Figure 5.29 shows the floating-point addition of 7.875 (1.11111×2^2) and 0.1875 (1.1×2^{-3}). The result is 8.0625 (1.0000001×2^3). After the fraction and exponent bits are extracted and the implicit leading 1 is prepended in steps 1 and 2, the exponents are compared by subtracting the smaller exponent from the larger exponent. The result is the number of bits by which the smaller number is shifted to the right to align the implied binary point (i.e., to make the exponents equal) in step 4. The aligned numbers are added. Because the sum has a mantissa that is greater than or equal to 2.0, the result is normalized by shifting it to the right one bit and incrementing the exponent. In this example, the result is exact, so no rounding is necessary. The result is stored in floating-point notation by removing the implicit leading one of the mantissa and prepending the sign bit.

Floating-point arithmetic is usually done in hardware to make it fast. This hardware, called the *floating-point unit* (*FPU*), is typically distinct from the *central processing unit* (*CPU*). The infamous *floating-point division* (*FDIV*) bug in the Pentium FPU cost Intel $475 million to recall and replace defective chips. The bug occurred simply because a lookup table was not loaded correctly.

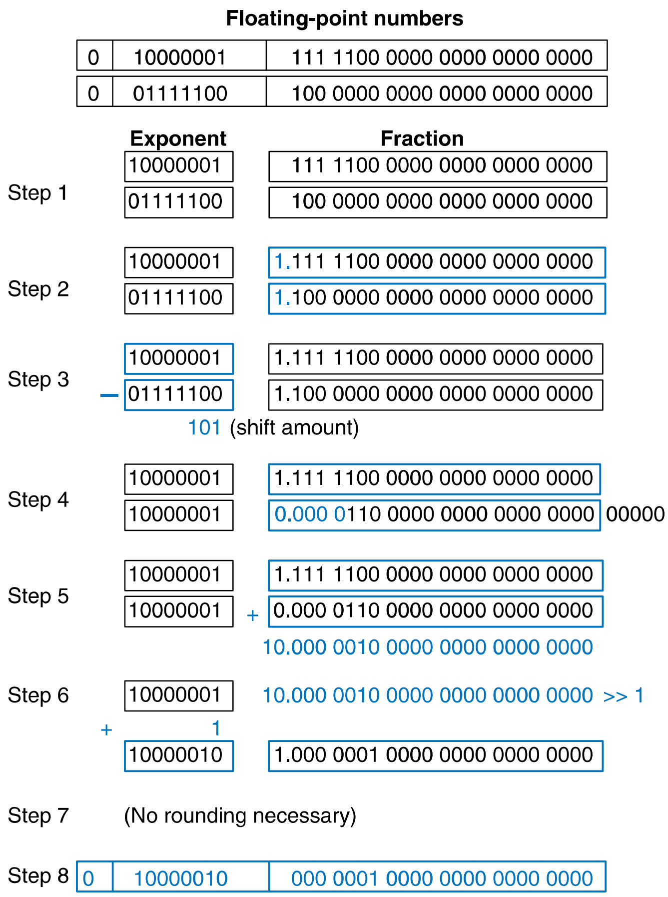

Figure 5.29 Floating-point addition

Table 5.3 Single- and double-precision floating-point formats

Format	Total Bits	Sign Bits	Exponent Bits	Fraction Bits
single	32	1	8	23
double	64	1	11	52

Floating-point cannot represent some numbers exactly, like 1.7. However, when you type 1.7 into your calculator, you see exactly 1.7, not 1.69999. . . . To handle this, some applications, such as calculators and financial software, use *binary coded decimal* (*BCD*) numbers or formats with a base 10 exponent. BCD numbers encode each decimal digit using four bits with a range of 0 to 9. For example the BCD fixed-point notation of 1.7 with four integer bits and four fraction bits would be 0001.0111. Of course, nothing is free. The cost is increased complexity in arithmetic hardware and wasted encodings (A–F encodings are not used), and thus decreased performance. So for compute-intensive applications, floating-point is much faster.

defines 64-bit *double-precision* (also called *double*) numbers that provide greater precision and greater range. Table 5.3 shows the number of bits used for the fields in each format.

Excluding the special cases mentioned earlier, normal single-precision numbers span a range of $\pm 1.175494 \times 10^{-38}$ to $\pm 3.402824 \times 10^{38}$. They have a precision of about seven significant decimal digits (because $2^{-24} \approx 10^{-7}$). Similarly, normal double-precision numbers span a range of $\pm 2.22507385850720 \times 10^{-308}$ to $\pm 1.79769313486232 \times 10^{308}$ and have a precision of about 15 significant decimal digits.

