mat can be done with an integer adder, while multiplication requires some extra shifting. With this representation, although the binary point is not represented physically, the position of the binary point, established by design, cannot be changed. To transform all the numbers into this format, a series of scaling and shifting operations must be performed, attaching scale factors to numbers. These factors must be handled by program, which increases the computing time. Other representations that have been proposed involve storing the logarithm of a number and performing multiplication by adding the logarithms, or using a pair of integers (x, y) to represent the fraction x/y.

A better solution is to use an automatic scaling technique, known as the floating-point representation (also referred to as *scientific notation*). In this case, the scaling factor becomes part of the computer word, and the position of the binary point changes automatically for each number.

3.4.1. Floating-Point Representation

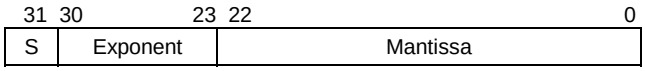
3.4.1.1. Principles

In general, a floating-point (FP) number N can be represented in the following form:

$$N = \pm M * B^{\pm E} \tag{3.27}$$

A floating-point number has two components. The first component is the *mantissa* (M), which represents the exact value of the number in a specific range, and is normally stored as a signed binary fraction. The second component is the *exponent* (E), which indicates the number's order of magnitude. In the above expression, B represents the base (radix) of the exponent.

This representation can be stored in a binary word with three fields: sign, mantissa, and exponent. For example, assuming that the word has 32 bits, a possible assignment of bits to each field could be the following:



This is a *sign-magnitude* representation, since the sign has a separate bit from the rest of the number. The *sign* field consists of one bit and indicates the sign of the number, 0 for positive and 1 for negative. No field is assigned to the exponent base B. This is because the base B is the same for all numbers, and often it is assumed to be 2. Therefore, there is no need to store the base.

Usually, the *exponent* field does not contain the true exponent, but the *characteristic* (C), obtained by adding a fixed value, called *bias*, to the true exponent, so that the resulting value is always positive:

$$C = E + bias \tag{3.28}$$

In this way, there is no need to reserve a separate field for the sign of the exponent. This representation is known as a *biased* representation. The true exponent can be determined by subtracting the bias from the content of the exponent field. In the previous example, the exponent field consists of 8 bits, which can represent numbers 0 to 255. Assuming the bias value to be 128 (80h), the true exponents are in the range from –128 to +127, being negative if $C < 128$, positive if $C > 128$, and zero if $C = 128$. Thus, the exponent is represented *in excess* 128.

One of the advantages of the biased representation is that the operations performed with the exponent are simplified, since there are no negative exponents. The second advantage refers to the representation of zero. The mantissa of number zero has bits of 0 in all positions. Theoretically, the exponent of number zero could have any value, the result being zero. In some computers, if the mantissa of the result is zero, the value of the exponent is not changed, resulting an *impure zero*.

For the majority of computers, it is recommended for the number zero to have an exponent with the minimum value, resulting a *pure zero*. For the biased representation, the minimum value of the exponent is 0. In this way, by using a biased representation for the exponent, the floating-point representation of number zero is the same with the fixed-point representation, all the bits being zero. This means that the same circuits can be used to check if a value is zero.

Another advantage of using biased exponents is that floating-point nonnegative numbers are ordered in the same way as integers. That is, the magnitude of floating-point numbers can be compared using an integer comparator.

A drawback of using biased exponents is that adding them is slightly complicated, because it requires that the bias be subtracted from their sum.

In the previous example, the *mantissa* consists of 23 bits. Although the binary point is not represented, it is assumed to be at the left side of the most significant bit of the mantissa. For example, if $B = 2$, the floating-point number 1.75 can be represented in several forms:

$+0.111 * 2^1$	0	1000 0001	1110 0000 0000 0000 0000 000
$+0.00111 * 2^3$	0	1000 0011	0011 1000 0000 0000 0000 000

To simplify the operations with floating-point numbers and increase their precision, these numbers are always represented in normalized form. A floating-point number is said to be *normalized* if the leftmost bit (the most significant bit) of the mantissa is 1. Therefore, in the two representations presented for 1.75, the first representation, which is normalized, is used.

