

Suppose that the number of cycles per instruction would be 1.5 with perfect branch prediction. What is the percentage slowdown caused by mispredicted branches? Now suppose that we have a superscalar processor on which the number of cycles per instruction would be 0.6 with perfect branch prediction. Now what is the percentage slowdown caused by mispredicted branches? What do your answers tell you about the importance of branch prediction on superscalar machines?

- 5.23 Consider the code in Figure ©5.9. In an attempt to eliminate the remaining delay, and reduce the overhead of the bookkeeping (loop control) instructions, one might consider *unrolling* the loop: creating a new loop in which each iteration performs the work of  $k$  iterations of the original loop. Show the code for  $k = 2$ . You may assume that  $n$  is even, and that your target machine supports displacement addressing. Schedule instructions as tightly as you can. How many cycles does your loop consume per vector element?

## 5.8 Explorations

- 5.24 Skip ahead to the sidebar on decimal types on page 296 of the main text. Write algorithms to convert BCD numbers to binary, and vice versa. Try writing the routines in assembly language for your favorite machine (if your machine has special instructions for this purpose, pretend you're not allowed to use them). How many cycles are required for the conversion?
- 5.25 Is microprogramming an idea that has outlived its usefulness, or are there application domains for which it still makes sense to build a microprogrammed machine? Defend your answer.
- 5.26 If you have access to both CISC and RISC machines, compile a few programs for both machines and compare the size of the target code. Can you generalize about the "space penalty" of RISC code?
- 5.27 The Intel IA-64 (Itanium) architecture is neither CISC nor RISC. It belongs to an architectural family known as *long instruction word* (LIW) machines (Intel calls it *explicitly parallel instruction set computing* [EPIC]). Find an Itanium manual or tutorial and learn about the instruction set. Compare and contrast it with the x86 and MIPS instruction sets. Discuss, from a compiler writer's point of view, the challenges and opportunities presented by the IA-64.
- 5.28 Research the history of the x86. Learn how it has been extended over the years. Write a brief paper describing the extensions. Identify the portions of the instruction set that are still useful today (i.e., are targeted by modern compilers), and the portions that are maintained solely for the sake of backward compatibility.
- 5.29 The x86-64 architecture is a backward-compatible 64-bit extension of the x86. Find a manual or tutorial and learn about the instruction set. Describe

the extensions it provides. Explain how it can execute 32-bit x86 instructions without an explicit “compatibility mode.”

- 5.30 Several computers have provided more general versions of the *conditional move* instructions described in Exercise ©5.16. Examples include the c. 1965 IBM ACS, the Cray 1, the HP PA-RISC, the ARM, and the Intel IA-64 (Itanium). General-purpose conditional execution is sometimes known as *predication*.

Learn how predication works in ARM or IA-64. Explain how it can sometimes improve performance even when it causes the processor to execute *more* instructions.

- 5.31 If you have access to computers of more than one type, compile a few programs on each machine and time their execution. (If possible, use the same compiler [e.g., gcc] and options on each machine.) Discuss the factors that may contribute to different run times. How closely do the ratios of run times mirror the ratios of clock rates? Why don't they mirror them exactly?
- 5.32 Branch prediction can be characterized as *control speculation*: it makes a guess about the future control flow of the program that saves enough time when it's right to outweigh the cost of cleanup when it's wrong. Some researchers have proposed the complementary notion of *value speculation*, in which the processor would predict the value to be returned by a cache miss, and proceed on the basis of that guess. What do you think of this idea? How might you evaluate its potential?
- 5.33 Can speculation be useful in software? How might you (or a compiler or other tool) be able to improve performance by making guesses that are subject to future verification, with (software) rollback when wrong? (Hint: Think about operations that require communication over slow Internet links.)
- 5.34 Translate the high-level pseudocode for vector variance (Example ©5.18) into your favorite programming language, and run it through your favorite compiler. Examine the resulting assembly language. Experiment with different levels of optimization (code improvement). Discuss the quality of the code produced.
- 5.35 Try to write a code fragment in your favorite programming language that requires so many registers that your favorite compiler is forced to spill some registers to memory (compile with a high level of optimization). How complex does your code have to be?
- 5.36 If you have access to a compiler that generates code for a machine with architecturally visible load delays, run some programs through it and evaluate the degree of success it has in filling delay slots (an unfilled slot will contain a nop instruction). What percentage of slots is filled? Suppose the machine had interlocked loads. How much space could be saved in typical executable programs if the nops were eliminated?
- 5.37 Experiment with small subroutines in C++ to see how much time can be saved by expanding them inline.