5 Target Machine Architecture

Processor implementations change over time, as people invent better ways of doing things, and as technological advances (e.g., increases in the number of transistors that will fit on one chip) make things feasible that were not feasible before. Processor architectures also change, for at least two reasons. Some technological advances can be exploited only by changing the hardware/software interface—for example by increasing the number of bits that can be added or multiplied in a single instruction. In addition, experience with compilers and applications sometimes suggests that certain new instructions would make programs simpler or faster.

Occasionally, technological and intellectual trends converge to produce a revolutionary change in both architecture and implementation. We will discuss four such changes in Section C-5.4: the development of microprogramming in the early 1960s, the development of the microprocessor in the early to mid-1970s, the development of reduced instruction set computing (RISC) in the early 1980s, and the move to multithreaded and multicore processors in the first decade of the 21st century.

This chapter provides a quick overview of those aspects of computer architecture most essential to the task of compiler construction. In Sections C-5.1–C-5.3 we consider the hierarchical organization of memory, the types (formats) of data found in memory, and the instructions used to manipulate those data. The coverage is necessarily somewhat cursory and high-level; much more detail can be found in books on computer architecture or organization (e.g., Chapters 2–5 of Patterson and Hennessy's outstanding text [PH12]).

We consider the interplay between architecture and implementation in Section C-5.4. As illustrative examples, we consider the widely used x86 and ARM instruction sets. Finally, in Section C-5.5, we consider some of the issues that make compiling for modern processors a challenging task.

	Typical access time	Typical capacity
Registers	0.2–0.5 ns	256–1024 bytes
Primary (L1) cache	0.4–1 ns	32 K–256 K bytes
L2 or L3 (on-chip) cache	4–30 ns	1–32 M bytes
off-chip cache	10–50 ns	up to 128 M bytes
Main memory	50–200 ns	256 M–16 G bytes
Flash	40–400 μs	4 G bytes to 1 T bytes
Disk	5–15 ms	500 G bytes and up
Tape	1–50 s	effectively unlimited

Figure 5.1 **The memory hierarchy of a workstation-class computer.** Access times and capacities are approximate, based on 2015 technology. Registers are accessed within a single clock cycle. Primary cache typically responds in 1 to 2 cycles; off-chip cache in more like 20 cycles. Main memory on a supercomputer can be as fast as off-chip cache; on a workstation it is typically much slower. Flash times vary with manufacturing technology, and are longer for writes than reads. Disk and tape times are constrained by the movement of physical parts.

5.1 The Memory Hierarchy

Memory on most machines consists of a numbered sequence of 8-bit bytes. The size of the sequence—the number of distinct locations—is limited by the number of bits used to represent an address. This is a sufficiently important number that it is often used to categorize machines. Programs on a "32-bit machine" can address no more than 2^{32} bytes (4 GB) of memory. Programs on a "64-bit machine" can (at least in principle) address 4 billion times as much.

It is not uncommon for modern workstations to contain multiple gigabytes of memory—much too much to fit on the same chip as the processor. The time it takes to reach memory depends on its distance from the processor. Off-chip memory—particularly when located on the other side of an interconnection network shared by other processors and devices—is particularly slow. Most computers therefore employ a *memory hierarchy*, in which things that are used more often are kept close at hand. A typical memory hierarchy, with access times and capacities, is shown in Figure C-5.1. ■

EXAMPLE 5.1
Memory hierarchy stats

Only three of the levels of the memory hierarchy—registers, memory, and devices—are a visible part of the hardware/software interface. Compilers manage registers explicitly, loading them from memory when needed and storing them back to memory when done, or when the registers are needed for something else. Caches are managed by the hardware. Devices are generally accessed only by the operating system.

Registers hold small amounts of data that can be accessed very quickly. A typical modern machine has two sets of registers—one to hold integer operands, the other floating-point. Additional sets may be used for special purposes—for example, vector instructions, which operate, in parallel, on a sequence of shorter

values packed into a longer register. There are usually several special-purpose registers as well, including the *program counter* (PC) and the *processor status register*. The program counter holds the address of the next instruction to be executed. It is incremented automatically when fetching most instructions; branches work by changing it explicitly. The processor status register contains a variety of bits of importance to the operating system (privilege level, interrupt priority level, trap enable bits) and, on some machines, a few bits of importance to the compiler writer. Principal among these are *condition codes*, which indicate whether the most recent arithmetic or logical operation resulted in a zero, a negative value, and/or arithmetic overflow. (We will consider condition codes in more detail in Section C-5.3.2.)

Because registers can be accessed every cycle, while memory, generally, cannot, good compilers expend a great deal of effort trying to make sure that the data they need most often are in registers, and trying to minimize the amount of time spent moving data back and forth between registers and memory. We will consider algorithms for register management in Section C-5.5.2.

Caches are generally smaller but faster than main memory. They are designed to exploit *locality*: the tendency of most computer programs to access the same or nearby locations in memory repeatedly. By automatically moving the contents of these locations into cache, a hierarchical memory system can dramatically improve performance. The idea makes intuitive sense: loops tend to access the same local variables in every iteration, and to walk sequentially through arrays. Instructions, likewise, tend to be loaded from consecutive locations, and code that accesses one element of a structure (or member of a class) is likely to access another.

Cache architecture varies quite a bit across machines. Primary caches, also known as *level-1 (L1) caches*, are invariably located on the same chip as the processor, and usually come in pairs: one for instructions (the L1 I-cache) and another for data (the L1 D-cache), both of which can be accessed every cycle. Secondary caches are larger and slower, but still faster than main memory. In a modern desktop or laptop system they are typically also on the same chip as the processor. High-end desktop or server-class machines may have an on-chip tertiary (L3)

DESIGN & IMPLEMENTATION

5.2 The processor/memory gap

For roughly 50 years, from the 1950s until about 2004, processor speed increased much faster than memory speed. As a result, the number of processor cycles required to access memory grew dramatically, and caches became increasingly critical to performance. To improve the effectiveness of caching, programmers need to choose algorithms whose data access patterns have a high degree of locality. High-quality compilers, likewise, need to consider locality of access when choosing among the many possible translations of a given program.

and/or an off-chip cache as well. Multicore processors, which have more than one processing core on a single chip, may share the L2 or (on-chip) L3 among cores. Small embedded processors may have only a single level of on-chip cache. Caches are managed entirely in hardware on most machines, but compilers can increase their effectiveness by generating code with a high degree of locality.

A memory access that finds its data in the cache is said to be a *cache hit*. An access that does not find its data in the cache is said to be a *cache miss*. On a miss, the hardware automatically loads a *line* of the cache with a contiguous block of data containing the requested location, obtained from the next lower level of cache or main memory. Cache lines vary from as few as 16 to as many as 512 bytes in length. Assuming that the cache was already full, the load will displace some other line, which is written back to memory if it has been modified.

A final characteristic of memory that is important to the compiler is known as data *alignment*. Most machines are able to manipulate operands of several sizes, typically one, two, four, and eight bytes. Most modern instruction sets refer to these as byte, half-word, word, and double-word operands, respectively; on the x86 they are byte, word, double-word, and quad-word operands. Many recent architectures require n-byte operands to appear in memory at addresses that are evenly divisible by n (at least for $n \leq 4$). A 4-byte integer, for example, must typically appear at a location whose address is evenly divisible by four. This restriction occurs for two reasons. First, buses are designed in such a way that data are delivered to the processor over bit-parallel, aligned communication paths. Loading an integer from an odd address would require that the bits be shifted, adding logic (and time) to the load path. The x86 and ARM, which allow most operands to appear at arbitrary addresses, run faster if those operands are properly aligned. Second, on machines with fixed-size instructions, there are generally not enough bits to specify both an operation (e.g., `load`) and a full address. As we shall see in Section C-5.3.1, it is typical to specify an address in terms of an *offset* from some *base location* specified by a register. Requiring that integers be word-aligned allows the offset to be specified in words, rather than in bytes, quadrupling the amount of memory that can be accessed using offsets from a given base register.

5.2 Data Representation

Data in the memory of most computers are untyped: bits are simply bits. *Operations* are typed, in the sense that different operations *interpret* the bits in memory in different ways. Typical *data formats* include instructions, addresses, binary integers of various lengths, floating-point (real) numbers of various lengths, and characters.

Integers typically come in half-word, word, and double-word lengths. Floating-point numbers typically come in word and double-word lengths, commonly referred to as *single-* and *double-precision*. Some machines store the least-significant byte of a multiword datum at the address of the datum itself, with

EXAMPLE 5.2
Big- and little-endian

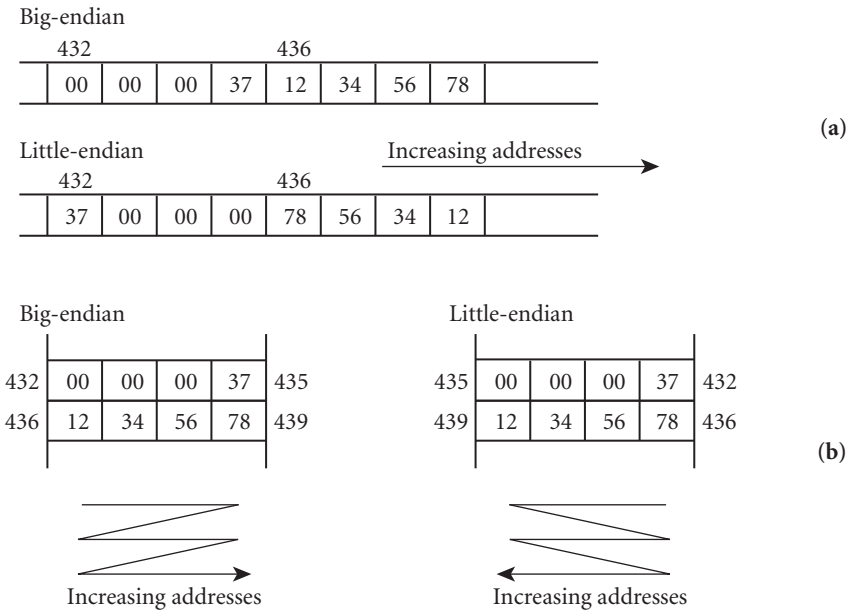


Figure 5.2 **Big-endian and little-endian byte orderings.** (a) Two 4-byte quantities, the numbers 37_{16} and $12\,34\,56\,78_{16}$, stored at addresses 432 and 436, respectively. (b) The same situation, with memory visualized as a byte-addressable array of words.

bytes of increasing numeric significance at higher-numbered addresses. Other machines store the bytes in the opposite order. The first option is called *little-endian*; the second is called *big-endian*. In either case, an n-byte datum stored at address t occupies bytes t through $t + n - 1$. The advantage of a little-endian organization is that it is tolerant of variations in operand size. If the value 37 is stored as a word and then a byte is read from the same location, the value 37 will be returned. On a big-endian machine, the value 0 will be returned (the upper eight bits of the number 37, when stored in 32 bits). The problem with the little-endian approach is that it seems to scramble the bytes of integers, when read from left to right (see Figure C-5.2a). Little-endian-ness makes a bit more sense if one thinks of memory as a (byte-addressable) array of words (Figure C-5.2b). The x86 is little-endian. IBM's z Series (mainframe) machines are big-endian. Most other common processors, including the ARM, SPARC, MIPS, and Power families, can run in either mode, at the choice of the operating system. ■

Support for characters varies widely. A few machines can perform arbitrary arithmetic and logical operations on 1-byte quantities. Most can load and store bytes from or to memory, but operate only on longer quantities in registers. Some legacy machines, including the x86, provide instructions that perform operations on strings of characters, such as copying, comparing, or searching. On more modern machines (again including the x86), vector instructions can also be used to operate on strings.

0 0 0 0	0	1 0 0 0	8
0 0 0 1	1	1 0 0 1	9
0 0 1 0	2	1 0 1 0	a
0 0 1 1	3	1 0 1 1	b
0 1 0 0	4	1 1 0 0	c
0 1 0 1	5	1 1 0 1	d
0 1 1 0	6	1 1 1 0	e
0 1 1 1	7	1 1 1 1	f

Figure 5.3 The hexadecimal digits.

