

enhancements were expected to migrate to the new instruction set. AMD took a more conservative approach, at least from a marketing perspective, and developed a backward-compatible 64-bit extension to the x86 instruction set; its AMD64 processors provided a much smoother upward migration path. In response to market demand, Intel subsequently licensed the AMD64 architecture (which it now calls Intel 64) for use in its 64-bit Pentium processors.

As processor and compiler technology continue to evolve, it is likely that processor implementations will continue to become more complex, and that compilers will take on additional tasks in order to harness that complexity. Traditional CISC machines remain popular almost entirely due to the need for backward compatibility, but both the CISC and RISC “design philosophies” are still very much alive [SW94]. The “CISC-ish” philosophy says that newly available resources (e.g., increases in chip area) should be used to implement functions that must currently be performed in software, such as vector or graphics operations, decimal arithmetic, new addressing modes, or perhaps transactional memory (to be described in Section 12.4.4). The “RISC-ish” philosophy says that resources should be used to improve the speed of existing functions, for example by increasing cache size, employing faster but larger functional units, increasing the number of cores, or deepening the pipeline and decreasing the cycle time.

Where the first-generation RISC machines from different vendors differed from one another only in minor details, later generations diverged, with the ARM and MIPS taking the more RISC-ish approach, the Power/PowerPC family taking the more CISC-ish approach, and the SPARC somewhere in the middle. It is not yet clear which approach will ultimately prove most effective, nor is it even clear that this is the interesting question anymore. Heat dissipation and limited ILP are increasingly the main constraints on uniprocessor performance. In response to these constraints, most vendors are now pursuing multicore versions of their respective architectures. It is entirely possible that future processors will be highly heterogeneous, with multiple implementation strategies—or even multiple instruction set architectures—deployed in different cores, each optimized for a different sort of program. Such processors will certainly require new compiler techniques. At perhaps no time in the past 25 years has the future of microarchitecture been in so much flux. However it all turns out, it is clear that processor and compiler technology will continue to evolve together.

5.7 Exercises

- 5.1 Consider sending a message containing a string of integers over the Internet. What problems may occur if the sending and receiving machines have different “endian-ness”? How might you solve these problems?
- 5.2 What is the largest positive number in 32-bit two’s complement arithmetic? What is the smallest (largest magnitude) negative number? Why are these numbers not the additive inverse of each other?

- 5.3 (a) Express the decimal number 1234 in hexadecimal.
 (b) Express the unsigned hexadecimal number 0x2ae in decimal.
 (c) Interpret the hexadecimal bit pattern 0xffd9 as a 16-bit 2's complement number. What is its decimal value?
 (d) Suppose that n is a negative integer represented as a k -bit 2's complement bit pattern. If we reinterpret this bit pattern as an unsigned number, what is its numeric value as a function of n and k ?
- 5.4 What will the following C code print on a little-endian machine such as a Pentium? What will it print on a big-endian machine such as a Sun?

```
unsigned short n = 0x1234;
unsigned char *p = (unsigned char *) &n;
printf ("%d\n", *p);
```

- 5.5 (a) Suppose we have a machine with hardware support for 8-bit integers. What is the decimal value of 11011001_2 , interpreted as an unsigned quantity? As a signed, two's complement quantify? What is its two's complement additive inverse?
 (b) What is the 8-bit binary sum of 11011001_2 and 10010001_2 ? Does this sum result in overflow if we interpret the addends as unsigned numbers? As signed two's complement numbers?
- 5.6 In Section 5.2.1 we observed that overflow occurs in two's complement addition when we add two nonnegative numbers and obtain an apparently negative result, or add two negative numbers and obtain an apparently non-negative result. Prove that it is equivalent to say that a two's complement addition operation overflows if and only if the carry into most significant place differs from the carry out of most significant place. (This trivial check is the one typically performed in hardware.)
- 5.7 In Section 5.2.1 we claimed that a two's complement integer could be correctly negated by flipping the bits, adding 1, and discarding any carry out of the left-most place. Prove that this claim is correct.
- 5.8 What is the single-precision IEEE floating-point number whose value is closest to 6.022×10^{23} ?
- 5.9 Occasionally one sees a C program in which a double-precision floating-point number is used as an integer counter. Why might a programmer choose to do this?
- 5.10 Modern compilers often find they don't have enough registers to hold all the things they'd like to hold. At the same time, VLSI technology has reached the point at which there is room on a chip to hold many more registers than are found in the typical ISA. Why are we still using instruction sets with only 32 integer registers? Why don't we make, say, 64 or 128 of them visible to the programmer?