Rounding

Arithmetic results that fall outside of the available precision must round to a neighboring number. The rounding modes are: (1) round down, (2) round up, (3) round toward zero, and (4) round to nearest. The default rounding mode is round to nearest. In the round to nearest mode, if two numbers are equally near, the one with a 0 in the least significant position of the fraction is chosen.

Recall that a number *overflows* when its magnitude is too large to be represented. Likewise, a number *underflows* when it is too tiny to be represented. In round to nearest mode, overflows are rounded up to $\pm\infty$ and underflows are rounded down to 0.

Floating-Point Addition

Addition with floating-point numbers is not as simple as addition with two's complement numbers. The steps for adding floating-point numbers with the same sign are as follows:

1. Extract exponent and fraction bits.
2. Prepend leading 1 to form the mantissa.
3. Compare exponents.
4. Shift smaller mantissa if necessary.
5. Add mantissas.
6. Normalize mantissa and adjust exponent if necessary.
7. Round result.
8. Assemble exponent and fraction back into floating-point number.

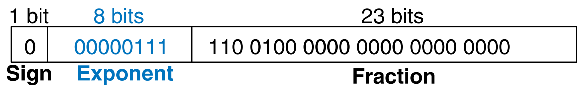

Figure 5.27 Floating-point version 2

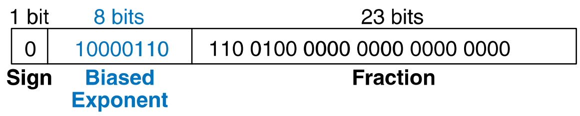

Figure 5.28 IEEE 754 floating-point notation

We make one final modification to the exponent field. The exponent needs to represent both positive and negative exponents. To do so, floating-point uses a *biased* exponent, which is the original exponent plus a constant bias. 32-bit floating-point uses a bias of 127. For example, for the exponent 7, the biased exponent is $7 + 127 = 134 = 10000110_2$. For the exponent -4, the biased exponent is: $-4 + 127 = 123 = 01111011_2$. Figure 5.28 shows $1.11001_2 \times 2^7$ represented in floating-point notation with an implicit leading one and a biased exponent of 134 (7 + 127). This notation conforms to the IEEE 754 floating-point standard.

As may be apparent, there are many reasonable ways to represent floating-point numbers. For many years, computer manufacturers used incompatible floating-point formats. Results from one computer could not directly be interpreted by another computer.

The Institute of Electrical and Electronics Engineers solved this problem by defining the *IEEE 754 floating-point standard* in 1985 defining floating-point numbers. This floating-point format is now almost universally used and is the one discussed in this section.

Special Cases: 0, ±∞, and NaN

The IEEE floating-point standard has special cases to represent numbers such as zero, infinity, and illegal results. For example, representing the number zero is problematic in floating-point notation because of the implicit leading one. Special codes with exponents of all 0's or all 1's are reserved for these special cases. Table 5.2 shows the floating-point representations of 0, ±∞, and NaN. As with sign/magnitude numbers, floating-point has both positive and negative 0. NaN is used for numbers that don't exist, such as $\sqrt{-1}$ or $\log_2(-5)$.

Single- and Double-Precision Formats

So far, we have examined 32-bit floating-point numbers. This format is also called *single-precision, single,* or *float*. The IEEE 754 standard also

Table 5.2 IEEE 754 floating-point notations for 0, ±∞, and NaN

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
−∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

```
     0000.1010   Binary Magnitude
     1111.0101   One's Complement
+            1   Add 1
     1111.0110   Two's Complement
```

Figure 5.23 **Fixed-point two's complement conversion**

```
   0000.1100            0.75
+  1111.0110      +  (–0.625)
  10000.0010            0.125
(a)               (b)
```

Figure 5.24 **Addition: (a) binary fixed-point (b) decimal equivalent**

5.3.2 Floating-Point Number Systems*

$$\pm M \times B^{E}$$

Figure 5.25 **Floating-point numbers**

Floating-point numbers are analogous to scientific notation. They circumvent the limitation of having a constant number of integer and fractional bits, allowing the representation of very large and very small numbers. Like scientific notation, floating-point numbers have a *sign*, *mantissa* (M), *base* (B), and *exponent* (E), as shown in Figure 5.25. For example, the number 4.1×10^3 is the decimal scientific notation for 4100. It has a mantissa of 4.1, a base of 10, and an exponent of 3. The decimal point *floats* to the position right after the most significant digit. Floating-point numbers are base 2 with a binary mantissa. 32 bits are used to represent 1 sign bit, 8 exponent bits, and 23 mantissa bits.