5.2.1 Integer Arithmetic

Binary integers are almost universally represented in two related formats: straightforward binary place-value for unsigned numbers, and *two's complement* for signed numbers. An n-bit unsigned integer has a value in the range $0 \ldots 2^n-1$, inclusive. An n-bit two's complement integer has a value in the range $-2^{n-1} \ldots 2^{n-1} - 1$, inclusive. Most instruction sets provide two forms of most of the arithmetic operators: one for unsigned numbers and one for signed numbers. Even for languages in which integers are always signed, unsigned arithmetic is important for the manipulation of addresses (e.g., pointers).

An n-bit unsigned integer with binary representation $b_{n-1}\, b_{n-2}\, \ldots\, b_2\, b_1\, b_0$ has the value $\sum_{0 \le i < n} b_i 2^i$. Because the bit pattern corresponding to a given decimal value is non-obvious, and because bit patterns written as strings of 0's and 1's are cumbersome, computer scientists commonly represent integer values in *hexadecimal*, or base-16 notation. Hexadecimal uses the letters a to f as six additional digits, representing the values 10 to 15 in decimal (see Figure C-5.3). Because $2^4 = 16$, every digit in a hexadecimal number corresponds to exactly four bits of binary, making conversions between hexadecimal and binary trivial. In textual contexts, hexadecimal values are often written with a leading `0x`. Referring to Figure C-5.3, the hexadecimal value `0xabcd` corresponds to the binary value `1010 1011 1100 1101`. Similarly, `0x400` $= 2^{10} = 1024$, commonly written `1K`, and `0x100000` $= 2^{20} = 1048576$, commonly written `1M`. ■

EXAMPLE 5.3
Hexadecimal numbers

Perhaps the most obvious representation for signed integers would reserve one bit to indicate the sign ($+$ or $-$) and use the remaining $n - 1$ bits to represent the magnitude, as in unsigned numbers. Unfortunately, this approach requires different algorithms (and hence separate circuits) for addition and subtraction. The almost universally adopted alternative is called *two's complement* arithmetic. It capitalizes on the observation that arithmetic on unsigned n-digit numbers, when we ignore carries out of the left-most place, is actually arithmetic on what mathematicians call the *ring of integers modulo* 2^n. The sum $A + B$, for example, is really $(A + B) \bmod 2^n$. There is no particular reason, however, why we need to interpret the bit patterns on which we are doing our arithmetic as the numbers $0 \,..\, 2^n - 1$. We can actually pick any contiguous range of 2^n integers, anywhere on

the number line, and say that we're doing modulo arithmetic on them instead. In particular, we can pick the range $-2^{n-1} \ldots 2^{n-1} - 1$.

The smallest n-digit two's complement value, -2^{n-1}, is represented by a one followed by $n-1$ zeros. Successive values are obtained by repeatedly adding one, using ordinary place-value addition. This choice of representation has several desirable properties:

1. Non-negative numbers have the same bit patterns as they do in unsigned format.
2. The most significant bit of every negative number is one; the most significant bit of every non-negative number is zero.
3. A single addition algorithm works for all combinations of negative and non-negative numbers.

EXAMPLE 5.4
Two's complement

A list of 4-bit two's complement numbers appears in Figure C-5.4. ■

The addition algorithm for both unsigned and two's complement binary numbers is the obvious binary analogue of the familiar right-to-left addition of decimal numbers. Given a fixed word size, however we must consider the issue of *overflow*. By definition we should see overflow whenever the sum of two integers (not the bit patterns, but the actual integers they represent) is outside the range of values that can be represented in 2^n bits. For unsigned integers, this is easy: overflow occurs when we have a carry out of the most significant (left-most) place. For two's complement numbers, detection is somewhat trickier. First, note that the sum of a negative and a positive number can never overflow: the result is guaranteed to be closer to zero than the larger-magnitude addend. But if the sum is positive (it has a zero left-most bit), then there must have been a carry out of the left-most place, because one of the addends had a 1 in that place.

If we discard carries out of the left-most place (i.e., we stay within the ring of integers mod 2^n), then we can decree that two's complement overflow has occurred when we add two non-negative numbers and get an apparently negative result (because we wrapped past the largest positive number), or when we add two negative numbers and get an apparently non-negative result (because we wrapped past the smallest [largest magnitude] negative number). For example,

EXAMPLE 5.5
Overflow in two's complement addition

DESIGN & IMPLEMENTATION

5.3 How much is a megabyte?

The fact that $2^{10} \approx 10^3$ facilitates "back-of-the-envelope" approximations, but can sometimes lead to confusion when precision is required. Which meaning is intended when we see `1K` and `1M`? The answer, sadly, depends on context. Main memory sizes and addresses are typically measured with powers of two, while other quantities are measured with powers of ten. Thus a 1 GHz, 1 GB personal computer may start a new instruction 1,000,000,000 times per second, but have 1,073,741,824 bytes of memory. Its 1 TB hard disk will hold 10^{12} bytes.

0 1 1 1	7	1 1 1 1	−1
0 1 1 0	6	1 1 1 0	−2
0 1 0 1	5	1 1 0 1	−3
0 1 0 0	4	1 1 0 0	−4
0 0 1 1	3	1 0 1 1	−5
0 0 1 0	2	1 0 1 0	−6
0 0 0 1	1	1 0 0 1	−7
0 0 0 0	0	1 0 0 0	−8

Figure 5.4 **Four-bit two's complement numbers.** Note that there is a negative number (−8) that doesn't have a positive equivalent. There is only one zero, however.

with 4-bit two's complement numbers, 1100 + 0110 (−4 + 6) does not overflow, even though there is a carry out of the left-most place (which we discard). On the other hand, 0101 + 0100 (5 + 4) yields 1001, an apparently negative result for positive addends, and 1011 + 1100 (−5 + −4) yields 0111 in the low four bits, an apparently positive result for negative addends. Both of these cases indicate overflow.[1]

Different machines handle overflow in different ways. Some generate a fault (a hardware exception) on overflow. Some set a bit that can be tested in software. Some provide two `add` instructions, one for each option. Some provide a single `add` that can be made to do either, depending on the value of a bit in a special register.

It turns out that one can obtain the additive inverse of a two's complement number by flipping all the bits, adding one, and discarding any carry out of the left-most place (we defer a proof to Exercise C-5.7). Subtraction can thus be implemented almost trivially using an adder, by flipping the bits of the subtrahend, providing a one as the "carry" into the least-significant place, and "adding" as usual. Multiplication and division of signed numbers are a bit trickier than addition and subtraction, but still more or less straightforward.

Note that if we take any two's complement number and its additive inverse and add them together as if they were unsigned values, keeping the final carry bit, the sum will be 2^n. This observation is the source of the name "two's complement." Of course if we discard the carry bit we get zero, which is what one would expect of $k + (-k)$.

5.2.2 Floating-Point Arithmetic

Floating-point numbers are the computer equivalent of scientific notation: they consist of a *mantissa* or *significand*, *sig*, an *exponent*, *exp*, and (usually) a sign bit, *s*. The value of a (binary) floating-point number is then $-1^s \times sig \times 2^{exp}$. Prior to

1 Exercise C-5.6 considers an alternative but equivalent overflow detection mechanism, which is particularly easy to implement in hardware.

the mid-1980s, floating-point formats and semantics tended to vary greatly across brands and even models of computers. Different manufacturers made different choices regarding the number of bits in each field, their order, and their internal representation. They also made different choices regarding the behavior of arithmetic operators with respect to rounding, overflow, underflow,[2] invalid operations, and the representation of numbers that are almost—but not quite—too small to represent. With the completion in 1985 of IEEE standard number 754 (extended in 2008), the situation changed dramatically. Most processors developed in subsequent years conform to the formats and semantics of this standard.

The 1985 version of the IEEE 754 standard defines two sizes of floating-point numbers. *Single-precision* numbers have a sign bit, eight bits of exponent, and 23 bits of significand. They are capable of representing numbers whose magnitudes vary from roughly 10^{-38} to 10^{38}. *Double-precision* numbers have 11 bits of exponent and 52 bits of significand. They represent numbers whose magnitudes vary from roughly 10^{-308} to 10^{308}. The exponent is *biased* by subtracting the most negative possible value from it, so that it may be represented by an unsigned number. In single-precision, for example, the exponent 12 is represented by the value $12 - (-127) = 139 = \texttt{0x8b}$. The exponent -12 is represented by the value $-12 - (-127) = 115 = \texttt{0x73}$.

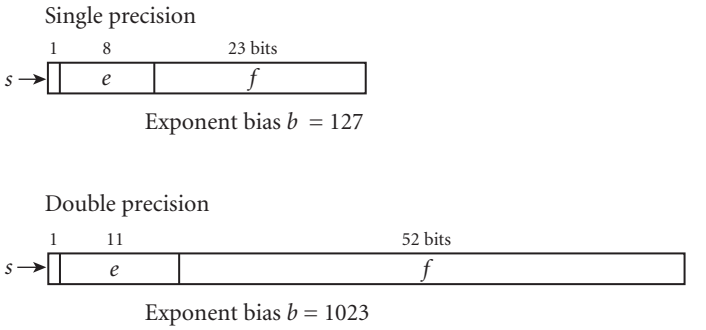
EXAMPLE 5.6
Biased exponents

Most values in the IEEE standard are *normalized* by shifting the significand until it is greater than or equal to 1, and less than 2. (The exponent is adjusted accordingly, so that the value represented doesn't change.) After normalization, we know that the leading bit of the significand will always be one, and need not be stored explicitly: to represent the value $1.something \times 2^{exp}$, we only need bits for the fractional part and the exponent. Exceptions to this rule occur near zero: very small numbers can be represented (with reduced precision) as $0.something \times 2^{min+1}$, where *min* is the smallest (most negative) exponent available in the format. Many older floating-point standards disallow such *subnormal numbers*, leading to a *gap* between zero and the smallest representable positive number that is larger than the gap between the two smallest representable positive numbers. Because it includes subnormals, the IEEE standard is said to provide for *gradual underflow*. Subnormal numbers are represented with a zero in the exponent field (denoting a maximally negative exponent) together with a nonzero fraction. (In the 1985 version of the standard, subnormal numbers were referred to as *denormal*.)

EXAMPLE 5.7
IEEE floating-point

Key conventions of the IEEE 754 standard are summarized in Figure C-5.5. In addition to the single- and double-precision formats shown here, the 2008 revision of the standard defines 16-bit *half-precision* and 128-bit *quad-precision* binary formats, as well as decimal (power-of-ten) formats in 32-, 64-, and 128-bit lengths. Both the old and new versions of the standard also permit vendor-defined "extended" formats that exceed the precision of some standard format

2 *Underflow* occurs when the result of a computation is too close to zero to represent—that is, when its exponent is a negative number whose magnitude is too large to represent in the number of available bits.



	e	f	Value
Zero	0	0	± 0
Infinity	$2b + 1$	0	$\pm \infty$
Normalized	$1 \leq e \leq 2b$	<*any*>	$\pm 1.f \times 2^{e-b}$
Denormalized	0	$\neq 0$	$\pm 0.f \times 2^{1-b}$
NaN	$2b + 1$	$\neq 0$	NaN

Figure 5.5 **The IEEE 754 floating-point standard**. For normalized numbers, the exponent is $e - 127$ or $e - 1023$, depending on precision. The significand is $(1 + f) \times 2^{-23}$ or $(1 + f) \times 2^{-52}$, again depending on precision. Field f is called the *fractional part*, or *fraction*. Bit patterns in which e is all ones (255 for single-precision, 2047 for double-precision) are reserved for infinities and NaNs. Bit patterns in which e is zero but f is not are used for subnormal (gradual underflow) numbers.

(this accommodates, among other things, the 80-bit internal format of legacy floating point in x86 processors). We focus here on the single- and double-precision binary formats, which remain the most widely used. ■

Floating-point arithmetic is sufficiently complicated that entire books have been written about it. Some of the characteristics of the IEEE standard of particular interest to compiler writers include:

- Zero is represented by a bit pattern consisting entirely of zeros. There is also (confusingly) a "negative zero," consisting of a sign bit of one and zeros in all other positions.
- Two bit patterns are reserved to represent positive and negative infinity. These values behave in predictable ways. For example, any positive number divided by zero yields positive infinity. Similarly, the arctangent of positive infinity is $\pi/2$.
- Certain other bit patterns are reserved for special "not-a-number" (NaN) values. These values are generated by nonsensical operations, such as square root of a negative number, addition of positive and negative infinity, or division of zero by zero. Almost any operation on an NaN produces another NaN. As a result, many algorithms can dispense with internal error checks: they can follow the steps that make sense in the absence of errors, and then check the final

result to make sure it's not an NaN. Some NaNs, not normally generated by arithmetic operations, can be set by the compiler explicitly to represent uninitialized variables or other special situations; these *signaling NaNs* produce a hardware exception if used.