- 5.11 Some early RISC machines (e.g., the SPARC) provided a “multiply step” instruction that performed one iteration of the standard shift-and-add algorithm for binary integer multiplication. Speculate as to the rationale for this instruction.
- 5.12 Why do you think RISC machines standardized on 32-bit instructions? Why not some smaller or larger length? Why not variable lengths?
- 5.13 Consider a machine with three condition codes, N, Z, and O. N indicates whether the most recent arithmetic operation produced a negative result. Z indicates whether it produced a zero result. O indicates whether it produced a result that cannot be represented in the available precision for the numbers being manipulated (i.e., outside the range $0 \dots 2^n$ for unsigned arithmetic, $-2^{n-1} \dots 2^{n-1}-1$ for signed arithmetic). Suppose we wish to branch on condition $A \text{ op } B$, where A and B are unsigned binary numbers, for $\text{op} \in \{<, \leq, =, \neq, >, \geq\}$. Suppose we subtract B from A, using two’s complement arithmetic. For each of the six conditions, indicate the logical combination of condition-code bits that should be used to trigger the branch. Repeat the exercise on the assumption that A and B are signed, two’s complement numbers.
- 5.14 We implied in Section ©5.4.1 that if one adds a new instruction to a non-pipelined, microcoded machine, the time required to execute that instruction is (to first approximation) independent of the time required to execute all other instructions. Why is it not strictly independent? What factors could cause overall execution to become slower when a new instruction is introduced?
- 5.15 Suppose that loads constitute 25% of the typical instruction mix on a certain machine. Suppose further that 15% of these loads miss in the on-chip (primary) cache, with a penalty of 40 cycles to reach main memory. What is the contribution of cache misses to the average number of cycles per instruction? You may assume that instruction fetches always hit in the cache. Now suppose that we add an off-chip (secondary) cache that can satisfy 90% of the misses from the primary cache, at a penalty of only 10 cycles. What is the effect on cycles per instruction?
- 5.16 Many recent processors provide a *conditional move* instruction that copies one register into another if and only if the value in a third register is (or is not) equal to zero. Give an example in which the use of conditional moves leads to a shorter program.
- 5.17 The x86-64 architecture is backward compatible with the x86 instruction set, just as the x86 is backward compatible with the 16-bit 8086 instruction set. Less transparently, the IA-64 Itanium is capable of running legacy x86 applications in “compatibility mode.” But recent members of the ARM and MIPS processor families support *new* 16-bit instructions as an *extension* to the architecture. Why might designers have chosen to introduce these new, less powerful modes of execution?

5.18 Consider the following code fragment in pseudo-assembler notation:

```

1.    r1 := K
2.    r4 := &A
3.    r6 := &B
4.    r2 := r1 × 4
5.    r3 := r4 + r2
6.    r3 := *r3          -- load (register indirect)
7.    r5 := *(r3 + 12)   -- load (displacement)
8.    r3 := r6 + r2
9.    r3 := *r3          -- load (register indirect)
10.   r7 := *(r3 + 12)   -- load (displacement)
11.   r3 := r5 + r7
12.   S := r3           -- store

```

- (a) Give a plausible explanation for this code (what might the corresponding source code be doing?).
 - (b) Identify all flow, anti-, and output dependences.
 - (c) Schedule the code to minimize load delays on a single-pipeline, in-order processor.
 - (d) Can you do better if you rename registers?
- 5.19** With the development of deeper, more complex pipelines, delayed loads and branches have become significantly less appealing as features of a RISC instruction set. Why is it that designers have been able to eliminate delayed loads in more recent machines, but have had to retain delayed branches?
- 5.20** Some processors, including the PowerPC and recent members of the x86 family, require one or more cycles to elapse between a condition-determining instruction and a branch instruction that uses that condition. What options does a scheduler have for filling such delays?
- 5.21** Branch prediction can be performed statically (in the compiler) or dynamically (in hardware). In the static approach, the compiler guesses which way the branch will usually go, encodes this guess in the instruction, and schedules instructions for the expected path. In the dynamic approach, the hardware keeps track of the outcome of recent branches, notices branches or patterns of branches that recur, and predicts that the patterns will continue in the future. Discuss the tradeoffs between these two approaches. What are their comparative advantages and disadvantages?
- 5.22** Consider a machine with a three-cycle penalty for incorrectly predicted branches and a zero-cycle penalty for correctly predicted branches. Suppose that in a typical program 20% of the instructions are conditional branches, which the compiler or hardware manages to predict correctly 75% of the time. What is the impact of incorrect predictions on the average number of cycles per instruction? Suppose the accuracy of branch prediction can be increased to 90%. What is the impact on cycles per instruction?

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