Since the leftmost bit of the mantissa of a normalized floating-point number is always 1, this bit is often not stored and is assumed to be a *hidden bit* to the right of the binary point. This allows the mantissa to have one more significant bit. Thus the 23-bit field is used to store a 24-bit mantissa with a value between 0.5 and 1.0.

With this representation, Figure 3.27 indicates the range of numbers that can be represented in a 32-bit word.

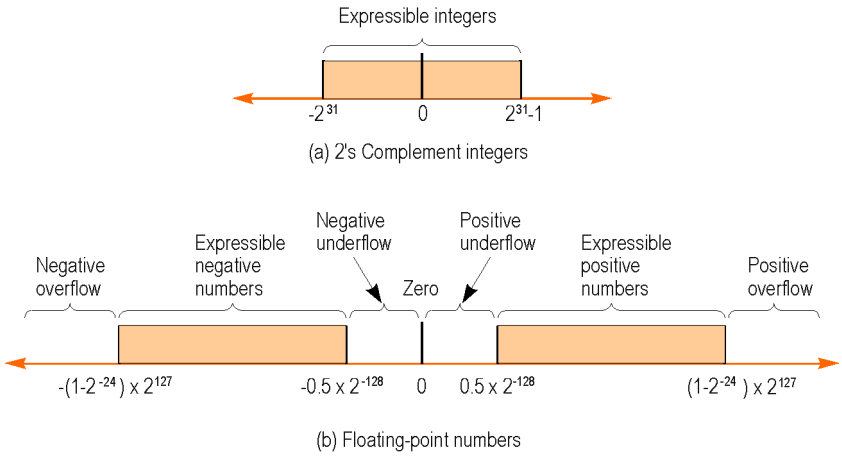


Figure 3.27. Expressible numbers in typical 32-bit formats.

With a 32-bit word, using 2's complement integer representation, all of the integers from -2^{31} to $2^{31} - 1$ can be represented, for a total of 2^{32} different numbers. With the floating-point format presented, the following ranges of numbers are possible (Figure 3.27):

- Negative numbers between $-(1 - 2^{-24}) * 2^{127}$ and $-0.5 * 2^{-128}$;
- Positive numbers between $0.5 * 2^{-128}$ and $(1 - 2^{-24}) * 2^{127}$.

Five regions on the number line are not included in these ranges:

- Negative numbers less than $-(1 - 2^{-24}) * 2^{127}$, called *negative overflow*;
- Negative numbers greater than $-0.5 * 2^{-128}$, called *negative underflow*;
- Zero;
- Positive numbers less than $0.5 * 2^{-128}$, called *positive underflow*;
- Positive numbers greater than $(1 - 2^{-24}) * 2^{127}$, called *positive overflow*.

Sometimes, the hidden bit is assumed to be to the left of the binary point. That is, the stored mantissa M will actually represent the value $1.M$. In this case, the normalized 1.75 will have the following form:

$+1.11 * 2^0$

0	1000 0000	1100 0000 0000 0000 0000 000

Assuming a hidden bit to the left of the binary point in the floating-point format presented, a non-zero normalized number represents the following value:

$$(-1)^S * (1.M) * 2^{E-128} \tag{3.29}$$

where S denotes the sign bit.

This format can represent the following range of numbers:

- Negative numbers between $-[1 + (1 - 2^{-23})] * 2^{127}$ and $-1.0 * 2^{-128}$;
- Positive numbers between $1.0 * 2^{-128}$ and $[1 + (1 - 2^{-23})] * 2^{127}$.

The problem with the example floating-point format is that there is no representation for the value zero. This is because a zero cannot be normalized since it does not contain a non-zero digit. However, in practice the floating-point representations reserve a special bit pattern to designate zero. Often a zero is represented by all 0's in the mantissa and exponent. A good example of such bit pattern assignment for 0 is the standard format defined by the *IEEE Computer Society* (the IEEE 754 standard, Section 3.4.1.2).

Overflow occurs when the result is larger than the allowable representation, for example, when the exponent is too large to be represented in the exponent field. *Underflow* occurs when the result is smaller than the allowable representation, for example, when the negative exponent is too large to fit in the exponent field. Underflow is a less serious problem because the result can generally be approximated to 0. Processors have certain mechanisms for detecting, handling, and signaling overflow and underflow.

