*Computer Science 52*

# *Bits, Words, and Integers*

*Spring Semester, 2017*

In this document, we look at how bits are organized into meaningful data. In particular, we will see the details of how integers are represented in a computer.

## *I Bits and Data*

The basic unit of information in a computer is the *bit;* it is simply a quantity that takes one of two values, 0 or 1. A sequence of $k$ bits is a *$k$-bit word.* Words have no intrinsic meaning beyond the value of the bits. We can assign meaning to words by interpreting them in various ways. Depending on the context we impose, a word can be viewed as an integer, as a boolean value, as a floating point number, as a string of letters, or as an instruction to the computer.

We refer to the possible values of a bit in several ways:

- 0 and 1,
- false and true,
- $\bot$ and $\top$ (pronounced“bottom” and “top”),
- low and high, or
- unset and set.

The last two pairs are usually used only when working with circuits, a topic that we will encounter later. You will find yourself using many of these words at various times, depending on the context.

A *byte* is an 8-bit word. Memory in modern computers is arranged as a sequence of bytes, and adjacent bytes are considered to be longer words. As we shall see shortly, bytes can be viewed as the binary representations of integers ranging either between 0 and $2^8 - 1$ or between $-2^7$ and $2^7 - 1$.

## *II Operations on Bits*

We can operate on bits in much the same way that we can add, subtract, and multiply numbers. The system is called *boolean logic,* and here are the common operations.

- $\neg$ means “not” or “(boolean) negation.”
- $\wedge$ means “and.”

- $\vee$ means "(inclusive) or."
- $\Rightarrow$ means "implies" or "not greater than."
- $\oplus$ means "exclusive or" or "unequal." and
- $\equiv$ means "equivalent" or "equal"

The operation $\neg$ is *unary,* meaning that it is applied to a single bit. The other operations are *binary,* and the operator symbol is placed between the two arguments—just like $+$ and $\times$.

The actions of the connectives are specified using *truth tables.* (Remember that we identify 0 with "false" and 1 with "true.") The table for $\neg$ is simple.

| $A$ | $(\neg A)$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

The tables for the other operations are only a little more complicated. In general, the result of the operation corresponds to our understanding of the words we use for the operation: "and," "or," and so on.

| $A$ | $B$ | $(A \wedge B)$ | $(A \vee B)$ | $(A \Rightarrow B)$ | $(A \oplus B)$ | $(A \equiv B)$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 |

We adopt the convention that, in the absence of parentheses, the operation $\neg$ is done first, followed in order by $\wedge$, $\oplus$, $\vee$, $\Rightarrow$, and $\equiv$.

The truth tables for the connectives can be used to evaluate logical expressions. For example, we can verify one of the de Morgan laws,

$$\neg(A \wedge B) = \neg A \vee \neg B,$$

by writing out a truth table for each side of the equality.

| $A$ | $B$ | $(A \wedge B)$ | $\neg(A \vee B)$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

and

| $A$ | $B$ | $(\neg A)$ | $(\neg B)$ | $(\neg A \vee \neg B)$ |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |

The identical results in the last columns show that the two expressions always have the same value. There are many such identities. Table 1 shows some of the common ones.

| | |
|---|---|
| $A \vee 1 = 1$ | $A \wedge 1 = A$ |
| $A \vee 0 = A$ | $A \wedge 0 = 0$ |
| $A \oplus 1 = \neg A$ | $1 \Rightarrow A = A$ |
| $A \oplus 0 = A$ | $0 \Rightarrow A = 1$ |
| | $A \Rightarrow 1 = 1$ |
| | $A \Rightarrow 0 = \neg A$ |
| $A = \neg\neg A$ | $(\neg A \Rightarrow B) \wedge \neg B = A$ |
| $A = A \wedge A$ | $A = A \vee A$ |
| $A \vee \neg A = 1$ | $A \wedge \neg A = 0$ |
| $A \Rightarrow A = 1$ | $A \equiv A = 1$ |
| $A \oplus \neg A = 1$ | $A \oplus A = 0$ |
| $A \vee B = B \vee A$ | $A \wedge B = B \wedge A$ |
| $A \equiv B = B \equiv A$ | $A \Rightarrow B = \neg B \Rightarrow \neg A$ |
| $A \oplus B = B \oplus A$ | |
| $A \vee (B \vee C) = (A \vee B) \vee C$ | $A \wedge (B \wedge C) = (A \wedge B) \wedge C$ |
| $A \equiv (B \equiv C) = (A \equiv B) \equiv C$ | $A \oplus (B \oplus C) = (A \oplus B) \oplus C$ |
| $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$ | $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$ |
| $A \vee (A \wedge B) = A$ | $A \wedge (A \vee B) = A$ |
| $A \equiv B = (A \Rightarrow B) \wedge (B \Rightarrow A)$ | $A \oplus B = \neg(A \equiv B)$ |
| $A \Rightarrow B = \neg A \vee B$ | $A \Rightarrow B = \neg(A \wedge \neg B)$ |
| $A \vee B = \neg(\neg A \wedge \neg B)$ | $A \wedge B = \neg(\neg A \vee \neg B)$ |
| $A \vee B = \neg A \Rightarrow B$ | $A \wedge B = \neg(A \Rightarrow \neg B)$ |