- The bit patterns used to represent non-negative, non-NaN floating-point numbers are ordered in the same way as integers. As a result, an ordinary integer comparison operation can (in certain contexts) be used to determine which of two numbers is larger.

An excellent introduction to both integer and floating-point arithmetic, together with suggestions for further reading, can be found in David Goldberg's appendix to Hennessy and Patterson's architecture text [HP12, App. J].

✓ CHECK YOUR UNDERSTANDING

1. Explain how to compute the additive inverse (negative) of a two's complement number.
2. Explain how to detect overflow in two's complement addition.
3. Do two's complement numbers use a bit to indicate their sign? Explain.
4. Summarize the key features of IEEE 754 floating-point arithmetic.
5. What is the approximate range of single- and double-precision floating-point values? What is the precision (in bits) of each?
6. What is a floating-point NaN?

5.3 Instruction Set Architecture (ISA)

The instructions available on a given machine, and their encoding in machine language, are referred to as the *instruction set architecture* (ISA). Existing ISAs vary quite a lot, but all include instructions for:

Computation — arithmetic and logical operations, tests, and comparisons on values held in registers (and possibly in memory)

Data movement — loads from memory to registers, stores from registers to memory, copies from one register (or memory location) to another

Control flow — conditional and unconditional branches (`goto`s), subroutine calls and returns, traps into the operating system

As we shall see in Section C-5.4, there have been several points in history at which the dominant style of instruction set design has undergone significant change. In particular, in the early to mid-1980s, designers shifted from an emphasis on *complex instruction set computing* (CISC), which sought to maximize

the useful work performed per machine instruction, to *reduced instruction set computing* (RISC), which sought to maximize the number of instructions that could be completed per second. Some of the largest differences among machines today can be seen in those whose ISAs began their evolution before and after 1980. In Section C-5.4.5 we will consider one ISA in each camp: the x86, begun in 1976, and the ARM, begun in 1983.

Among ISAs still in widespread use, significant differences can be seen in *addressing modes*, which specify the locations of operands; condition testing and branching; and the bit-level encoding of instructions. We will consider the first two of these in Sections C-5.3.1 and C-5.3.2 below. In the area of encoding, the most important design decision is whether to specify each instruction and its operands in a fixed, constant number of bits (typically 32), or whether to use different numbers of bits for different instructions or different numbers of arguments. Fixed-length instructions have the benefit of uniformity: they make it easier to locate and decode successive instructions, thereby facilitating the construction of *pipelined* processors (to be discussed in Section C-5.4). Their principal disadvantage is that certain natural operations require more than 32 bits of encoding, and thus cannot be captured in a single instruction.

From an architectural and performance perspective, the distinction between CISC and RISC ISAs is no longer of great concern: modern implementations of CISC ISAs (e.g., all recent x86 and z Series processors) incorporate a hardware "front end" that translates the legacy ISA, on the fly, into a RISC-like internal form amenable to heavily pipelined execution.

5.3.1 Addressing Modes

One can imagine many different ways in which a computational or data movement instruction might specify the location of its operand(s)—its *address*, in a broad sense of the word. A given operand might be in a register, in memory, or, in the case of read-only constants, in the instruction itself (these latter are referred to as *immediate* values).

One of the standard features of RISC machines is that computational instructions operate only on values held in registers or the instruction: a load instruction must be used to bring a value from memory into a register before it can be used as an operand. CISC machines usually allow all or most computational instructions to access operands directly in memory. RISC machines are therefore said to provide a *load-store* or *register-register* architecture; CISC machines are said to provide a *register-memory* architecture.

For binary operations, instructions on many machines can specify three addresses—two sources and a destination. Others, including the x86, provide only two-address instructions—one of the operands is always overwritten by the result. Two-address instructions are more compact, but three-address instructions are more flexible—they allow both operands to be reused in subsequent operations.

If an operand is in memory, its address might be found in a register, in memory, or in the instruction, or it might be derived from some combination of values

in various locations. Instruction sets differ greatly in the *addressing modes* they provide to capture these various options. On a simple RISC machine, load and store instructions may support only the *displacement* addressing mode, in which the operand's address is found by adding some small constant (the *displacement*) to the value found in a specified register (the *base*). The displacement is contained in the instruction. Displacement addressing with respect to the frame pointer provides an easy way to access local variables. Displacement addressing with a displacement of zero is sometimes called *register indirect* addressing.

Some ISAs, including the Power family, SPARC, and ARM, also allow load and store instructions to use an *indexed* addressing mode, in which the operand's address is found by adding the values in two registers. Indexed addressing is useful for arrays: one register (the *base*) contains the address of the array; the second (the *index*) contains the offset of the desired element.

CISC machines typically provide the richest set of addressing modes, and allow them to be used in computational instructions, as well as in loads and stores. On the x86, for example, the address of an operand can be calculated by multiplying the value in one register by a small constant, adding the value found in a second register, and then adding another small constant, all in one instruction.

5.3.2 Conditions and Branches

All instruction sets provide a *branching* mechanism to update the program counter under program control. Branches allow compilers to implement conditional statements, subroutines, and loops. Conditional branches may be controlled in several ways. On many machines they use *condition codes*. As mentioned in Section C-5.1, condition codes are usually implemented as a set of bits in a special *processor status register*. All or most of the arithmetic, logical, and data-movement instructions update the condition codes as a side effect. The exact number of bits varies from machine to machine, but three and four are common: one bit each to indicate whether the instruction produced a zero value, a negative value, and/or an overflow or carry. To implement the following test, for example,

EXAMPLE 5.8

An `if` statement in x86 assembler

```
A := B + C
if A = 0 then
    body
```

a compiler for the x86[3] might generate

3 Readers familiar with the x86 should be warned that this example uses the assembler syntax of the GNU compiler collection (`gcc`) and its assembler, `gas`. This syntax differs in several ways from Microsoft and Intel assembler. Most notably, it specifies operands in the opposite order. The instruction `addl B, %eax`, for example, adds the value in `B` to the value in register `%eax` and leaves the result in `%eax`: in GNU assembler the *destination* operand is listed second. In Intel and Microsoft assembler it's the other way around: `addl B, %eax` would add the value in register `%ebx` to the value in `B` and leave the result in `B`.

```
        movl    C, %eax     ; move long-word C into register eax
        addl    B, %eax     ; add
        movl    %eax, A     ; and store
        jne     L1          ; branch (jump) if result not equal to zero
         body
L1:
```

The first three instructions all set the condition codes. The fourth (`jne`) tests the codes in the wake of the `movl` that stores to A. It branches if the codes indicate that the value was not zero. ■

EXAMPLE 5.9
Compare and test instructions

For cases in which the outcome of a branch depends on a value that has not just been computed or moved, most machines provide `compare` and `test` instructions. Again on the x86:

if A $\leq$ B then *body*

```
        movl    A, %eax     ; move long-word A into register eax
        cmpl    B, %eax     ; compare to B
        jg      L1          ; branch (jump) if greater
         body
L1:
```

if A $>$ 0 then *body*

```
        testl   %eax, %eax  ; compare %eax (A) to 0
        jle     L2          ; branch if less than or equal
         body
L2:
```

The x86 `cmpl` instruction subtracts its source operand from its destination operand and sets the condition codes according to the result, but it does *not* overwrite the destination operand. The `testl` instruction ands its two operands together and compares the result to zero. Most often, as shown here, the two operands are the same. When they are different, one is typically a *mask* value that allows the programmer or compiler to test individual bits or bits fields in the other operand. ■

Unfortunately, traditional condition codes make it difficult to implement some important performance enhancements. In particular, the fact that they are set by almost every instruction tends to preclude implementations in which logically unrelated instructions might be executed in between (or in parallel with) the instruction that tests a condition and the branch that relies on the outcome of the test. There are several possible ways to address this problem. The ARM and SPARC architectures make setting of the condition codes optional on an instruction-by-instruction basis. The Power architecture provides eight separate sets of condition codes; compare and branch instructions can specify the set to use. The MIPS has no condition codes (at least not for integer operations); it uses Boolean values in registers instead.

EXAMPLE 5.10
Conditional move

Several ISAs, including Power, SPARC, and recent generations of the x86, provide a *conditional move* instruction that copies one register into another if and

only if the condition codes are appropriately set. On the x86, the code fragment C := max(A, B) might naively be translated

```
    movl    A, %ecx
    movl    B, %edx
    cmpl    %edx, %ecx  ; compare %edx (A) to %ecx (B)
    jle     L1          ; branch if less than or equal
    movl    %ecx, C     ; store A to C
    jmp     L2
L1:
    movl    %edx, C     ; store B to C
L2:
```

With a conditional move instruction it can become the following instead:

```
    movl    B, %ecx
    movl    A, %edx
    cmpl    %ecx, %edx  ; compare %edx (A) to %ecx (B)
    cmovgl  %edx, %ecx  ; move A into %ecx if greater
    movl    %ecx, C     ; store to C
```

A few ISAs, including 32-bit ARM and IA-64 (Itanium), allow almost any instruction to be marked as conditional. This more general mechanism is known as *predication*. It allows an if... then ... else construct to be translated into straight-line (branch-less) code: instructions in the then and else paths are prefixed with complementary conditions, causing one path to take effect and the other to function as a sequence of *no-op*s—instructions that have no effect. When both paths are short, it may be cheaper (at least in some processor implementations) to execute the no-ops than it would have been to execute a branch.

✓ CHECK YOUR UNDERSTANDING

7. What is the most popular instruction set architecture for desktop and laptop machines?
8. What is the most popular instruction set architecture for tablets and cell phones?
9. What is the difference between big-endian and little-endian addressing?
10. What is the purpose of a cache?
11. Why do many machines have more than one *level* of cache?
12. How many processor cycles does it typically take to access primary (level-1) cache? How many cycles does it typically take to access main memory?

13. What is data *alignment*? Why do many processors insist upon it?
14. List four common formats (interpretations) for bits in memory.
15. What is IEEE standard number 754? Why is it important?
16. What are the tradeoffs between two-address and three-address instruction formats?
17. Describe at least five different addressing modes. Which of these are commonly supported on RISC machines?
18. What are condition codes? Why do some architectures not provide them? What do they provide instead?

5.4 Architecture and Implementation

The typical processor implementation consists of a collection of *functional units*, one (or more) for each logically separable facet of processor activity: instruction fetch, instruction decode, operand fetch from registers, arithmetic computation, memory access, write-back of results to registers, and so on. One could imagine an implementation in which all of the work for a particular instruction is completed before work on the next instruction begins, and in fact this is how the earliest computers were constructed. The problem with this organization is that most of the functional units are idle most of the time. Modern processor implementations have a substantially more complicated organization, in which the executions of many instructions *overlapping* one another in time. To generate fast code, a compiler must understand the details of this organization.

Pipelining is the most fundamental form of instruction overlap. Originally developed for supercomputers of the 1960s, it moved into single-chip processors with the RISC revolution of the 1980s. On a pipelined machine, functional units work like the stations on an assembly line, with different instructions passing through different pipeline *stages* concurrently. Pipelining appears today in even the most inexpensive personal computers, and in all but the simplest processors for the embedded market. A simple processor may have 3–6 pipeline stages. The ARM Cortex-A12 (used in many cell phones) and the Intel Core i7 (used in many laptops) have 11 and 14 stages, respectively. The "superpipelined" Intel Pentium 4E had 31.

By allowing (parts of) multiple instructions to execute in parallel, pipelining can dramatically increase the number of instructions that can be completed per second, but it is not a panacea. In particular, a pipeline will *stall* if the same functional unit is needed in two different instructions simultaneously, or if an earlier instruction has not yet produced a result by the time it is needed in a later instruction, or if the outcome of a conditional branch is not known (or guessed) by the time the next instruction needs to be fetched.