To design a floating-point format, it is important to find a compromise between the size of the mantissa and the size of the exponent. Increasing the size of the mantissa enhances the precision of the numbers, and increasing the size of the exponent increases the range of numbers that can be represented. The only way to increase both range and precision is to use more bits. Thus, most computers offer, at least, *single-precision* numbers and *double-precision* numbers. For example, a single-precision format might have a size of 32 bits, and a double-precision format 64 bits.

3.4.1.2. IEEE 754 Floating-Point Standard

In the past, the execution of floating-point operations varied considerably from one computer family to another. The variations involved the number of bits allocated to the exponent and mantissa, the range of exponents, the rounding mode, and the operations performed on exceptional conditions such as overflow and underflow. To facilitate the portability of programs from one computer to another and to encourage the development of sophisticated numerically-oriented programs, the *IEEE Computer Society* has developed the IEEE 754 standard for floating-point representation and arithmetic. This standard was released in 1985.

The focus of the IEEE standard is the microprocessor environment, where individual manufacturers may provide only limited numerical capability. As a result of this standard, vendors have developed chips that implement the standard and can be incorporated into microcomputer systems. For example, most of the floating-point units and math coprocessors, including those of the *Intel* processors', conform to this standard.

The IEEE 754 standard defines the following formats or precisions: *single*, *single extended*, *double*, and *double extended*. The main parameters of these formats are presented in Table 3.9. The standard does not require to implement all the formats, but

recommends to support either the combination of single and single extended formats, or the single, double, and double extended formats.

Table 3.9. Format parameters specified by the IEEE 754 floating-point standard.

	Single	Single extended	Double	Double extended
Bits of mantissa	24	≥ 32	53	≥ 64
Maximum real exponent	127	≥ 1023	1023	≥ 16383
Minimum real exponent	-126	≤ -1022	-1022	≤ -16382
Exponent bias	127	Not specified	1023	Not specified

In all the formats, the implied exponent base is assumed to be 2. The single-precision, double-precision and double-precision extended formats are presented in Figure 3.28. These formats are often implemented in the math coprocessors and floating-point units of processors.

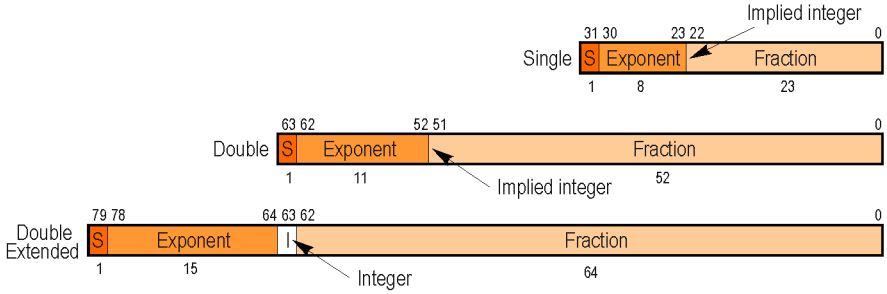


Figure 3.28. The single, double, and double extended formats defined by the IEEE 754 floating-point standard.

S represents the sign of the number. The biased exponent is represented on 8 bits for single-precision, on 11 bits for double-precision, and on 15 bits for double extended-precision. The exponent biases for the three formats are 127 (3Fh), 1023 (3FFh), and 16,383 (3FFFh), respectively. The minimum values (0) and the maximum values of the exponent (255, 2047, and 32,767) are not used for normalized numbers, being used to represent special values.

A hidden bit is also used in the IEEE 754 standard, but the mantissa is represented differently. The representation of the mantissa is called *significand* in the IEEE standard. For the single-precision and double-precision formats, the mantissa consists of an implicit bit of 1 (the integer part), the implicit binary point, and the bits of the fraction *F*:

$$M = 1.F$$

If all the fraction bits are 0, the mantissa is 1.0; if all the fraction bits are 1, the mantissa is nearly 2.0. Thus:

$$1.0 \le M < 2.0$$

The double extended format is used by the floating-points units of the processors and by the math coprocessors, in order to reduce the rounding errors. In this format, bit 63 represents the integer part of the mantissa, which is not implicit. The numbers in this format are not always normalized, therefore they do not always begin with a bit of 1. For this reason, this bit is represented explicitly, and is denoted by *I* in Figure 3.28. In this case, the value of the mantissa is:

$$M = I.F$$

The value of a number in single-precision (N_S), in double-precision (N_D), and in double extended-precision (N_E) is:

$$N_S = (-1)^s \cdot M \cdot 2^{E-127} \tag{3.30}$$

$$N_D = (-1)^s \cdot M \cdot 2^{E-1023} \tag{3.31}$$

$$N_E = (-1)^s \cdot M \cdot 2^{E-16383} \tag{3.32}$$

The range of numbers which can be represented in the single-precision format is approximately 1.18×10^{-38} to 3.4×10^{38}, in the double-precision format is 2.23×10^{-308} to 1.79×10^{308}, and in the double-precision extended format is 3.37×10^{-4932} to 1.18×10^{4932}.

Example 3.7

Show the IEEE 754 binary representation of the number –0.75 in single precision.

Answer

The number – 0.75 is also – 3/4 or – 0.11 in binary. In scientific notation the value is -0.11×2^0, and in normalized scientific notation it is -1.1×2^{-1}. The exponent field will contain – 1 + 127 = 126 (7Eh). The single precision representation of -0.75 is then:

31	30 … 23	22 … 0
1	0111 1110	1000 0000 0000 0000 0000 000

Example 3.8

What decimal number is represented by the following word?

31	30 … 23	22 … 0
1	1000 0001	0100 0000 0000 0000 0000 000

Answer

The sign bit is 1, the exponent field contains 81h = 129, and the significand field contains $1 \times 2^{-2} = 0.25$. The value of the number is:

$$(-1)^1 \times 1.25 \times 2^{(129-127)} = -1.25 \times 2^2 = -1.25 \times 4 = -5.0$$

One of the problems that could appear in floating-point calculation refers to handling underflows and overflows. Another problem is the representation of undefined values. Besides the normalized numbers, the IEEE standard specifies the representation of some special values. The special values for the single-precision and the double-precision formats are presented in Table 3.10. This table also indicates the values of normalized numbers in these two formats.

Table 3.10. Values of IEEE floating-point numbers.

Single-Precision (32 bits)			Double-Precision (64 bits)		
Exponent	**Significand**	**Value**	**Exponent**	**Significand**	**Value**
0	0	$(-1)^S\ 0$	0	0	$(-1)^S\ 0$
0	≠0	$(-1)^S\ 2^{E-126}\ (0.F)$	0	≠0	$(-1)^S\ 2^{E-1022}\ (0.F)$
1…254	any value	$(-1)^S\ 2^{E-127}\ (1.F)$	1…2046	any value	$(-1)^S\ 2^{E-1023}\ (1.F)$
255	0	$(-1)^S\ \infty$	2047	0	$(-1)^S\ \infty$
255	≠0	NaN	2047	≠0	NaN

The value *zero* is represented as all 0's in the significand and exponent. There are two representations for value 0, depending on the sign bit: + 0 or – 0. The hidden bit on the left of the binary point is 0 instead of 1.

When the result of one operation is smaller than the minimum possible normalized number, normally the result is set to zero and the computations continue, or an underflow condition is signaled. None of these solutions is acceptable. For this reason, rather than having a gap between 0 and the smallest normalized number, the IEEE 754 standard allows some numbers to be represented in unnormalized form. These are called *denormal* numbers, also called *subnormal* numbers. They have a zero exponent, but a non-zero significand. In this case, the hidden bit is 0.

A denormal number is computed through a technique called *gradual underflow*. Table 3.11 gives an example of gradual underflow in the denormalization process. Here the single-precision format is used, so that the minimum unbiased exponent is -126. The true result in this example requires an exponent of –129 in order to obtain a normalized number. Since –129 is beyond the allowable exponent range, the result is denormalized by inserting leading zeros in the mantissa and incrementing the exponent until the minimum allowed exponent of –126 is reached.

Table 3.11. Example of denormalization process using the gradual underflow technique.