Table 1: Some common identities of boolean logic.

## *III Words as Integers*

Words often represent integers. There are two commonly-used ways to interpret words as integers. Unsigned integers represent non-negative values, while signed integers include negative values.

### *Unsigned Integers*

A $k$-bit word can be interpreted as an *unsigned integer* by viewing the bits as the "digits" in a binary expansion. Normally we take the right-most bit as the least significant, or the "ones place." If the word is $b_{k-1}b_k \ldots b_1 b_0$, then the value of the unsigned word is

$$2^{k-1}b_{k-1} + 2^{k-2}b_{k-2} + \ldots_+ 2b^1 + b_0 = \sum_{i=0}^{k-1} 2^i b_i.$$

Unsigned $k$-bit integers range between 0 and $2^k - 1$, inclusive. The operations on unsigned integers are the arithmetic ones: addition, subtraction, multiplication, division, remainder, etc. There are also the usual comparisons: greater, less, greater or equal, and so forth.

Often, people make no distinction between a word and the unsigned integer it represents. One use of unsigned integers is to specify locations, or addresses, in a computer's memory.

### *Signed Integers*

A $k$-bit word can also be interpreted as a *signed integer* in the range $-2^{k-1}$ through $2^{k-1} - 1$. The words that have a most significant, or leftmost, bit equal to 1 represent the negative values. The leftmost bit is called the *sign bit.* The signed value of the word $b_{k-1}b_k \ldots b_1 b_0$ is

$$-2^{k-1}b_{k-1} + 2^{k-2}b_{k-2} + \ldots_+ 2b^1 + b_0 = -2^{k-1}b_{k-1} + \sum_{i=0}^{k-2} 2^i b_i.$$

The difference between the unsigned value and the signed value is the minus sign on the largest power of two. Notice that the word "signed" does not mean "negative." The signed interpretation gives us positive *and* negative values.

This representation of signed integers is called *two's complement.* Other representations of signed integers exist, but they are rarely used. The programming languages Java and SML use two's complement signed integers exclusively. The programming languages C and C++ have both signed and unsigned integer types.

A word may be interpreted as an unsigned integer or a signed one—the two values may or may not be the same. The table at the right shows

| integer | unsigned | signed |
|---|---|---|
| 7 | 111 | |
| 6 | 110 | |
| 5 | 101 | |
| 4 | 100 | |
| 3 | 011 | 011 |
| 2 | 010 | 010 |
| 1 | 001 | 001 |
| 0 | 000 | 000 |
| −1 | | 111 |
| −2 | | 110 |
| −3 | | 101 |
| −4 | | 100 |

the 3-bit words that represent small integer values. It is easy to scale up the table to include more bits. Notice that the signed value of the word with all 1's is −1.

Interestingly, addition and subtraction can be carried out on unsigned and signed interpretations using the same algorithms. We simply use the binary analogs of the usual methods for adding and subtracting decimal numbers. Other operations, like comparisons and multiplication, require different methods—depending on whether we are thinking of the values as unsigned or signed.

*Negation and Sign Extension*

To negate a two's complement signed value, simply add one to the bitwise compliment of the word. (Verification: Let $x$ be a $k$-bit word and $\overline{x}$ be its bitwise complement. The sum $x + \overline{x}$ is a word with all 1's, whose value as a signed integer is $-1$. Therefore, $x + \overline{x} = -1$, so that $\overline{x} + 1 = -x$.)