We shall see in Section C-5.5 that many stalls can be avoided by adding a little extra hardware and then choosing carefully among the various ways of translating a given construct into target code. A typical example occurs in the case of floating-point arithmetic, which tends to be much slower than integer arithmetic. Rather than stall the entire pipeline while executing a floating-point instruction, we can build a separate functional unit for floating-point math, and arrange for it to operate on a separate set of floating-point registers. In effect, this strategy leads to a *pair* of pipelines—one for integers and one for floating-point—that share their first few stages. The integer branch of the pipeline can continue to execute while the floating-point unit is busy, so long as subsequent instructions do not require the floating-point result. The need to reorder, or *schedule*, instructions so that those that conflict with or depend on one another are separated in time is one of the principal reasons why compiling for modern processors is hard.

5.4.1 Microprogramming

As technology advances, there are occasionally times when it becomes feasible to design machines in a very different way. During the 1950s and the early 1960s, the instruction set of a typical computer was implemented by soldering together large numbers of discrete components (transistors, capacitors, etc.) that performed the required operations. To build a faster computer, one generally designed new, more powerful instructions, which required extra hardware. This strategy had the unfortunate effect of requiring assembly language programmers (or compiler writers, though there weren't many of them back then) to learn a new language every time a new and better computer came along.

A fundamental breakthrough occurred in the early 1960s, when IBM hit upon the idea of *microprogramming*. Microprogramming allowed a company to provide the *same* instruction set across a whole line of computers, from inexpensive slow machines to expensive fast machines. The basic idea was to build a "micro-engine" in hardware that executed an interpreter program in "firmware." The interpreter in turn implemented the "machine language" of the computer—in this case, the IBM 360 instruction set. More expensive machines had fancier micro-engines, with more direct support for the instructions seen by the assembly-level programmer. The top-of-the-line machines had everything in hardware. In effect, the architecture of the machine became an abstract interface behind which hardware designers could hide implementation details, much as the interfaces of modules in modern programming languages allow software designers to limit the information available to users of an abstraction.

In addition to allowing the introduction of computer families, microprogramming made it comparatively easy for architects to extend the instruction set. Numerous studies were published in which researchers identified some sequence of instructions that commonly occurred together (e.g., the instructions that jump to a subroutine and update bookkeeping information in the stack), and then introduced a new instruction to perform the same function as the sequence. The new

instruction was usually faster than the sequence it replaced, and almost always shorter (and code size was more important then than now).

5.4.2 Microprocessors

A second architectural breakthrough occurred in the mid-1970s, when very large scale integration (VLSI) chip technology reached the point at which a simple microprogrammed processor could be implemented entirely on one inexpensive chip. The chip boundary is important because it takes much more time and power to drive signals across macroscopic output pins than it does across intrachip connections, and because the number of pins on a chip is limited by packaging issues. With an entire processor on one chip, it became feasible to build a commercially viable personal computer. Processor architectures of this era include the MOS Technology 6502, used in the Apple II and the Commodore 64, and the Intel 8080 and Zilog Z80, used in the Radio Shack TRS-80 and various CP/M machines. Continued improvements in VLSI technology led, by the mid-1980s, to 32-bit microprogrammed microprocessors such as the Motorola 68000, used in the original Apple Macintosh, and the Intel 80386, used in the first 32-bit IBM PCs.

From an architectural standpoint, the principal impact of the microprocessor revolution was to constrain, temporarily, the number of registers and the size of operands. Where the IBM 360 (*not* a single-chip processor) operated on 32-bit data, with 16 general-purpose 32-bit registers, the Intel 8080 operated on 8-bit data, with only seven 8-bit registers and a 16-bit stack pointer. Over time, as VLSI density increased, registers and instruction sets expanded as well. Intel's 32-bit 80386 was introduced in 1985.

5.4.3 RISC

By the early 1980s, several factors converged to make possible a third architectural breakthrough. First, VLSI technology reached the point at which a pipelined 32-bit processor with a sufficiently simple instruction set could be implemented on a single chip, *without* microprogramming. Second, improvements in processor speed were beginning to outstrip improvements in memory speed, increasing the relative penalty for accessing memory, and thereby increasing the pressure to keep things in registers. Third, compiler technology had advanced to the point at which compilers could often match (and sometimes exceed) the quality of code produced by the best assembly language programmers. Taken together, these factors suggested a *reduced instruction set computer* (RISC) architecture with a fast, all-hardware implementation, a comparatively low-level instruction set, a large number of registers, and an optimizing compiler.

The advent of RISC machines ran counter to the ever-more-powerful-instructions trend in processor design, but was to a large extent consistent with established trends for supercomputers. Supercomputer instruction sets had always

been relatively simple and low-level, in order to facilitate pipelining. Among other things, effective pipelining depends on having most instructions take the same, constant number of cycles to execute, and on minimizing dependences that would prevent a later instruction from starting execution before its predecessors have finished.

The most basic rule of processor performance holds that total execution time on any machine equals the number of instructions executed times the average number of cycles per instruction times the length in time of a cycle. What we might call the "CISC design philosophy" is to minimize execution time by reducing the number of instructions, letting each instruction do more work. What we might call the "RISC design philosophy" is to reduce the length of the cycle and the number of (nonoverlapped) cycles per instruction (CPI). Though once cast as design alternatives, these philosophies are not mutually exclusive: complex instructions can successfully be added to a RISC design, so long as their implementation does not compromise CPI or cycle time.

High performance processors attempt to minimize CPI by executing as many instructions as possible in parallel. One core of an IBM Power7, for example, can have as many as 120 instructions simultaneously "in flight" (and each processor chip has 8 cores). Some processors have very deep pipelines, allowing the work of an instruction to be divided into very short cycles. Many are *superscalar*: they have multiple parallel pipelines, and start more than one instruction each cycle. (This requires, of course, that the compiler and/or hardware identify instructions that do not depend on one another, so that parallel execution is semantically indistinguishable from sequential execution.) To minimize artificial dependences between instructions (as, for instance, when one instruction must finish using a register as an operand before another instruction overwrites that register with a new value), many machines perform *register renaming*, dynamically assigning logically independent uses of the same architectural register to different locations in a larger set of physical (implementation) registers. A high performance processor may actually execute mutually independent instructions *out of order* when it can increase instruction-level parallelism by doing so. These techniques dramatically increase implementation complexity but not architectural complexity; in fact, it is architectural *simplicity* that makes them possible.

5.4.4 Multithreading and Multicore

For 50 years, improvements in silicon fabrication technology have fueled a seemingly inexorable increase in the density of integrated circuits. This trend, first observed by Gordon Moore in 1965, has seen the number of transistors on a chip double roughly every two years since the mid 1960s—a million-fold increase since the early 1970s. Processor designers have used this amazing windfall in several major ways:

Faster clocks. Since smaller transistors can charge and discharge more quickly, higher-density chips can run at a higher clock rate. The Intel 8080 ran at

2 MHz in 1974. Rates in excess of 2 GHz (1000× faster) are commonplace today.

Instruction-level parallelism (ILP). As noted in the previous subsection, modern processors employ pipelined, superscalar, and out-of-order execution to keep a very large number of instructions "in flight," and to execute those instructions as soon as their operands become available.

Speculation. To keep the pipeline full, a modern processor guesses which way control will go at every branch, and *speculatively* executes instructions along the predicted control path. Some processors employ additional forms of speculation as well: they may, for example, guess the value that will be returned by a read that misses in the cache. So long as guesses are right, the processor avoids "unnecessary" waiting. It must always check after the fact, however, and be prepared to undo any erroneous operations in the event that a guess was wrong.

Larger caches. As noted in Sidebar C-5.2, caches play a critical role in coping with the processor-memory gap induced by higher clock rates. Higher VLSI density makes room for larger caches.

Unfortunately, by roughly 2004, the first three of these standard techniques had pretty much hit a dead end. Both faster clocks and speculation lead to very high energy consumption. To first approximation, a chip's energy requirements are proportional to its physical area and clock frequency. While caches take less energy than average (they're comparatively passive), the bookkeeping circuits required for speculation are very power-hungry. Where the 8080 consumed about 1.3 W, a desktop processor today may consume 130 W—more heat per unit area than the burner of a hot plate, and essentially at the limit of what we can cool without refrigeration. Simultaneously, ILP exploitation and speculative execution have approached the inherent limits of traditional sequential code. Bluntly put, modern single-core processors execute as many instructions in parallel as traditional programs will allow.

Robbed of the ability to run a single program faster, processor designers began building *multithreaded* and *multicore* chips that can run more than one program at once. Multithreading was introduced first. It allows several programs (threads), represented by several sets of registers and instruction fetching logic, to share the back end (execution units) of a single processor. In effect, the extra threads serve to fill "bubbles" (stalls) in the processor's pipeline. A multicore processor, by contrast, has the equivalent of two or more complete processors (cores) on a single chip (by convention, a single chip is referred to as "a processor," regardless of the number of cores). Compared to a high-end uniprocessor from 2004, the cores of a modern chip may run at a somewhat slower clock rate, and expend less energy on speculation and ILP discovery, in order to maximize performance per watt.

In moving to multicore processors, the computer industry has effectively given up on running conventional programs faster, and is banking instead on running better programs. This makes the multicore revolution very different from previ-

ous changes in design philosophy. Where previous changes were mostly invisible to programmers (code might perhaps have to be recompiled to make the best use of a new machine), the multicore revolution requires programs to be *rewritten* in some explicitly concurrent language. Successes in high-end scientific and commercial computing have demonstrated that this task is possible for expert programmers in certain problem domains. It is not yet clear whether it will be possible for "ordinary" programmers in arbitrary problem domains.

Most computers do several things at once: they update the display, check for mail, play music, and execute user commands by switching the processor from one task to another many times a second. With several cores available, each task can run on a different core, reducing the need for switching. But what will happen when we have 100 cores? Where will we find 100 runnable threads? This is perhaps the most vexing problem currently facing the field of computer systems.

✓ CHECK YOUR UNDERSTANDING

19. What is microprogramming? What breakthroughs did its invention make possible?
20. What technological threshold was crossed in the mid-1970s, enabling the introduction of microprocessors? What subsequent threshold, crossed in the early 1980s, made RISC machines possible?
21. What is pipelining?
22. Summarize the difference between the CISC and RISC philosophies in instruction set design.
23. Why do RISC machines allow only load and store instructions to access memory?
24. Name three CISC architectures. Name three RISC architectures. (If you're stumped, see the Summary and Concluding Remarks [Section C-5.6].)
25. How can the designer of a pipelined machine cope with instructions (e.g., floating-point arithmetic) that take much longer than others to compute?
26. Why are microprocessor clock rates no longer increasing?
27. Explain the difference between *multithreaded* and *multicore* processors.
28. Explain why the multicore revolution poses an unprecedented challenge for computer systems.

5.4.5 Two Example Architectures: The x86 and ARM

EXAMPLE 5.11
The x86 ISA

We can illustrate much of the variety in ISA design—including the CISC and RISC philosophies—by examining a pair of representative architectures. The x86

is the most widely used ISA in the server, desktop, and laptop markets. The original implementation, the 8086, was announced in 1978. Major changes were introduced in Intel's 8087, 80286, 80386, Pentium Pro, Pentium/MMX, Pentium III, and Pentium 4, and in AMD's K8 (Opteron). Though technically backward compatible, these changes were often out of keeping with the philosophy of earlier generations. The result is an architecture with numerous stylistic inconsistencies and special cases. While both AMD and Intel have trade names for the instruction set, the name "x86" is widely used to refer to it generically. When necessary, "x86-32" and "x86-64" are used to refer to the 32- and 64-bit versions, both of which are in widespread use today. Some vendors use "x64" to refer to the 64-bit version.

Early generations of the x86 were extensively microprogrammed. More recent generations still use microprogramming for the more complex portions of the instruction set, but simpler instructions are translated directly (in hardware) into between one and four microinstructions that are in turn fed to a heavily pipelined, RISC-like computational core. ■

EXAMPLE 5.12
The ARM ISA

The original version of the ARM architecture, developed by Acorn Computers of Cambridge, England, was announced in 1983. Acorn's contemporary descendant, ARM Holdings, oversees the evolution of the instruction set, and designs—but does not fabricate—implementations. The company licenses both the instruction set and the designs to scores of partner firms, which incorporate ARM processors in everything from toasters and fuel injectors to cell phones and tablet computers. ARM is, in fact, the best-selling architecture in the world, but its presence historically was limited mostly to embedded applications (i.e., device control). In recent years, ARM has come to dominate the cell phone and tablet market, bringing it into increasing competition with the x86. In general, x86 processors tend to have higher overall performance, while ARM processors tend to have higher performance per watt—a crucial advantage for mobile applications.