Operation	Sign	Exponent	Significand
True Result	0	–129	1.01011100000…00
Denormalize	0	–128	0.10101110000…00
Denormalize	0	–127	0.01010111000…00
Denormalize	0	–126	0.00101011100…00
Denormal result	0	–126	0.00101011100…00

In the extreme case, all the significant bits are shifted out to the right by leading zeros, creating a zero result.

In the case of overflow, there is a special representation for *infinity*, consisting of the maximum value of the exponent for the format used, and a zero significand. Depending on the sign bit, $+\infty$ and $-\infty$ are possible. The value of infinity can be used as an operand, using rules such as:

$$\infty + n = \infty$$
$$n / \infty = 0$$
$$n / 0 = \infty$$

In this way, the user can decide to interpret the overflow as an error condition, or to continue the computations with the value of infinity.

To signal various exception conditions, as in the case of undefined operations such as ∞/∞, $\infty-\infty$, $\infty * 0$, $0/\infty$, $0/0$, or taking the square root of a negative number, a special format is provided, which does not represent an ordinary number, and is called *Not a Number* (NaN). The exponent has the maximum possible value, and the significand is non-zero. Thus, there is an entire family of NaNs.

The IEEE standard specifies that when an argument of an operation is a NaN, the result should be a NaN. Because of the rules for performing arithmetic with NaNs, writing floating-point subroutines that can accept NaN as an argument does not require special case checks. For example, suppose that *arccos* is computed in terms of *arctan*, using the formula $arccos(x) = 2arctan\left(\sqrt{(1-x)/(1+x)}\right)$. If *arctan* handles an argument of NaN properly, *arccos* will do so too. If x is a NaN, $1+x$, $1-x$, $(1+x)(1-x)$ and $\sqrt{(1-x)/(1+x)}$ will also be NaNs. No checking for NaNs is required.

Another feature of the IEEE standard with implications for hardware is the rounding rule. When operating on two floating-point numbers, usually the result cannot be exactly represented as another floating-point number. The standard specifies four rounding modes: round toward 0, round toward $+\infty$, round toward $-\infty$, and round to nearest. The last rounding mode is the default, and it is provided for the situations when the actual number is exactly halfway between two floating-point representations. This mode rounds to an even number.

The IEEE standard defines five exceptions: underflow, overflow, divide by zero, inexact result, and invalid operation. By default, when these exceptions occur,

they set a flag and the computation continues. The standard recommends for implementations to provide an enable bit for each exception. When an exception with an enabled bit occurs, a software exception handler is called.

The underflow, overflow and divide-by-zero exceptions are found in most floating-point systems. The *inexact exception* occurs when the result of an operation must be rounded. This is not really an exceptional condition, because occurs frequently. Thus, enabling a software exception handler for inexact results could have a severe impact on performance. The *invalid exception* occurs for invalid operations, such as $0/0$, $\infty - \infty$ or $\sqrt{-1}$.

The main advantage of the IEEE standard is that helps to write portable software libraries that deal with floating-point exceptions. The standard also has some drawbacks:

1. It was originally intended for microprocessors, so the requirements of high-performance implementations were not given high priority.
2. The standard contains optional parts. For implementors, it is difficult to decide which part should they implement. For portable software writers, the question is whether they should avoid using any of the optional part of the standard.
3. Gradual underflow has usually been implemented in a way that is orders of magnitude slower than setting the result to zero, so users often disable it.
4. The standard does not describe the integer arithmetic and the transcendental functions (*sin*, *cos*, *exp*). In particular, it does not specify the accuracy that the transcendental functions should have, or the exceptional values of transcendental functions, such as 0^0.

3.4.2. Floating-Point Operations

Consider two floating-point numbers:

$$X = M_X B^{E_X} \tag{3.33}$$

$$Y = M_Y B^{E_Y} \tag{3.34}$$

Table 3.12. Floating-point arithmetic operations.

Operation	Result
$X + Y$	$(M_X B^{E_X - E_Y} + M_Y) \times B^{E_Y}, E_X \le E_Y$
$X - Y$	$(M_X B^{E_X - E_Y} - M_Y) \times B^{E_Y}, E_X \le E_Y$
$X \times Y$	$(M_X \times M_Y) \times B^{E_X + E_Y}$
$X \div Y$	$(M_X \div M_Y) \times B^{E_X - E_Y}$