Appendix L

Number Systems

Binary Numbers

Decimal notation represents numbers as powers of 10, for example

$$1729_{\text{decimal}} = 1 \times 10^3 + 7 \times 10^2 + 2 \times 10^1 + 9 \times 10^0$$

There is no particular reason for the choice of 10, except that several historical number systems were derived from people's counting with their fingers. Other number systems, using a base of 12, 20, or 60, have been used by various cultures throughout human history. However, computers use a number system with base 2 because it is far easier to build electronic components that work with two values, which can be represented by a current being either off or on, than it would be to represent 10 different values of electrical signals. A number written in base 2 is also called a *binary* number.

For example,

$$1101_{\text{binary}} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 1 = 13$$

For digits after the "decimal" point, use negative powers of 2.

$$\begin{aligned} 1.101_{\text{binary}} &= 1\times 2^{0} + 1\times 2^{-1} + 0\times 2^{-2} + 1\times 2^{-3} \\ &= 1 + \frac{1}{2} + \frac{1}{8} \\ &= 1 + 0.5 + 0.125 = 1.625 \end{aligned}$$

In general, to convert a binary number into its decimal equivalent, simply evaluate the powers of 2 corresponding to digits with value 1, and add them up. Table 1 shows the first powers of 2.

Table 1 Powers of Two

Power	Decimal Value
2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1,024
2^{11}	2,048
2^{12}	4,096
2^{13}	8,192
2^{14}	16,384
2^{15}	32,768
2^{16}	65,536

To convert a decimal integer into its binary equivalent, keep dividing the integer by 2, keeping track of the remainders. Stop when the number is 0. Then write the remainders as a binary number, starting with the *last* one. For example,

$$\begin{aligned} 100 \div 2 &= 50 \text{ remainder } 0 \\ 50 \div 2 &= 25 \text{ remainder } 0 \\ 25 \div 2 &= 12 \text{ remainder } 1 \\ 12 \div 2 &= 6 \text{ remainder } 0 \\ 6 \div 2 &= 3 \text{ remainder } 0 \\ 3 \div 2 &= 1 \text{ remainder } 1 \\ 1 \div 2 &= 0 \text{ remainder } 1 \end{aligned}$$

Therefore, $100_{\text{decimal}} = 1100100_{\text{binary}}$.

Conversely, to convert a fractional number less than 1 to its binary format, keep multiplying by 2. If the result is greater than 1, subtract 1. Stop when the number is 0. Then use the digits before the decimal points as the binary digits of the fractional part, starting with the *first* one. For example,

$$\begin{aligned} 0.35 \cdot 2 &= 0.7 \\ 0.7 \cdot 2 &= 1.4 \\ 0.4 \cdot 2 &= 0.8 \\ 0.8 \cdot 2 &= 1.6 \\ 0.6 \cdot 2 &= 1.2 \\ 0.2 \cdot 2 &= 0.4 \end{aligned}$$

Here the pattern repeats. That is, the binary representation of 0.35 is 0.01 0110 0110 0110 . . .

To convert any floating-point number into binary, convert the whole part and the fractional part separately.

Two's Complement Integers

To represent negative integers, there are two common representations, called "signed magnitude" and "two's complement". Signed magnitude notation is simple: use the leftmost bit for the sign (0 = positive, 1 = negative). For example, when using 8-bit numbers,

$$-13 = 10001101_{\text{signed magnitude}}$$

However, building circuitry for adding numbers gets a bit more complicated when one has to take a sign bit into account. The two's complement representation solves this problem. To form the two's complement of a number,

- Flip all bits.
- Then add 1.

For example, to compute –13 as an 8-bit value, first flip all bits of 00001101 to get 11110010. Then add 1:

$$-13 = 11110011_{\text{two's complement}}$$

Now no special circuitry is required for adding two numbers. Simply follow the normal rule for addition, with a carry to the next position if the sum of the digits and the prior carry is 2 or 3. For example,

```
      1 1111 111
+13     0000 1101
-13     1111 0011
-------------------
    1 0000 0000
```

But only the last 8 bits count, so +13 and –13 add up to 0, as they should.