Like the x86, ARM has evolved considerably over time, and given its wide range of applications, it is available in a bewildering array of versions; these vary not only in speed and cost, but also in feature set. Unlike the x86, ARM never had 8- or 16-bit versions: until recently it was 32-bit only. A 64-bit extension, ARMv8, designed to compete in the desktop and server markets, was announced in 2011; the first commercial implementations became available in 2013. ■

Among the most significant differences between the x86 and ARM are their memory access mechanisms, their register sets, and the variety of instructions they provide. Like all RISC architectures, ARM allows only load and store instructions to access memory; all computation is done with values in registers (or in immediate fields of the current instruction). Like most CISC architectures, the x86 allows computational instructions to operate on values in either registers or memory. Like most RISC architectures, 64-bit ARM has 32 integer registers and 32 floating-point registers. On 32-bit ARM machines, there are only 16 integer registers (and only 16 are visible on a 64-bit machine when running in 32-bit mode). The x86, by contrast, has 16 registers of each kind when running in 64-bit mode, and only 8 in 32-bit mode. (There is also a separate set of 8 floating-point registers, 80 bits in length. These are used by an older set of floating-point instruc-

tions; they are increasingly ignored by modern compilers.) ARM provides many fewer distinct instructions than does the x86, and its instruction set is much more internally consistent; the x86 has a huge number of special cases. ARM instructions are normally 4 bytes long, though there is a special version of 32-bit mode called "Thumb" that provides 2-byte encodings of the most commonly used instructions. Instructions on the x86 vary from 1 to 15 bytes in length.

Memory Access and Addressing Modes

Although ARM is a register-register architecture, while the x86 is register-memory, the addressing modes of the two machines are actually quite similar: ARM has a richer set of options than many other RISC designs.

In 32-bit mode, an ARM address is formed by adding an *offset* to the value in a specified *base* register. The offset can be either an immediate *displacement* or the value in a second, *index* register. In the latter case, the offset can be shifted (*scaled*) up to 31 bit positions, effectively multiplying it by an arbitrary power of 2. With either kind of offset, the base register can optionally be updated (either before or after using its value), by adding or subtracting the (already shifted) offset. This *pre-* or *post-indexing* mechanism facilitates iteration through arrays. To economize on encoding bits, some of the addressing combinations are unavailable in 64-bit mode.

As we shall see under "Registers" below, 32-bit ARM assigns a register number to the program counter (PC), allowing that register to be used at the base in load and store instructions. This convention makes it easy to read values from the code of the running program—a trick that facilitates the construction of *position-independent code* (to be discussed in Section C-15.7.1). It also means that a branch is simply a store to the PC.

On the x86, an address is also formed by adding an offset to the value in a base register, but in this case the offset can reflect *both* an immediate displacement *and* the (possibly scaled) value in an index register. Scaling factors are less general than on ARM: possible values are 1, 2, 4, and 8. Pre- and post-increment options are also unavailable, though there are separate `push` and `pop` instructions that use the stack pointer (SP) as a base register, and automatically update it. A special PC-relative addressing mode is available in 64-bit mode, but not in 32-bit mode.

X86 instructions are two-address: the result of a computation overwrites one of the operands, which may be in either a register or memory. Computation is normally three-address on ARM (two sources and a destination can all be separate registers), but two-address when running in Thumb mode.

Registers

EXAMPLE 5.13
x86 and ARM register sets

The user-visible registers of the two architectures are illustrated pictorially in Figure C-5.6. As is immediately obvious, the ARM registers are both more numerous and more regular in structure than those of the x86. To a large extent this reflects the designs' respective histories. The 8086 was introduced in 1978 with 16-bit integer registers. (It was source-code compatible, though not binary compatible, with the earlier 8-bit 8080.) Intel expanded the registers to 32 bits in 1985 with

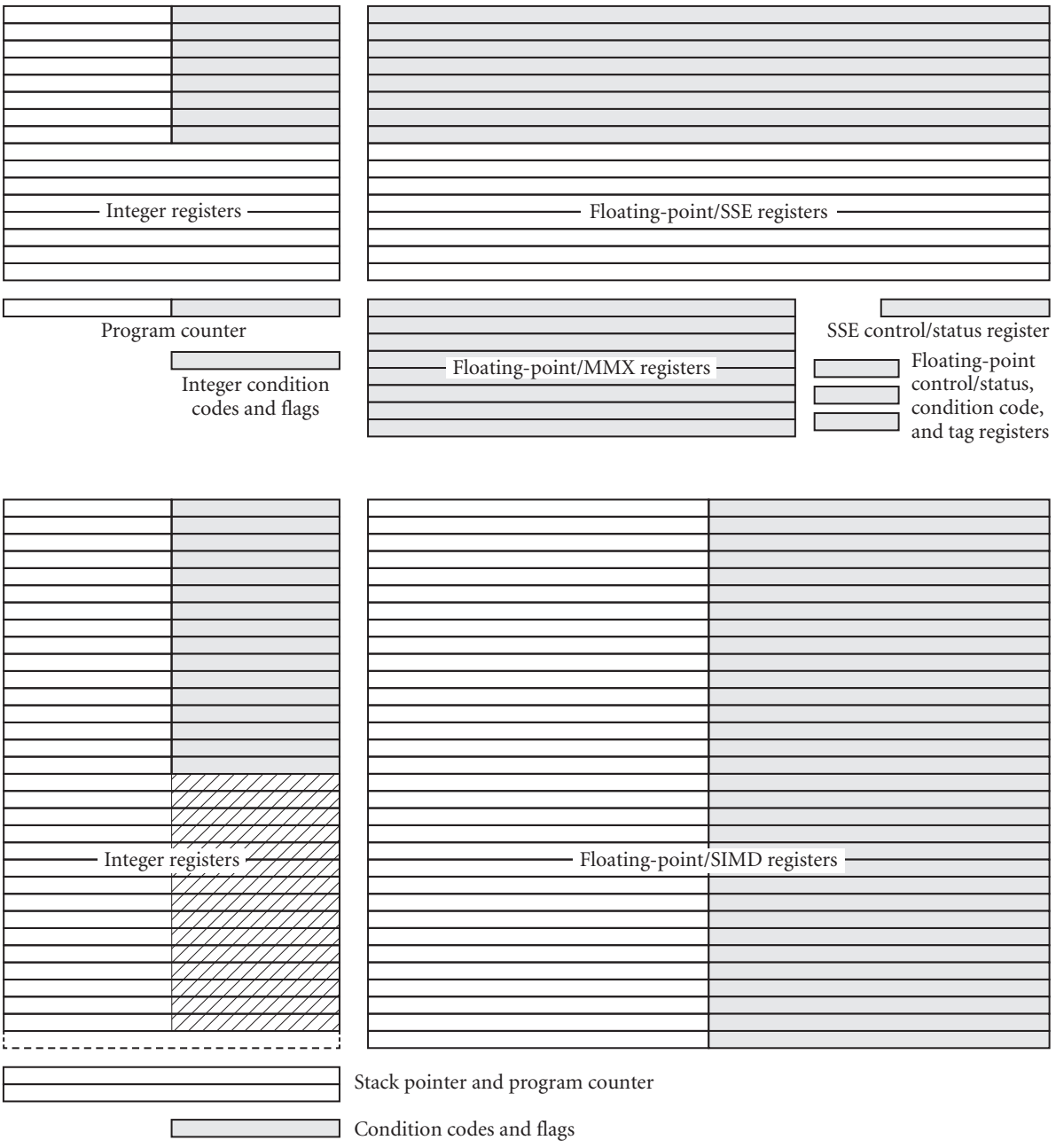


Figure 5.6 **User-visible registers of the x86-64 (top) and ARM v8 (bottom).** For both architectures, shaded areas indicate the subset visible in 32-bit mode. The last of the integer registers on ARM (shown with a dotted line) is virtual; it behaves as if it always contained a zero. The cross-hatched area indicates "banked" copies of the 32-bit registers, which are mapped into the bottom halves of the higher-numbered 64-bit registers. Other special registers, of use only in privileged code, are omitted for both architectures. Also omitted are the eight *segment registers* of the x86, which support the obsolete 80286 addressing system, and are not (for the most part) employed by modern compilers.

the 80386, and AMD expanded them again to 64 bits in 2000. The ARM, by contrast, has seen less re-engineering. It was introduced with 32-bit registers in 1983, and was only extended to 64 in 2011. ■

The x86-32 has eight 32-bit integer registers, plus the program counter and the processor status word, which includes the condition codes. For historical reasons, the integer registers are named `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `esp`, and `ebp`. They can be used interchangeably in most instructions, but certain instructions use them in special ways. Registers `eax` and `edx`, for example, are implicitly the destination registers for integer multiplication and division operations. Register `ecx` is read and updated implicitly by certain loop-control instructions. Registers `esi` and `edi` are used implicitly by instructions that copy, search, or compare strings of characters in memory. Register `esp` is used as a stack pointer; it is read and written implicitly by `push`, `pop`, and subroutine call/return instructions. Register `ebp` is typically used as a frame pointer; it is manipulated by instructions designed to allocate and deallocate stack frames.

For backward compatibility with 16-bit code, there are separate names for the lower halves of all eight integer registers: `ax`, `bx`, `cx`, `dx`, `si`, `di`, `sp`, and `bp`. Four of these (`ax`, `bx`, `ax`, and `ax`) have separate names for their upper and lower halves: `ah`, `al`, `bh`, `bl`, `ch`, `cl`, `dh`, and `dl`. The x86-64 doubles the length of the 32-bit registers, naming them `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `rsp`, and `rbp`. It then adds another 8, named `r8` through `r15`. Registerrbp is no longer used as a frame pointer in 64-bit mode.

Floating-point instructions were originally designed (in the 8087) to operate on a stack of eight additional registers, each 80 bits in length. Three 16-bit companion registers hold IEEE floating-point status and control, floating-point condition codes, and "tag" bits that indicate whether the values in the various floating-point registers are normal, subnormal, NaN, or garbage. All computation in this legacy "x87" portion of the instruction set is performed in extended precision; values are converted to and from IEEE single- and double-precision floating-point when written to or read from memory.

Vector instructions were added to the x86 with the Pentium/MMX in 1997. To avoid requiring the operating system to save additional state when switching between processes, MMX instructions were designed to share the x87 registers. In practice the arrangement proved less than ideal: the extra internal precision of x87 floating point could cause programs to behave differently than they did on other IEEE 754-compliant machines, and stack-based addressing impeded code improvement. Moreover MMX lacked support for floating-point vectors, and the small total number of registers made it difficult to use vectors and floating point in the same program. To a large degree, both x87 floating point and MMX have been supplanted by a series of extensions known as SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions), begun in 1999. These extensions employ a separate set of 128, 256, or 512–bit registers—8 of them in 32-bit mode, 16 in 64-bit mode—and provide full support for IEEE floating point. While some 32-bit compilers continue to use the older instructions and register file, 64-bit compilers typically use only SSE and AVX.

ARM v7 has a total of 48 registers: 16 integer and 32 floating-point, named `r0`–`r15` and `d0`–`d31`. Registers `r13`, `r14`, and `r15` double as the stack pointer (SP), link register (return address—LR), and program counter (PC), respectively. All of the integer registers are 32 bits wide. There is also a 32-bit processor status register that includes the condition codes.

To facilitate fast, low-power interrupt handling in embedded applications, with minimal saving and restoring of state, ARM provides separate "banked" copies of the SP and LR register for each of several different interrupt (privilege) levels. A so-called "fast interrupt" level has additional copies of `r8`–`r12`. While these banked copies are generally of interest only to systems software, they need to be mentioned in order to fully understand the 64-bit version of the ISA.