We sometimes want to convert a $k$-bit word into a $j$-bit word, with $j$ being greater than $k$. If we are thinking of the words as unsigned integers, we just add zeroes on the left. If we are thinking of them as signed integers, then we copy the sign bit on the left as many times as necessary to create the longer integer. The latter process is called *sign extension,* and it yields a longer (more bits) word representing the same signed integer. For example, −3 is represented in three bits by 101 and in six bits by 111101.

To convince yourself that sign extension actually preserves the signed value, consider the case of adding just one bit. The lefthand part of the expression for the signed value changes

$$\begin{aligned} &\text{from} && -2^{k-1}b_{k-1} + \ldots \\ &\text{to} && -2^{k}b_{k} + 2^{k-1}b_{k-1} + \ldots, \end{aligned}$$

and the bits $b_{k-1}$ and $b_k$ are the same. If those bits are both 0, then the lefthand subexpressions above are both zero. If the bits are both 1, then the subexpressions are both $-2^{k-1}$. Either way, the signed value does not change.

*Words in Modern Computer Systems*

Most current processors use 32- or 64-bit words as their basic unit of data, with the 64 bits becoming increasingly common. In our examples, we use fewer bits to make our examples easier, but the ideas are the same.

Programming languages and systems use different size words for different purposes, and unfortunately, there is considerable variation from one language or system to another. A byte, however, is always 8 bits.

In Java, a `short` is a 16-bit integer quantity, an `int` has 32 bits, and a `long` has 64. All of these are interpreted as signed.

In C, the number of bits in a `short`, `int`, and `long` can vary across different computers, but nowadays, C usually agrees with Java. Any of these can be given signed or unsigned interpretation.

*Addition*

As mentioned earlier, the process of addition is the same for unsigned and signed quantities. The addition

$$\begin{array}{r} 010 \\ +\,001 \\ \hline 011 \end{array}$$

is correct whether we are thinking of unsigned or signed numbers. However, sometimes a result is "out of range," the conditions for which differ depending on how we interpret the numbers. The addition

$$\begin{array}{r} 011 \\ +\,110 \\ \hline 001 \end{array}$$

is a correct instance of binary addition. The "carry out" from the left-most bit is discarded. With *unsigned* integers, the result is erroneous; the result of the sum $3 + 6$ is out of the range of three-bit integers. On the other hand, with *signed* integers, the addition is $3 + (-2)$ and the result is correct. Contrastingly, the result of the addition

$$\begin{array}{r} 011 \\ +\,010 \\ \hline 101 \end{array}$$

is correct for the unsigned interpretation, but in the signed case, it adds two positive values and obtains a negative one. Colloquially, such problematic situations are termed "overflow." Below, we discuss ways to detect it.

*Subtraction*

Subtraction is simply negation followed by addition. One complements the bits in the number to be subtracted and then adds with an extra "carry" into the low order bit. Thus

$$\begin{array}{r} 011 \\ -\,010 \\ \hline 001 \end{array} \quad \text{becomes} \quad \begin{array}{r} {\scriptstyle 1} \\ 011 \\ +\,101 \\ \hline 001 \end{array}$$

The "carry out" from the leftmost column is discarded. This method for subtraction is correct in both the unsigned and signed cases. As in the case of addition, there is the opportunity for erroneous results due to "overflow."

*Overflow and comparisons*

When addition is performed on many computers, four one-bit quantities (besides the usual result) are produced:

$C$ is the the carry-out bit out of the leftmost column

$Z$ is 1 if result is zero and 0 otherwise,

$N$ is the sign bit of result, and

$V$ is 1 if there is "signed overflow" and 0 otherwise.

"Signed overflow" means that two quantities with the same sign produce a sum with the opposite sign.

For subtraction, the bits are set similarly, except that $C$ is the "borrow bit," which is set if the subtraction requires a borrow into the leftmost column. Signed overflow for the subtraction $x - y$ means that $x$ and $y$ have different signs, and the sign of $x - y$ is different from $x$.

The flags allow us to compare integer values. When two quantities are subtracted, the flags will tell us whether the first was less than, equal to, or greater than the second. Any decision about the comparison can be based entirely on the flags; the actual result of the subtraction is not required.

To compare two unsigned integers, one subtracts the quantities, discards the result, and observes the flags. For unsigned interpretations, $x < y$ when there is a "borrow" into the leftmost column upon computing $x - y$. That corresponds to the condition $C = 1$. We see that

$x < y$ if $C = 1$,
$x \leq y$ if $C = 1$ or $Z = 1$,
$x = y$ if $Z = 1$,
$x \neq y$ if $Z = 0$,
$x \geq y$ if $C = 0$, and
$x > y$ if $C = 0$ and $Z = 0$.

