implement a version of Tomasulo's algorithm for MIPS to issue one instruction per clock; your implementation should also be capable of in-order issue. Assume fully pipelined functional units and the execution times in Figure 3.64.

Choose 5–10 small FP benchmarks (with loops) to run; compare the performance with and without dynamic scheduling. Try scheduling the loops by hand and see how close you can get with the statically scheduled processor to the dynamically scheduled results.

Change the processor to the configuration shown in Figure 3.65. Rerun the loops and compare the performance of the dynamically scheduled processor and the statically scheduled processor.

3.26 [45] <3.6> Perform the investigation of Exercise 3.25 but for a multiple-issue version of Tomasulo's algorithm. Use and/or adapt appropriate simulation tools.

3.27 [30/30] <3.8, 3.9> This exercise involves a programming assignment to evaluate what types of parallelism might be expected in more modest, and more realistic, processors than those studied in Section 3.8. These studies will require execution traces. You may be able to find traces or obtain them from a tracing program. For simplicity, assume perfect caches. For a more ambitious project, assume a real cache. To simplify the task, make the following assumptions:

- Assume perfect branch and jump prediction: hence you can use the trace as the input to the window, without having to consider branch effects—the trace is perfect.
- Assume there are 64 spare integer and 64 spare floating-point registers; this is easily implemented by stalling the issue of the processor whenever there are more live registers required.
- Assume a window size of 64 instructions (the same for alias detection). Use greedy scheduling of instructions in the window. That is, at any clock cycle, pick for execution the first n instructions in the window that meet the issue constraints.

Unit	Execution time (clocks)
Integer	7
Branch	9
Load-store	11
FP add	13
FP mul	15
FP divide	17

Figure 3.64 Execution times for functional unit pipelines.

Unit	Execution time (clocks)
Integer	19
Branch	21
Load-store	23
FP add	25
FP mul	27
FP divide	29

Figure 3.65 Execution times for functional units, configuration 2.

a. [30] <3.8, 3.9> Determine the effect of limited instruction issue by performing the following experiments:
   - Vary the issue count from 4 to16 instructions per clock.
   - Assuming eight issues per clock, determine the effect of restricting the processor to two memory references per clock.

b. [30] <3.8, 3.9> Determine the impact of latency in instructions. Remember that with limited issue and a greedy scheduler, the impact of latency effects will be greater. Assume the following latency models for a processor that issues up to 16 instructions per clock:
   - Model 1: All latencies are 1 clock.
   - Model 2: Load latency and branch latency are 1 clock; all FP latencies are 2 clocks.
   - Model 3: Load and branch latency is 2 clocks; all FP latencies are 5 clocks.

3.28 [Discussion] <3.2, 3.3> There is a subtle problem that must be considered when implementing Tomasulo's algorithm. It might be called the "two ships passing in the night problem." What happens if an instruction is being passed to a reservation station during the same clock period as one of its operands is going onto the common data bus? Before an instruction is in a reservation station, the operands are fetched from the register file; but once it is in the station, the operands are

always obtained from the CDB. Since the instruction and its operand tag are in transit to the reservation station, the tag cannot be matched against the tag on the CDB. So there is a possibility that the instruction will then sit in the reservation station forever waiting for its operand, which it just missed. How might this problem be solved? You might consider subdividing one of the steps in the algorithm into multiple parts. (This intriguing problem is courtesy of J. E. Smith.)

3.29 [Discussion] <3.3, 3.7> Dynamic instruction scheduling requires a considerable investment in hardware. In return, this capability allows the hardware to run programs that could not be run at full speed with only compile time, static scheduling. What trade-offs should be taken into account in trying to decide between a dynamically and a statically scheduled implementation? What situations in either hardware technology or program characteristics are likely to favor one approach or the other? Most speculative schemes rely on dynamic scheduling; how does speculation affect the arguments in favor of dynamic scheduling?

3.30 [Discussion] <3.5> A branch-target buffer offers potential performance gains, but at a cost. Power consumption and clock cycle time may be critical design issues for a processor. Discuss the effect that a branch-target buffer has on these parameters as a function of buffer size. How is this BTB effect on power and clock cycle time similar to or different from that of other pipeline structures, such as the register file or ALU? What ways can you think of to improve the benefits of a BTB while at the same time meeting or exceeding power consumption and clock cycle time goals?

3.31 [Discussion] <3.6, A> Discuss the advantages and disadvantages of a superscalar implementation and a superpipelined implementation in the context of MIPS. What types of ILP favor each approach? What other concerns would you consider in choosing which type of processor to build? How does speculation affect the results?