Example 5.5 32-BIT FLOATING-POINT NUMBERS

Show the floating-point representation of the decimal number 228.

Solution: First convert the decimal number into binary: $228_{10} = 11100100_2 = 1.11001_2 \times 2^7$. Figure 5.26 shows the 32-bit encoding, which will be modified later for efficiency. The sign bit is positive (0), the 8 exponent bits give the value 7, and the remaining 23 bits are the mantissa.

---

1 bit	8 bits	23 bits
0	00000111	111 0010 0000 0000 0000 0000
Sign	Exponent	Mantissa

Figure 5.26 **32-bit floating-point version 1**

In binary floating-point, the first bit of the mantissa (to the left of the binary point) is always 1 and therefore need not be stored. It is called the *implicit leading one*. Figure 5.27 shows the modified floating-point representation of $228_{10} = 11100100_2 \times 2^0 = 1.11001_2 \times 2^7$. The implicit leading one is not included in the 23-bit mantissa for efficiency. Only the fraction bits are stored. This frees up an extra bit for useful data.

5.3 NUMBER SYSTEMS

Computers operate on both integers and fractions. So far, we have only considered representing signed or unsigned integers, as introduced in Section 1.4. This section introduces fixed- and floating-point number systems that can also represent rational numbers. Fixed-point numbers are analogous to decimals; some of the bits represent the integer part, and the rest represent the fraction. Floating-point numbers are analogous to scientific notation, with a mantissa and an exponent.

5.3.1 Fixed-Point Number Systems

Fixed-point notation has an implied *binary point* between the integer and fraction bits, analogous to the decimal point between the integer and fraction digits of an ordinary decimal number. For example, Figure 5.21(a) shows a fixed-point number with four integer bits and four fraction bits. Figure 5.21(b) shows the implied binary point in blue, and Figure 5.21(c) shows the equivalent decimal value.

(a) 01101100

(b) 0110.1100

(c) $2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$

Figure 5.21 **Fixed-point notation of 6.75 with four integer bits and four fraction bits**

Signed fixed-point numbers can use either two's complement or sign/magnitude notations. Figure 5.22 shows the fixed-point representation of −2.375 using both notations with four integer and four fraction bits. The implicit binary point is shown in blue for clarity. In sign/magnitude form, the most significant bit is used to indicate the sign. The two's complement representation is formed by inverting the bits of the absolute value and adding a 1 to the least significant (rightmost) bit. In this case, the least significant bit position is in the 2^{-4} column.

(a) 0010.0110

(b) 1010.0110

(c) 1101.1010

Figure 5.22 **Fixed-point representation of −2.375: (a) absolute value, (b) sign and magnitude, (c) two's complement**

Like all binary number representations, fixed-point numbers are just a collection of bits. There is no way of knowing the existence of the binary point except through agreement of those people interpreting the number.

Fixed-point number systems are commonly used for banking and financial applications that require precision but not a large range.

Example 5.4 ARITHMETIC WITH FIXED-POINT NUMBERS

Compute 0.75 + −0.625 using fixed-point numbers.

Solution: First convert 0.625, the magnitude of the second number, to fixed-point binary notation. $0.625 \geq 2^{-1}$, so there is a 1 in the 2^{-1} column, leaving $0.625 - 0.5 = 0.125$. Because $0.125 < 2^{-2}$, there is a 0 in the 2^{-2} column. Because $0.125 \geq 2^{-3}$, there is a 1 in the 2^{-3} column, leaving $0.125 - 0.125 = 0$. Thus, there must be a 0 in the 2^{-4} column. Putting this all together, $0.625_{10} = 0000.1010_2$

Use two's complement representation for signed numbers so that addition works correctly. Figure 5.23 shows the conversion of −0.625 to fixed-point two's complement notation.

Figure 5.24 shows the fixed-point binary addition and the decimal equivalent for comparison. Note that the leading 1 in the binary fixed-point addition of Figure 5.24(a) is discarded from the 8-bit result.