For signed integers, we compute $x - y$ in a similar fashion and use a different interpretation of the values of the flags:

$x < y$ if $N \oplus V = 1$,
$x \leq y$ if $Z = 1$ or $N \oplus V = 1$,
$x = y$ if $Z = 1$,
$x \neq y$ if $Z = 0$,
$x \geq y$ if $N \oplus V = 0$, and
$x > y$ if $Z = 0$ or $N \oplus V = 0$.

To see why these conditions are correct, consider the example of $x < y$. The result $N \oplus V = 1$ means $N \neq V$, which is to say that the result $x - y$ is negative and correct, or it is non-negative and incorrect. Either way, $x < y$.

## *IV Hexadecimal Notation*

A $k$-bit word can be written as a sequence of $k$ zeroes and ones, but that representation is long and awkward. An alternative is to take the unsigned value of a word and write it in *hexadecimal*, base 16. It is convenient because a group of four bits corresponds to one hexadecimal "digit."

The digits are the usual decimal digits `0` through `9` followed by the letters `A` through `F`. To convert a multibyte word to hexadecimal, simply divide the bits into groups of four and write their values as hexadecimal digits. For example,

0011 1011 1110 0100

is `3BE4` in hexadecimal. Hexadecimal representations are often prefixed by `0x` to distinguish, for example, `0x15` from the decimal value 15.

When working with hexadecimal notation, avoid the temptation to convert to decimal, do a calculation, and convert back to hexadecimal. It takes too long, and there is a tremendous chance for error. Instead, consider doing the calculation by hand in hexadecimal. With a little practice, you will find it easy for addition or subtraction. Or, you can find a hexadecimal calculator; they are common on the internet.

## *V Other Operations on Words*

In addition to the arithmetic operations, we have shifts and bitwise logical operations on words. Most programming languages provide these operations.

A *left shift* shifts the bits to the left. The most significant bits are lost, and zeroes are shifted in on the right. For integers, a left shift is a

handy way to multiply by a power of two. It does not matter if we are imposing an unsigned or a signed representation. Java and C++ use the notation `x<<k` to shift the bits in `x` to the left by `k` places, effectively multiplying by $2^k$.

A *logical right shift* moves the bits to the right. The least significant bits are lost, and zeroes are shifted in on the left. For *unsigned* integers, the logical right shift is the equivalent of dividing by a power of two and discarding the fractional part.

An *arithmetic right shift* also shifts bits to the right, but the new bits on the left are copies of the sign bit. An arithmetic right shift preserves the sign of the original word, and it is a handy way to divide a *signed* integer by a power of two.

Java uses the notation `x>>>k` for the logical right shift of `x` by `k` places and `x>>k` for the arithmetic right shift. The programming language C has only one right shift operator, `>>`. Most C compilers choose the arithmetic right shift when the first operand is a signed integer and the logical right shift when the first operand is unsigned.

As we mentioned, a sequence of $k$ bits is a *$k$-bit word.* The operations of propositional logic may be applied to words, in which case they operate on the bits in parallel. For example, the negation of the three-bit word 011 is 100, and the result of an and-operation on 011 and 100 is 000. The study of these operations is called "digital logic."

Many programming languages use the ampersand `&` to refer to the the bitwise and-operation, and `|` for the bitwise or-operation. Often `^` is used for the bitwise xor-operation.

Do not confuse the bitwise logical operators `&` and `|` with the boolean operators `&&` and `||`.

There are many times when bitwise operations are handy. For example, two $k$-bit words `w` and `x` are equal if

```
w ^ x
```

yields a word with all zeroes.

Remember that the low order bit of an odd integer is 1, so the integer represented by the word `w` is odd exactly when

```
w & 1
```

is not zero. Similarly, the high order bit of a word representing a signed integer is 1 when the integer is negative. We can test for a negative value in a $k$-bit word `w` by seeing if

```
w & (1<<(k-1))
```

is non-zero.

## *VI Other Kinds of Data*

So far, we have seen numbers: signed and unsigned integers and floating point numbers. The common non-numeric data types are characters, strings, boolean values, and pointers.