For ARM v8, designers increased the number of integer registers to 31, doubled their width, and named them `x0`–`x30`. In a convention common to RISC machines, a 32nd "virtual register" behaves as if it always contained a zero. As shown in Figure C-5.6, the lower halves of `x0`–`x15` overlap `r0`–`r15`. In `x16`–`x30`, the designers took the opportunity to overlap the banked copies of the 32-bit registers. This convention allows high-privilege-level 64-bit code (e.g., a virtual machine monitor) to more easily manipulate the state of medium-privilege-level 32-bit code (e.g., a guest operating system). To avoid security leaks, 64-bit code is never permitted to run at a lower privilege level than 32-bit code. The floating-point registers, for their part, were simply doubled in length, from 64 to 128 bits each. As in x86 SSE, they double as vector registers.

Register Conventions Beyond the special treatment given some registers in hardware, the designers of both the x86 and ARM recommend additional conventions to be enforced by software. On x86-32, register `ebp` is generally used for a frame pointer, whether or not the compiler makes use of special frame management instructions. Function values are returned in register `eax` (or in the pair `eax:edx` in the case of 64-bit return values). Any subroutine that modifies registers `ebx`, `esi`, or `edi` must save their old values in memory, and restore them before returning. Any caller that needs the values in `eax`, `ecx`, or `edx` must save them before making a call.

Additional conventions apply on x86-64. There is generally no frame pointer—`rsp` is used as the base when accessing data in the stack, and `rbp` is just an ordinary register. Moreover, the first six integer arguments to a subroutine are passed in registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`, respectively. If there are fewer arguments, these registers must be saved by the caller if their contents are needed later. Registers `rbx`, `rbp`, `r13`, `r14`, and `r15` must be saved by the callee. (Calling sequences will be discussed in more detail in Section 9.2.)

Conventions on ARM are similar. In 32-bit mode, in addition to `r13`, `r14`, and `r15` (SP, LR, and PC), which are special-cased in hardware, registers `r0`–`r3` are used by convention to hold the first four subroutine arguments and the return value, if any. `r9` is reserved for "platform-specific" purposes. `r12` is used as a scratch register for complex calls involving dynamic linking (to be discussed in Section C-15.7). In 64-bit mode, `x0`–`x7` are used for arguments and returns,

`r18` is platform-specific, and `r16` and `r17` are call-time scratch registers. In both modes, registers without special purposes are divided roughly 50-50 into caller-saves and callee-saves groups.

Instructions

While it can be difficult to count the instructions in a given instruction set (the x86 can branch on any of 16 different combinations of the condition codes; does this mean it has 16 conditional branch instructions, or one with 16 variants?), it is still clear that the x86 has more, and more complex, instructions than does ARM. Some of the features of the x86 not found on ARM include:

- Binary-coded decimal arithmetic (see Sidebar 7.4).
- Character-string search, compare, and copy operations.
- Bit string search and copy operations.
- Miscellaneous "combination" instructions. These perform the same task as some multi-instruction sequence, but require less code space and presumably run faster. Examples include subroutine calls and returns, stack operations, and loop control.
- Instructions to support the obsolete 80286 segmented memory system.

On the other hand, ARM provides:

- "Building-block" instructions that perform part of some operation too complex to propagate through the pipeline as a single instruction.
- "Saturating" arithmetic, which "holds" at the extreme values of a given integer type on overflow, rather than "rolling around" mod 2^{wordsize}.
- Combination shift-and-Φ instructions, for most arithmetic operations Φ.
- Predication.
- Pre- and post-decrement addressing.

More important than any difference in the number or types of instructions, however, is the difference in how those instructions are encoded. Like most CISC machines, the x86 places a heavy premium on minimizing code size (and thus the need for memory at run time), at the expense of comparatively difficult instruction decoding. Instructions range from 1 to 15 bytes in length, with a multitude of internal formats. Similar fields do not necessarily have the same length, or appear at the same offset, in different instructions. Operand specifiers vary in length depending on the choice of addressing mode. In 64-bit (16-register) mode, the 4th bit required to name a register is not contiguous with the other 3. One-byte *prefix codes* can be prepended to certain instructions to modify their behavior, causing them to repeat multiple times, access operands in a different segment of the 80286 address space, or lock the bus for atomic access to main memory.

The instruction encodings for ARM are substantially more regular, but they have their own peculiarities. In particular, where the myriad versions of the x86

share a single, common encoding, a 64-bit ARM machine supports three separate, quite different encodings, called A32, T32, and A64. (All three can be generated from a common assembly language.)

Like most RISC ISAs, A32 devotes 32 bits to every machine instruction. Its most unusual characteristic is the reservation of 4 bits in most instructions to encode predication conditions, and a 5th to indicate whether to set the condition codes. Operations that cannot be encoded in 32 bits (e.g., because they would require a 32-bit immediate value) must be expressed with multiple instructions. To load a 32-bit value into a register, for example, one might use a `MOV` instruction to load the lower half from a 16-bit intermediate value, followed by a `MOVT` (move top) instruction to load the upper half.

Uniform instruction length has the desirable property of simplifying the construction of a pipelined processor. Its principal shortcoming is that easily encoded (e.g., single-operand) instructions contain unneeded bits. Because it can capture such instructions in a smaller number of bytes, x86 code tends to be significantly denser than equivalent A32 code. To address this relative weakness, ARM introduced the T32 instruction set, also known as "Thumb." The most commonly executed, easily encoded instructions are specified in 16 bits in Thumb. Certain other instructions are encoded in 32 bits (though not with the same encoding as in A32). Because it lacks predication and some of the less common instructions, Thumb code tends to run slightly less quickly than equivalent A32 code. It is substantially more dense, however—a property of significant value in some embedded applications, where memory space or bandwidth may be scarce. A common practice for such applications is to compile the most performance-critical code to A32 and the rest to T32. The running program can switch from one instruction set to the other simply by executing a special branch instruction.

When designing x86-64, AMD was able to accommodate longer register names and new operations by adding an extra byte to existing instruction encodings. For ARM, fixed instruction lengths made this strategy infeasible. In a manner reminiscent of the previous design of Thumb, the company instead developed an entirely new encoding for A64—one that captures most preexisting instructions, a variety of new instructions (for 64-bit computation), and an extra bit per operand to accommodate expansion of the integer register set from 16 to 32. The key to making all of this fit in 32 bits was to reclaim the 4 bits devoted to predication in A32. The resulting instruction set and encoding are reminiscent of MIPS, Power, and SPARC, all of which have always had 32 integer registers each.

As noted under "Registers" above, ARM designers chose to identify the new integer registers of A64 with the "banked" register copies reserved for (privileged) exception handling code in A32. To prevent 64-bit applications from using this capability to "spy" on more privileged code, transitions between A64 and the existing A32 and T32 encodings occur only on exceptions, when they can be mediated by the operating system—user-level code cannot change into or out of 64-bit mode the way it can transition back and forth between A32 and T32.

CHECK YOUR UNDERSTANDING

29. Describe the most general (complex) addressing modes of the x86 and ARM architectures.
30. How many integer and floating-point registers are provided by each machine in 32-bit mode? In 64-bit mode? How wide are these registers?
31. Summarize the register usage conventions of the x86 and ARM.
32. Explain the utility of A64's "virtual" 32nd integer register.
33. List at least three "complex" instructions provided by the x86 instruction set but not provided by the ARM instruction set.
34. List at least two mechanisms provided by ARM but not by the x86.
35. Describe how floating-point support in the x86 has evolved over time.
36. Summarize the most important difference in how instructions are encoded on the x86 and ARM.
37. What is the purpose of ARM's T32 (Thumb) instruction encoding?
38. Contrast the strategies adopted by AMD and ARM in extending their respective architectures from 32 to 64 bits.

5.5 Compiling for Modern Processors

Programming a modern machine by hand, in assembly language, is a tedious undertaking. Values must constantly be shuffled back and forth between registers and memory, and operations that seem simple in a high-level language often require multiple instructions. With the rise of RISC-style instruction sets and implementations, complexity that was once hidden in microcode has been exported to the compiler. Fortunately, compilers don't get bored or make careless mistakes, and can easily deal with comparatively primitive instructions. In fact, when compiling for recent implementations of the x86, compilers generally limit themselves to a small, RISC-like subset of the instruction set, which the processor can pipeline effectively. Old programs that make use of more complex instructions still run, but not as fast; they don't take full advantage of the hardware.

EXAMPLE 5.14
Performance ≠ clock rate

The real difficulty in compiling for modern processors lies not in the need to use primitive instructions, but in the need to keep the pipeline full and to make effective use of registers. Early in this century, a user who traded in, say, a Pentium III PC for one with a Pentium 4 would typically find that while old programs ran faster on the new machine, the speed improvement was nowhere near as dramatic as the difference in clock rates would have led one to expect. Improvements would generally be better if one could obtain new program versions that were compiled with the newer processor in mind. ■

5.5.1 Keeping the Pipeline Full

Four main problems may cause a pipelined processor to stall:

1. *Cache misses.* A load instruction or an instruction fetch may miss in the cache.
2. *Resource hazards.* Two concurrently executing instructions may need to use the same functional unit at the same time.
3. *Data hazards.* An instruction may need an operand that has not yet been produced by an earlier but still executing instruction.
4. *Control hazards.* Until the outcome (and target) of a branch instruction is determined, the processor does not know the location from which to fetch subsequent instructions.

All of these problems are amenable, at least in part, to both hardware and software solutions. On the hardware side, misses can generally be reduced by building larger or more highly associative caches.[4] Resource hazards, likewise, can be addressed by building multiple copies of the various functional units (though most processors don't provide enough to avoid all possible conflicts). Misses, resource hazards, and data hazards can all be addressed by *out-of-order* execution, which allows a processor (at the cost of significant design complexity, chip area, and power consumption) to consider a lengthy "window" of instructions, and make progress on any of them for which operands and hardware resources are available.

Branches constitute something like 10% of all instructions in typical programs,[5] so even a one-cycle stall on every branch could be expected to slow down execution by 9% on average. On a deeply pipelined machine one might naively expect to stall for more like five or even ten cycles while waiting for a new program counter to be computed. To avoid such intolerable delays, most high-performance processors incorporate hardware to *predict* the outcome of each branch, based on past behavior, and to execute speculatively down the predicted path, in a way that can be "rolled back" in the event of misprediction. To the extent that it predicts correctly, such a processor can avoid control hazards altogether.

On the software side, the compiler has a major role to play in keeping the pipeline full. For any given source program, there is an unbounded number of

---

4 The degree of *associativity* of a cache is the number of distinct lines in the cache in which the contents of a given memory location might be found. In a one-way associative (*direct-mapped*) cache, each memory location maps to only one possible line in the cache. If the program uses two locations that map to the same line, the contents of these two locations will keep *evicting* each other, and many misses will result. More highly associative caches are slower, but suffer fewer such conflicts.

5 This is a very rough number. For the SPEC2000 benchmarks, Hennessy and Patterson report percentages varying from 1 to 25 [HP12, 3rd ed., pp. 138–139].

possible translations into machine code. In general we should prefer shorter translations over longer ones, but we must also consider the extent to which various translations will utilize the pipeline. On an in-order processor (one that always executes instructions in the order they appear in the machine language program), a stall will inevitably occur whenever a load is followed immediately by an instruction that needs the loaded value, because even first-level cache requires at least one extra cycle to respond. A stall may also occur when the result of a slow-to-complete floating-point operation is needed too soon by another instruction, when two concurrently executing instructions need the same functional unit in the same cycle, or, on a superscalar processor, when an instruction that uses a value is executed concurrently with the instruction that produces it. In all these cases performance may improve significantly if the compiler chooses a translation in which instructions appear in a different order.

The general technique of reordering instructions at compile time so as to maximize processor performance is known as *instruction scheduling*. On an in-order processor the goal is to identify a valid order that will minimize pipeline stalls at run time. To achieve this goal the compiler requires a detailed model of the pipeline. On an out-of-order processor the goal is simply to maximize *instruction-level parallelism* (ILP): the degree to which unrelated instructions lie near one another in the instruction stream (and thus are likely to fall within the processor's instruction window). A compiler for such an out-of-order machine may be able

DESIGN & IMPLEMENTATION

5.4 Delayed branch instructions

Successful pipelining depends on knowing the address of the next instruction before the current instruction has completed, or has even been fully decoded. With fixed-size instructions a processor can infer this address for straight-line code, but not for the code that follows a branch. In an attempt to minimize the impact of branch delays, several early RISC machines provided *delayed branch* instructions: with these, the instruction immediately after the branch would be executed regardless of the outcome of the branch.