In particular, –1 has two's complement representation 1111 . . . 1111, with all bits set.

The leftmost bit of a two's complement number is 0 if the number is positive or zero, 1 if it is negative.

Two's complement notation with a given number of bits can represent one more negative number than positive numbers. For example, the 8-bit two's complement numbers range from –128 to +127.

This phenomenon is an occasional cause for a programming error. For example, consider the following code:

```
byte b = . . .;
if (b < 0) b = -b;
```

This code does not guarantee that `b` is nonnegative afterwards. If `b` happens to be –128, then computing its negative again yields –128. (Try it out—take 10000000, flip all bits, and add 1.)

IEEE Floating-Point Numbers

The Institute for Electrical and Electronics Engineering (IEEE) defines standards for floating-point representations in the IEEE-754 standard. Figure 1 shows how single-precision (`float`) and double-precision (`double`) values are decomposed into

- A sign bit
- An exponent
- A mantissa

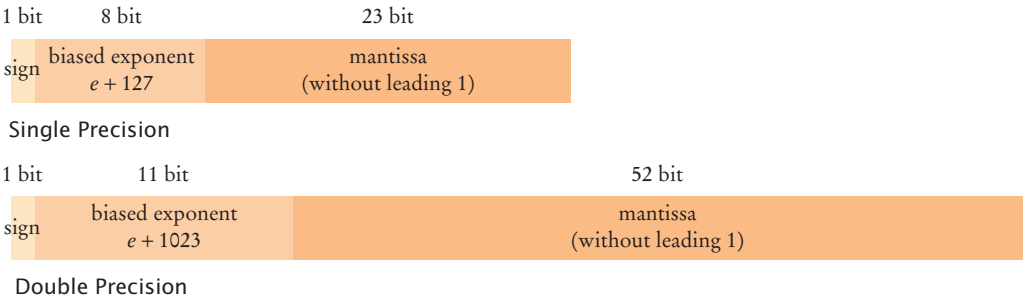

Figure 1 IEEE Floating-Point Representation

Floating-point numbers use scientific notation, in which a number is represented as

$$b_0.b_1b_2b_3\ldots \times 2^e$$

In this representation, e is the exponent, and the digits $b_0.b_1b_2b_3\ldots$ form the mantissa. The *normalized* representation is the one where $b_0 \neq 0$. For example,

$$100_{\text{decimal}} = 1100100_{\text{binary}} = 1.100100_{\text{binary}} \times 2^6$$

Because in the binary number system the first bit of a normalized representation must be 1, it is not actually stored in the mantissa. Therefore, you always need to add it on to represent the actual value. For example, the mantissa 1.100100 is stored as 100100.

The exponent part of the IEEE representation uses neither signed magnitude nor two's complement representation. Instead, a bias is added to the actual exponent. The bias is 127 for single-precision numbers, 1023 for double-precision numbers. For example, the exponent $e = 6$ would be stored as 133 in a single-precision number.

Thus,

$$100_{\text{decimal}} = \text{0 10000110 10010000000000000000000}_{\text{single-precision IEEE}}$$

In addition, there are several special values. Among them are:

- *Zero:* biased exponent = 0, mantissa = 0.
- *Infinity:* biased exponent = 11. . .1, mantissa = 0.
- *NaN* (not a number): biased exponent = 11. . .1, mantissa ≠ 10. . .0.

Hexadecimal Numbers

Because binary numbers can be hard to read for humans, programmers often use the hexadecimal number system, with base 16. The digits are denoted as 0, 1, . . . , 9, A, B, C, D, E, F (see Table 2).

Four binary digits correspond to one hexadecimal digit. That makes it easy to convert between binary and hexadecimal values. For example,

$$11\,|\,1011\,|\,0001_{\text{binary}} = 3B1_{\text{hexadecimal}}$$

In Java, hexadecimal numbers are used for Unicode character values, such as `\u03B1` (the Greek lowercase letter alpha). Hexadecimal integers are denoted with a `0x` prefix, such as `0x3B1`.

Table 2 Hexadecimal Digits

Hexadecimal	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111