Characters are represented as integers, with a mapping between the letters and the numbers that represent them. One common encoding is ASCII, which represents characters with a single byte. In that system, the byte 0011 0100 is the character `4` (not to be confused with the *integer* 4), and 0110 1101 is the lower-case letter `m`. The specifics of the correspondence are usually not important.

In recent years, people have recognized the limitations of the small size of the ASCII character set. An alternative is the Unicode encoding, which uses sixteen bits and has a much richer collection of letters. Java uses the Unicode system, and its primitive data type `char` is a sixteen-bit unsigned integer.

The boolean values, false and true, can be represented by single bits, but on most computers it is inconvenient to work with a data object smaller than a byte. Most frequently, boolean values are full words, the same size as integers. Usually the word with all 0's corresponds to false and any non-zero word corresponds to true. Other words may or may not correspond to boolean values. The operators `&&` and `||` are commonly used to refer to the logical operations on boolean values represented in this way. (In C, there is no specific boolean type. That language uses the type `int`, interpreting zero as false and any non-zero value as true. C++ recognizes the same convention, but recently the `bool` type was added to the language, and programmers are encouraged to use it.)

As we shall see in another part of the course, the memory in a computer is simply an array of bytes. Pointers are usually (but not always) interpreted as indices into that array. That is, a location in memory by counting the number of bytes from the beginning, and a pointer is an unsigned integer which signifies a location. It is a matter of some debate whether or not a programmer should make extensive use of this representation.

Arrays are formed by placing the elements next to one another in memory. Strings can be viewed as arrays of characters, and in many programming languages they are exactly that. In other languages, strings are have a more sophisticated representation.

Keep in mind that words are simply sequences of bits. They have no meaning until we impose a representation on them. A word has a

value when interpreted as a signed integer and a completely different meaning when interpreted as, say, a string.

### *Appendix: Floating-Point Numbers*

Many numbers that we encounter in our computations are not integers; they have fractional parts. For completeness, we describe the representation that most computers use for numbers that are not integers.

We introduce a "binary point" that is used analogously to a decimal point. The expression $-10011.11001_2$ is the binary representation for $-19.78125_{10}$. The positions to the right of the binary point represent $1/2$, $1/4$, $1/8$, and so on, so that the fractional part .11001 corresponds to $1/2 + 1/4 + 1/32$. We can write the number as

$$-1.001111001 \times 2^4.$$

A binary purist would have written the exponential part as $10_2{}^{100_2}$, but that seems excessively cumbersome.

These numbers are called *floating-point* because the binary point "floats" according to the exponent. Unless the number under consideration is zero, we normalize the presentation so that there is a single 1 to the left of the binary point.

#### *Single and Double Precision*

A number in binary scientific notation is determined by its sign, fraction, and exponent. Most computers now use IEEE Standard 754, which specifies a format for storing these three quantities. There are two variants of the standard: single precision and double precision. They correspond to `float` and `double`, respectively, in Java.

The *single precision* format uses four bytes, or 32 bits: one bit for the sign, eight bits for the exponent, and 23 bits for the fractional part. The sign bit is 1 if the number is negative and 0 otherwise. The sign bit is followed by the eight bits for the exponent, and then the fractional part is placed in the remaining bits, padded with 0's on the right if necessary.

The exponent is a signed value, but it is not represented as a two's complement integer. Instead it uses the *excess* 127 representation. The value of the exponent is the unsigned value of the eight bits minus 127. In this representation, the exponent can take on values from $-127$ through 128.

The number in the example above would have the single precision representation

1 10000011 00111100100000000000000.

The actual unsigned value of the eight exponent bits is $132_{10}$; subtracting 127 gives the actual exponent of 4. Note that only the fraction proper, and not the 1 to the left of the binary point, appears. That allows us to squeeze one extra bit of precision into the 32-bit single precision number.

The *double precision* format is similar, except that it uses eight bytes. The sign still takes only one bit. The exponent occupies 11 bits using excess 1023 representation, and the remaining 52 bits are devoted to the fraction. With a larger possible exponent and more bits in the fraction, double precision numbers can have larger values and more "significant figures" than single precision ones. Table 2 compares the two formats.

Notice in subsection 2 that the exponent range for single precision is from $-126$ to 127, not $-127$ to 128 as might be expected. The two extreme exponents are reserved for special purposes. Table 3 shows the five forms of floating-point numbers: normalized, denormalized, zero, infinity, and not a number.