Unfortunately, as architects moved to more aggressive, deeply pipelined processor implementations, the number of cycles required to correctly resolve a branch became more than one could cover with a single additional instruction. A few processors were designed with an architecturally visible branch delay of more than one cycle, but this proved not to be a viable strategy: it was simply too difficult for the compiler to find enough unrelated instructions to schedule into the slots. Instead, modern processors invariably rely on a hardware *branch predictor* to guess the outcome and targets of branches early, so that the pipeline can continue execution. That said, even when hardware is able to predict the outcome of branches, it can be useful for the compiler to do so also, in order to schedule instructions to minimize load delays in the most likely cross-branch code paths.

to make do with a less detailed processor model. At the same time, it may need to ensure a higher degree of ILP, since out-of-order execution tends to be found on machines with several pipelines.

Instruction scheduling can have a major impact on resource and data hazards. We will consider the topic of instruction scheduling in some detail in Section C-17.6. In the remainder of the current subsection we focus on the case of loads, where even an access that hits in the cache has the potential to delay subsequent instructions.

Software techniques to *reduce* the incidence of cache misses typically require large-scale restructuring of control flow or data layout. Though aggressive optimizing compilers may reorganize loops for better cache locality, especially in scientific programs (a topic we will consider in Section C-17.7.2), most simply assume that every memory access will hit in the primary cache, and aim to *tolerate* the delay that such a hit entails. The hit assumption is generally a good one: most programs on most machines find their data in (some level of) the cache more than 90% of the time (often over 99%). The goal of the compiler is to make sure that the pipeline can continue to operate during the time that it takes the cache to respond.

Consider a load instruction that hits in the primary cache. The number of cycles that must elapse before a subsequent instruction can use the result is known as the *load delay*. Even the fastest caches induce a one-cycle load delay. If the instruction immediately after a load attempts to use the loaded value, a one-cycle *load penalty* (a pipeline stall) will occur. Longer pipelines can have load delays of two or even three cycles.

EXAMPLE 5.15
Filling a load delay slot

To avoid load penalties (in the absence of out-of-order execution), the compiler may schedule one or more unrelated instructions into the *delay slot*(s) between a load and a subsequent use. In the following code, for example, a simple in-order pipeline might incur a one-cycle penalty between the second and third instructions:

```
r2 := r1 + r2
r3 := A              -- load
r3 := r3 + r2
```

If we swap the first two instructions, the penalty goes away:

```
r3 := A              -- load
r2 := r1 + r2
r3 := r3 + r2
```

The second instruction gives the first instruction time enough to retrieve A before it is needed in the third instruction. ■

To maintain program correctness, an instruction-scheduling algorithm must respect all *dependences* among instructions. These dependences come in three varieties:

Flow dependence (also called *true* or *read-after-write* dependence): a later instruction uses a value produced by an earlier instruction.

Anti-dependence (also called *write-after-read* dependence): a later instruction overwrites a value read by an earlier instruction.

Output dependence (also called *write-after-write* dependence): a later instruction overwrites a value written by a previous instruction.

EXAMPLE 5.16
Renaming registers for scheduling

A compiler can often eliminate anti- and output dependences by *renaming* registers. In the following, for example, anti-dependences prevent us from moving either the instruction before the load or the one after the add into the delay slot of the load:

```
r3 := r1 + 3        -- immovable ↕
r1 := A             -- load
r2 := r1 + r2
r1 := 3             -- immovable ↕
```

If we use a different register as the target of the load, however, then either instruction can be moved:

```
r3 := r1 + 3        -- movable ↓          r5 := A             -- load
r5 := A             -- load      becomes  r3 := r1 + 3
r2 := r5 + r2                             r1 := 3
r1 := 3             -- movable ↑          r2 := r5 + r2
```

■

The need to rename registers in order to move instructions can increase the number of registers needed by a given stretch of code. To maximize opportunities for concurrent execution, out-of-order processor implementations may perform

DESIGN & IMPLEMENTATION

5.5 Delayed load instructions

In order to enforce the flow dependence between a load of a register and its subsequent use, a processor must include so-called *interlock* hardware. To minimize chip area, several of the very early RISC processors provided this hardware only in the case of cache misses. The result was an architecturally visible *delayed load* instruction similar to the delayed branches discussed in Sidebar C-5.4. The value of the register targeted by a delayed load was undefined in the immediately subsequent instruction slot. Filling the delay slot of a delayed load with an unrelated instruction was thus a matter of correctness, not just of performance. If a compiler was unable to find a suitable "real" instruction, it had to fill the delay slot with a *no-op* (nop). More recent RISC machines have abandoned delayed loads; their implementations are fully interlocked. Within processor families old binaries continue to work correctly; the (nop) instructions are simply redundant.

register renaming dynamically in hardware, as noted in Section C-5.4.3. These implementations possess more physical registers than are visible in the instruction set. As instructions are considered for execution, any that use the same architectural register for independent purposes are given separate physical copies on which to do their work. If a processor does not perform hardware register renaming, then the compiler must balance the desire to eliminate pipeline stalls against the desire to minimize the demand for registers (so that they can be used to hold loop indices, local variables, and other comparatively long-lived values).

5.5.2 Register Allocation

A load-store architecture explicitly acknowledges that moving data between registers and memory is expensive. A store instruction costs a minimum of one cycle—more if several stores are executed in succession and the memory system can't keep up. A load instruction costs a minimum of one or two cycles (depending on whether the delay slot can be filled), and can cost scores or even hundreds of cycles in the event of a cache miss. In order to minimize the use of loads and stores, a good compiler must keep things in registers whenever possible. We saw an example in Chapter 1: the most striking difference between the "optimized" code of Example 1.2 and the naive code of Figure 1.7 is the absence in the former of most of the loads and stores. As improvements in processor speed outstripped improvements in memory speed through the 1980s and 90s, the cost in cycles of a cache miss continued to increase, making good register usage increasingly important.

Register allocation is typically a two-stage process. In the first stage the compiler identifies the portions of the abstract syntax tree that represent *basic blocks*: straight-line sequences of code with no branches in or out. Within each basic block it assigns a "virtual register" to each loaded or computed value. In effect, this assignment amounts to generating code under the assumption that the target machine has an unbounded number of registers. In the second stage, the compiler maps the virtual registers of an entire subroutine onto the architectural (hardware) registers, using the same architectural register when possible to hold different virtual registers at different times, and *spilling* virtual registers to memory when there aren't enough architectural registers to go around.

EXAMPLE 5.17

Register allocation for a simple loop

We will examine this two-stage process in more detail in Section C-17.8. For now, we illustrate the ideas with a simple example. Suppose we are compiling a function that computes the variance σ^2 of the contents of an n-element vector. Mathematically,

$$\sigma^2 = \frac{1}{n}\sum_i (x_i - \overline{x})^2 = \left(\frac{1}{n}\sum_i x_i^2\right) - \overline{x}^2$$

where $x_0 \ldots x_{n-1}$ are the elements of the vector, and $\overline{x} = 1/n \sum_i x_i$ is their average. In pseudocode,

```
 1.        v1 := &A              -- pointer to A[1]
 2.        v2 := n               -- count of elements yet to go
 3.        w1 := 0.0             -- sum
 4.        w2 := 0.0             -- squares
 5.        goto L2
 6.   L1:  w3 := *v1             -- A[i] (floating point)
 7.        w1 := w1 + w3         -- accumulate sum
 8.        w4 := w3 × w3
 9.        w2 := w2 + w4         -- accumulate squares
10.        v1 := v1 + 8          -- 8 bytes per double-word
11.        v2 := v2 − 1          -- decrement count
12.   L2:  if v2 > 0 goto L1
13.        w5 := w1 / n          -- average
14.        w6 := w2 / n          -- average of squares
15.        w7 := w5 × w5         -- square of average
16.        w8 := w6 − w7
17.        ...                   -- return value in w8
```

Figure 5.7 Pseudo-assembly code for a vector variance computation.

```
double sum := 0
double squares := 0
for int i in 0 . . n−1
    sum +:= A[i]
    squares +:= A[i] × A[i]
double average := sum / n
return (squares / n) − (average × average)
```

After some simple code improvements and the assignment of virtual registers, the assembly language for this function on a modern machine is likely to look something like Figure C-5.7. This code uses two integer virtual registers (v1 and v2) and eight floating-point virtual registers (w1–w8). For each of these we can compute the range over which the value in the register is useful, or *live*. This range extends from the point at which the value is defined to the last point at which the value is used. For register w4, for example, the range is only one instruction long, from the assignment at line 8 to the use at line 9. For register v1, the range is the union of two subranges, one that extends from the assignment at line 1 to the use (and redefinition) at line 10, and another that extends from this redefinition around the loop to the same spot again.

Once we have calculated live ranges for all virtual registers we can create a mapping onto the architectural registers. We can use a single architectural register for two virtual registers only if their live ranges do not overlap. If the number of architectural registers required is larger than the number available on the machine (after reserving a few for such special values as the stack pointer), then at various points in the code we shall have to write (spill) some of the virtual registers to memory in order to make room for the others.

```
1.        r1 := &A
2.        r2 := n
3.        f1 := 0.0
4.        f2 := 0.0
5.        goto L2
6.   L1:  f3 := *r1              -- no delay
7.        f1 := f1 + f3          -- 1-cycle wait for f3
8.        f3 := f3 × f3          -- no delay
9.        f2 := f2 + f3          -- 4-cycle wait for f3
10.       r1 := r1 + 8           -- no delay
11.       r2 := r2 − 1           -- no delay
12.  L2:  if r2 > 0 goto L1      -- no delay
13.       f1 := f1 / n
14.       f2 := f2 / n
15.       f1 := f1 × f1
16.       f1 := f2 − f1
17.       ...                    -- return value in f1
```

Figure 5.8 **The vector variance example with architectural registers assigned.** Also shown in the body of the loop are the number of stalled cycles that can be expected on a simple in-order pipelined machine, assuming a one cycle penalty for loads, two cycle penalty for floating-point adds, and four cycle penalty for floating-point multiplies.

In our example program, the live ranges for the two integer registers overlap, so they will have to be assigned to separate architectural registers. Among the floating-point registers, w1 overlaps with w2–w4, w2 overlaps with w3–w5, w5 overlaps with w6, and w6 overlaps with w7. There are several possible mappings onto three architectural floating-point registers, one of which is shown in Figure C-5.8. ■

Interaction with Instruction Scheduling

From the point of view of execution speed, the code in Figure C-5.8 has at least two problems. First, of the seven instructions in the loop, nearly half are devoted to bookkeeping: updating the pointer into the array, decrementing the loop count, and testing the terminating condition. Second, when run on a pipelined machine with in-order execution, the code is likely to experience a very high number of stalls. Exercise C-5.21 explores a first step toward addressing the bookkeeping overhead. We consider the stalls below, and return to both problems in more detail in Chapter 17.

EXAMPLE 5.18
Register allocation and instruction scheduling

We noted in Section C-5.5.1 that floating-point instructions commonly employ a separate, longer pipeline. Because they take more cycles to complete, there can be a significant delay before their results are available for use in other instructions. Suppose that floating-point add and multiply instructions must be followed by two and four cycles, respectively, of unrelated computation (these are modest figures; real machines often have longer delays). Also suppose that the result of a load is not available for a modest one-cycle delay. In the context of our

```
 1.        r1 := &A
 2.        r2 := n
 3.        f1 := 0.0
 4.        f2 := 0.0
 5.        goto L2
 6.   L1:  f3 := *r1
 7.        r1 := r1 + 8              -- no delay
 8.        f4 := f3 × f3             -- no delay
 9.        f1 := f1 + f3             -- no delay
10.        r2 := r2 − 1              -- no delay
11.        f2 := f2 + f4             -- 1-cycle wait for f4
12.   L2:  if r2 > 0 goto L1         -- no delay
13.        f1 := f1 / n
14.        f2 := f2 / n
15.        f1 := f1 × f1
16.        f1 := f2 − f1
17.        ...                       -- return value in f1
```

Figure 5.9 The vector variance example after instruction scheduling. All but one cycle of delay has been eliminated. Because we have hoisted the multiply above the first floating-point add, however, we need an extra architectural floating-point register.

vector variance example, these delays imply a total of five stalled cycles in every iteration of the loop, even if the hardware successfully predicts the outcome and target of the branch at the bottom. Added to the seven instructions themselves, this implies a total of 12 cycles per loop iteration (i.e., per vector element).

By rescheduling the instructions in the loop (Figure C-5.9) we can eliminate all but one cycle of stall. This brings the total number of cycles per iteration down to only eight, a reduction of 33%. The savings comes at a cost, however: we now execute the multiply instruction before the first floating-point add, and must use an extra architectural register to hold on to the add's second argument. This effect is not unusual: instruction scheduling has a tendency to overlap the live ranges of virtual registers whose ranges were previously disjoint, leading to an increase in the number of architectural registers required. ■

On a machine with out-of-order execution, hardware is likely (with the assistance of register renaming) to transform the code of Figure C-5.8 into something akin to Figure C-5.9 automatically on the fly, at the expense of chip area and density. As of this writing, there is still considerable debate in the architecture community regarding the relative merits of static (compiler) and dynamic (hardware) scheduling.

The Impact of Subroutine Calls

The register allocation scheme outlined above depends implicitly on the compiler being able to see all of the code that will be executed over a given span of time (e.g., an invocation of a subroutine). But what if that code includes calls to other subroutines? If a subroutine were called from only one place in the program,

we could allocate registers (and schedule instructions) across both the caller and the callee, effectively treating them as a single unit. Most of the time, however, a subroutine is called from many different places in a program, and the code improvements that we should like to make in the context of one caller may be different from the ones that we should like to make in the context of a different caller. For small, simple subroutines, the compiler may actually choose to expand a copy of the code at each call site, despite the resulting increase in code size. This *inlining* of subroutines can be an important form of code improvement, particularly for object-oriented languages, which tend to have very large numbers of very small subroutines.

When inlining is not an option, most compilers treat each subroutine as an independent unit. When a body of code for which we are attempting to perform register allocation makes a call to a subroutine, there are several issues to consider:

- Parameters must generally be passed. Ideally, we should like to pass them in registers.
- Any registers that the callee will use internally, but which contain useful values in the caller, must be spilled to memory and then reread when the callee returns.
- Any variables that the callee might load from memory, but which have been kept in a register in the caller, must be written back to memory before the call, so that the callee will see the current value.
- Any variables to which the callee might store a value in memory, but which have been kept in a register in the caller, must be reread from memory when the callee returns, so that the caller will see the current value.

If the caller does not know exactly what the callee might do (this is often the case—the callee might not have been compiled yet), then the compiler must make conservative assumptions. In particular, it must assume that the callee reads and writes *every* variable visible in its scope. The caller must write any such variable back to memory prior to the call, if its current value is (only) in a register. If it needs the value of such a variable after the call, it must reread it from memory.

With perfect knowledge of both the caller and the callee, we could arrange across subroutine calls to save and restore precisely those registers that are both

DESIGN & IMPLEMENTATION

5.6 In-line subroutines

Subroutine inlining presents, to a large extent, a classic time-space tradeoff. Inlining one instance of a subroutine replaces a relatively short calling sequence with a subroutine body that is typically significantly longer. In return, it avoids the execution overhead of the calling sequence, enables the compiler to perform code improvement across the call without performing interprocedural analysis, and typically improves locality, especially in the L1 instruction cache.

in use in the caller and needed (for internal purposes) in the callee. Without this knowledge, we can choose either for the caller to save and restore the registers it is using, before and after the call, or for the callee to save and restore the registers it needs internally, at the top and bottom of the subroutine. In practice it is conventional to choose the latter alternative for at least some static subset of the register set, for two reasons. First, while a subroutine may be called from many locations, there is only one copy of the subroutine itself. Saving and restoring registers in the callee, rather than the caller, can save substantially on code size. Second, because many subroutines (particularly those that are called most frequently) are very small and simple, the set of registers used in the callee tends, on average, to be smaller than the set in use in the caller. We will look at subroutine calling sequences and inlining in more detail in Sections 9.2 and 9.2.4, respectively.

✓ CHECK YOUR UNDERSTANDING

39. List the four principal causes of pipeline stalls.
40. What is a pipeline *interlock*?
41. What is a *delayed branch* instruction? A *delayed load* instruction?
42. What is *instruction scheduling*? Why is it important on modern machines?
43. What is the impact of out-of-order execution on compile-time instruction scheduling?
44. What is *branch prediction*? Why is it important?
45. Describe the interaction between instruction scheduling and register allocation.
46. What is the *live range* of a register?
47. What is *subroutine inlining*? What benefits does it provide? When is it possible? What is its cost?
48. Summarize the impact of subroutine calls on register allocation.

5.6 Summary and Concluding Remarks

Computer architecture has a major impact on the sort of code that a compiler must generate, and the sorts of code improvements it must effect in order to obtain good performance. Since the early 1980s, the trend in processor design has been to equip the compiler with more and more knowledge of the low-level details of processor implementation, so that the generated code can use the implementation to its fullest. This trend has blurred the traditional dividing line

between processor architecture and implementation: while a compiler can generate correct code based on an understanding of the architecture alone, it cannot generate fast code unless it understands the implementation as well. In effect, timing issues that were once hidden in the microcode of microprogrammed processors (and which made microprogramming an extremely difficult and arcane craft) have been exported into the compiler.

In the first several sections of this chapter we surveyed the organization of memory and the representation of data (including integer and floating-point arithmetic), the variety of typical assembly language instructions, and the evolution of modern architectures and implementations. As examples we compared the x86 and ARM. In the final section we discussed why compiling for modern machines is hard. The principal tasks include *instruction scheduling*, for load and branch delays and for multiple functional units, and *register allocation*, to minimize memory traffic. We noted that there is often a tension between these tasks, and that both are made more difficult by frequent subroutine calls.

As of 2015 there are four principal commercial RISC architectures: ARM (Apple, Motorola, nVidia, Qualcomm, Texas Instruments, and scores of others), MIPS (NEC, Toshiba, Freescale, and many others), Power/PowerPC (IBM, Freescale), and SPARC (Oracle, Texas Instruments, Fujitsu, and several others). ARM processors dominate the embedded and mobile markets, and are moving into laptops, desktops, and servers. MIPS processors, by contrast, began in the desktop and server space, but are now confined mainly to embedded devices. SPARC and Power processors are sold primarily to the server market. PowerPC processors appear in several brands of video game consoles.

Despite the burden of backward compatibility, the x86 overwhelmingly dominates the desktop and server market, thanks to the marketing prowess of IBM, Intel, and Microsoft, and to the engineering prowess of Intel and AMD, which have successfully decoupled the architecture from the implementation. The z architecture, for its part, enjoys a virtual monopoly in mainframe computing. While modern implementations of the x86 and z continue to implement their full respective ISAs, they do so on top of pipelined implementations with uncompromised performance.

With growing demand for a 64-bit address space, a major battle developed in the x86 world around the turn of the century. Intel undertook to design an entirely new (and very different) instruction set for their IA-64/Itanium line of processors. They provided an x86 compatibility mode, but it was implemented in a separate portion of the processor—essentially a Pentium subprocessor embedded in the corner of the chip. Application writers who wanted speed and address space 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 x86 processors. In designing its 64-bit extension, ARM has taken an intermediate approach: its 32- and 64-bit

modes share registers, and have essentially the same instructions, but use different instruction encodings.

As ARM pushes to claim a portion of the laptop/desktop/server market, and as Intel and AMD move to claim a portion of the mobile space, the ARM and x86 will undoubtedly come into more direct competition, likely resulting in ever more diverse implementations and instruction set extensions. In the development of extensions, both the CISC and RISC "design philosophies" are still very much alive [SW94]. The "CISC-ish" philosophy suggests that newly available resources (e.g., increases in chip area) be used to implement functions that would otherwise have to occur in software, such as decimal arithmetic, compression and encryption, media transcoding, or transactional synchronization (to be discussed in Section 13.4.4). The "RISC-ish" philosophy suggests that resources 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 cycle time. Depending on one's point of view, such "data-parallel" extensions as vector and graphic accelerators may be consistent with either philosophy.

Where the first-generation RISC machines from different vendors differed from one another only in minor details, later generations have diverged, with the ARM and MIPS taking the more RISC-ish approach, the Power 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, all the major vendors have developed multicore versions of their respective architectures. It seems increasingly likely that future processors will be highly heterogeneous, with multiple implementation strategies—and 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 30 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 like the x86? What will it print on a big-endian machine?

```
unsigned short n = 0x1234;  // 16 bits
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 C-5.2.1 we observed that overflow occurs in two's complement addition when we add two non-negative 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 C-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 (SPARC among them) 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? With the notable exception of ARM, why not multiple 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 $0 \,..\, 2^n$ for unsigned arithmetic, $-2^{n-1} \,..\, 2^{n-1}-1$ for signed arithmetic). Suppose we wish to branch on condition `A op B`, where `A` and `B` are unsigned binary numbers, for `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 C-5.4.1 that if one adds a new instruction to a nonpipelined, 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 last level of on-chip cache, with a penalty of 120 cycles to reach main memory. What is the contribution of last-level cache misses to the average number of cycles per instruction? You may assume that instruction fetches always hit in the L1 cache. Now suppose that we add an off-chip (L3 or L4) cache that can satisfy 90% of the misses from the last-level on-chip cache, at a penalty of only 30 cycles. What is the effect on cycles per instruction?

5.16 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?

5.17 With the development of deeper, more complex pipelines, delayed loads and branches became significantly less appealing as features of a RISC instruction set. In later generations, architects eliminated visible load delays but were unable to do so for branches. Explain.

5.18 Some processors, including the Power series and certain 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?

5.19 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?

5.20 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?

5.21 Consider the code in Figure C-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

5.22 Skip ahead to Sidebar 7.4 (Decimal types) in 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?

5.23 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.

5.24 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?

5.25 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 ARM instruction sets. Discuss, from a compiler writer's point of view, the challenges and opportunities presented by the IA-64.

5.26 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.

5.27 If you have access to computers with more than one kind of processor, 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?

5.28 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?

5.29 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.)

5.30 Translate the high-level pseudocode for vector variance (Example C-5.17) 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.

5.31 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?

5.32 Experiment with small subroutines in C++ to see how much time can be saved by expanding them in-line.

5.9 Bibliographic Notes

The standard reference in computer architecture is the graduate-level text by Hennessy and Patterson [HP12]. More introductory material can be found in the undergraduate computer organization text by the same authors [PH12]. Students without previous assembly language experience may be particularly interested in the text of Bryant and O'Hallaron [BO11], which surveys computer organization from the point of view of the systems programmer, focusing in particular on the correspondence between source-level programs in C and their equivalents in x86 assembler.

The "RISC revolution" of the early 1980s was spearheaded by three separate research groups. The first to start (though last to publish [Rad82]) was the 801 group at IBM's T. J. Watson Research Center, led by John Cocke. IBM's Power and PowerPC architectures, though not direct descendants of the 801, take significant inspiration from it. The second group (and the one that coined the term "RISC") was led by David Patterson [PD80, Pat85] at UC Berkeley. The commercial SPARC architecture is a direct descendant of the Berkeley RISC II design. The third group was led by John Hennessy at Stanford [HJBG81]. The commercial MIPS architecture is a direct descendant of the Stanford design.

Much of the history of pre-1980 processor design can be found in the text by Siewiorek, Bell, and Newell [SBN82]. This classic work contains verbatim reprints of many important original papers. In the context of RISC processor design, Smith and Weiss [SW94] contrast the more "RISCy" and "CISCy" design philosophies in their comparison of implementations of the Power and Alpha architectures. Hennessy and Patterson's architecture text includes an appendix that summarizes the similarities and differences among the major commercial instruction sets [HP12, App. K]. Current manuals for all the popular commercial processors are available from their manufacturers.

An excellent treatment of computer arithmetic can be found in Goldberg's appendix to the Hennessy and Patterson architecture text [Gol12]. Additional coverage of floating point can be found in the same author's 1991 *Computing Surveys*

article [Gol91]. The IEEE 754 floating-point standard was printed in *ACM SIGPLAN Notices* in 1985 [IEE87]. The texts of Muchnick [Muc97] and of Cooper and Torczon [CT04] are excellent sources of information on instruction scheduling, register allocation, subroutine optimization, and other aspects of compiling for modern machines.