The *normalized* form is the one that we have been describing so far. Let $B$ be the "excess" amount in the representation of the exponent; it is sometimes called the *bias.* In the case of single precision, $B = 127$. The floating-point number

| sign | exponent | fraction |
|---|---|---|

is in the normalized form if the exponent does not consist of all 0's or all 1's. It has the value

$$\pm 1.\text{fraction} \times 2^{E-B},$$

where $E$ is the unsigned value of the exponent bits. As always, the value is negative if the sign bit is 1.

| | Single Precision | Double Precision |
|---|---|---|
| Sign bits | 1 | 1 |
| Exponent bits | 8 | 11 |
| Fraction bits | 23 | 52 |
| Exponent system | excess 127 | excess 1023 |
| Exponent range | $-126$ to 127 | $-1022$ to 1023 |
| Largest normalized | about $2^{128}$ | about $2^{1024}$ |
| Smallest normalized | $2^{-126}$ | $2^{-1022}$ |
| Smallest denormalized | about $10^{-45}$ | about $10^{-324}$ |
| Decimal range | about $10^{-38}$ to $10^{38}$ | about $10^{-308}$ to $10^{308}$ |

Table 2: A summary of the two types of IEEE floating-point numbers. (Adapted from Tanenbaum, Structured Computer Organization, Prentice-Hall, third edition, 1990.)

| | Sign | Exponent | Fraction |
|---|---|---|---|
| Normalized | ± | $0 < \text{exp} < \text{max}$ | any |
| Zero | ± | 0 | zero |
| Denormalized | ± | 0 | non-zero |
| Infinity | ± | max | zero |
| Not a number | ± | max | non-zero |

Table 3: The five forms for floating-point numbers. The values in the exponent column are the unsigned values, ranging from 0 to a maximum of $11\ldots1_2$. (Also adapted from Tanenbaum.)

The *zero* form is the representation for the number zero. All bits, except perhaps the sign bit, are zero. There are two representations of zero, one "positive" and the other "negative." A computer must be designed to recognize that they are the same.

The *denormalized* form provides a way to represent much smaller positive numbers than would otherwise be possible. The sign bit and the exponent, which consists of all 0's, are interpreted as in the normalized case. The difference is that, with denormalized numbers, the bit to the left of the binary point is taken to be 0 instead of 1. The smallest positive value that can be represented in single precision has the denormalized representation

0 00000000 00000000000000000000001,

which translates to

$$+0.00000000000000000000001 \times 2^{0-127}.$$

There are 23 fraction bits, so the number is $2^{-150}$, quite a bit less than the smallest normalized positive number of $2^{-126}$.

The two remaining forms, *infinity* and *not a number*, arise as the result of overflow or other errors. The IEEE standard specifies the results of operating on these as well as the more conventional forms.

### *Numerical Computations*

Arithmetic on floating-point numbers is analogous to arithmetic on numbers expressed in decimal scientific notation. For addition, one must shift the binary point of one term so that the exponents are the same before adding. This requires the disassembly of the sign, exponent, and fraction part before the operation and subsequent reassembly. Usually, the calculation is carried out by specialized hardware. Other operations on real numbers, like the calculation of square roots and trigonometric functions, are done using approximation techniques.

The finite range and precision of floating-point numbers can cause errors that would not arise if one were working with perfectly exact

mathematical objects. An entire branch of mathematics, numerical analysis, is devoted to studying computation with these approximations that we call floating-point numbers. In this section, we are content to indicate a few of the most common considerations.

Although we sometimes talk about floating-point "reals," many of the usual mathematical laws for real numbers fail when applied to floating-point numbers. For example, the associative law $(x+y)+z = x+(y+z)$ does not hold for floating-point; can you find a counterexample?

The order of operations makes a difference. The expression

$$-1 + \underbrace{2^{-26} + 2^{-26} + \ldots + 2^{-26}}_{2^{26} \text{ terms}}$$

is zero mathematically, but the result will be $-1$ if it is computed from left to right with single precision numbers. Individually, the terms on the right are too small to change the $-1$. It does not help much to start at the right, either; the result is $-0.75$ in that case.

Even in the best of cases, floating-point results will be approximations. This is because of small errors that creep in when input values are converted from decimal to binary and round-off errors that can occur with every calculation. A result of this observation is good advice: *Never test floating-point numbers for equality!* Two numbers may be mathematically equal, but as results of floating-point computations, the bit patterns will be slightly different. Instead of asking whether $x$ and $y$ are equal, always ask whether $|x - y|$ is less than some (small) positive tolerance.