Jean-Michel Muller
Nicolas Brunie
Florent de Dinechin
Claude-Pierre Jeannerod
Mioara Joldes
Vincent Lefèvre
Guillaume Melquiond
Nathalie Revol
Serge Torres

Handbook of Floating-Point Arithmetic

Second Edition

Birkhäuser

Chapter 3

Floating-Point Formats and Environment

OUR MAIN FOCUS IN THIS CHAPTER is the IEEE[1] 754-2008 Standard for Floating-Point Arithmetic [267], a revision and merge of the earlier IEEE 754-1985 [12] and IEEE 854-1987 [13] standards. A paper written in 1981 by Kahan, *Why Do We Need a Floating-Point Standard?* [315], depicts the rather messy situation of floating-point arithmetic before the 1980s. Anybody who takes the view that the current standard is too constraining and that circuit and system manufacturers could build much more efficient machines without it should read that paper and think about it. Even if there were at that time a few reasonably good environments, the various systems available then were so different that writing portable yet reasonably efficient numerical software was extremely difficult. For instance, as pointed out in [553], sometimes a programmer had to insert multiplications by 1.0 to make a program work reliably.

The IEEE 754-1985 Standard for Binary Floating-Point Arithmetic was released in 1985, but the first meetings of the working group started more than eight years earlier [553]. William Kahan, a professor at the University of California at Berkeley, played a leading role in the development of the standard.

IEEE 754-1985 drastically changed the world of numerical computing, by clearly specifying formats and the way exceptions must be handled, and by requiring correct rounding of the arithmetic operations and the square root. Two years later, another standard, the IEEE 854-1987 Standard for "Radix-Independent" (in fact, radix 2 or 10) Floating-Point Arithmetic was released. It generalized to radix 10 the main ideas of IEEE 754-1985.

[1]IEEE is an acronym for the Institute of Electrical and Electronics Engineers. For more details, see https://www.ieee.org/about/index.html.

J.-M. Muller et al., *Handbook of Floating-Point Arithmetic*,
https://doi.org/10.1007/978-3-319-76526-6_3

IEEE 754-1985 was under revision between 2000 and 2008, and the new standard was adopted in June 2008. In the following, it will be called IEEE 754-2008. In the literature published before its official release (e.g., [116]), IEEE 754-2008 is sometimes called IEEE 754R. IEEE 754-2008 is also known as ISO/IEC/IEEE 60559:2011 *Information technology – Microprocessor Systems – Floating-Point arithmetic* [277]. At the time of writing these lines, IEEE 754-2008 is under revision, and a new version (which will not be much different) is to be released in 2018.

Some languages, such as Java and ECMAScript, are based on IEEE 754-1985. The ISO C11 standard (released in 2011) for the C language has optional support for IEEE 754-1985 in its normative annex F; support for IEEE 754-2008 is planned for the C2x standard. Details will be given in Chapter 6.

The description of the IEEE standard given in this chapter is not exhaustive: standards are big documents that contain many details. Anyone who wants to implement a floating-point arithmetic function compliant to IEEE 754-2008 must carefully read that standard. Also, readers interested in the older IEEE 754-1985 and IEEE 854-1987 standards will find brief descriptions of them in Appendix B.

3.1 The IEEE 754-2008 Standard

3.1.1 Formats

The standard requires that the radix β be 2 or 10, and that for all formats, the minimum and maximum exponents obey the following relation:

$$e_{\min} = 1 - e_{\max} .$$

It defines several *interchange formats*, whose encodings are fully specified as bit strings, and that enable lossless data interchange between different platforms.[2] The main parameters of the interchange formats of size up to 128 bits are given in Tables 3.1 and 3.2.

A format is said to be an *arithmetic format* if all the mandatory operations defined by the standard are supported by the format.

Among the interchange formats, the standard defines five *basic formats* which must also be arithmetic formats: the three binary formats on 32, 64, and 128 bits, and the two decimal formats on 64 and 128 bits. *A conforming implementation must implement at least one of them*.

To implement a function that must return a result in a basic format, it may be convenient to have the intermediate calculations performed in a somewhat wider format:

[2] Platforms may exchange character strings, or binary data. In the latter case, endianness problems (see Section 3.1.1.5) must be considered.

Name	binary16	binary32 (basic)	binary64 (basic)	binary128 (basic)
Former name	N/A	single precision	double precision	N/A
p	11	24	53	113
$e_{\max}$	$+15$	$+127$	$+1023$	$+16383$
$e_{\min}$	-14	-126	-1022	-16382

Table 3.1: Main parameters of the binary interchange formats of size up to 128 *bits specified by the 754-2008 standard [267]. In some articles and software libraries, 128-bit formats were sometimes called "quad precision." However, quad precision was not specified by IEEE 754-1985.*

Name	decimal32	decimal64 (basic)	decimal128 (basic)
p	7	16	34
$e_{\max}$	$+96$	$+384$	$+6144$
$e_{\min}$	-95	-383	-6143

Table 3.2: Main parameters of the decimal interchange formats of size up to 128 *bits specified by the 754-2008 standard [267].*

- the wider precision makes it possible to get a result that will almost always be significantly more accurate than that obtained with the basic format only;

- the wider range will drastically limit the occurrences of "apparent" or "spurious" under/overflow (that is, cases where there is an underflow or overflow in an intermediate result, whereas the final value would have been in the range of the basic format).

For this purpose, the standard also defines an *extended precision* format as a format that extends a basic format with a wider precision and range, in a language-defined or implementation-defined way. Also, an *extendable precision* format is similar to an extended format, but its precision and range are defined under program control. For both extended and extendable formats, the standard does not specify the binary encoding: it may be a straightforward generalization of the encoding of an interchange format, or not. Offering extended or extendable formats is optional.

Finally, the standard requires that conversions between any two supported formats be implemented.

Let us now describe the interchange formats in more detail.

3.1.1.1 Binary interchange format encodings

The binary interchange formats, whose main parameters are given in Table 3.1, are very much like the formats of the older IEEE 754-1985 standard. Actually, the binary32 and binary64 formats correspond to the single- and double-precision formats of IEEE 754-1985: the encodings are exactly the same. However, IEEE 754-1985 did not specify any 128-bit format. What was called "quad precision" in some articles and software libraries was sometimes slightly different from the new binary128 format of IEEE 754-2008. Some authors call "half precision" the binary16 format.

As illustrated in Figure 3.1, a floating-point number is encoded using a 1-bit sign, a W_E-bit exponent field, and a $(p-1)$-bit field for the trailing significand. Let us remind what we mean by "trailing significand." As we are going to see, the information according to which the number is normal or subnormal is encoded in the exponent field. As already explained in Section 2.1.2, since the first bit of the significand of a binary floating-point number is necessarily a "1" if the number is normal (i.e., larger than or equal to $2^{e_{\min}}$) and a "0" if the number is subnormal, there is no need to store that bit. We only store the $p-1$ other bits: these bits constitute the trailing significand (also known as "fraction"). That choice of not storing the leftmost bit of the significand is sometimes called the "hidden bit convention" or the "leading bit convention."

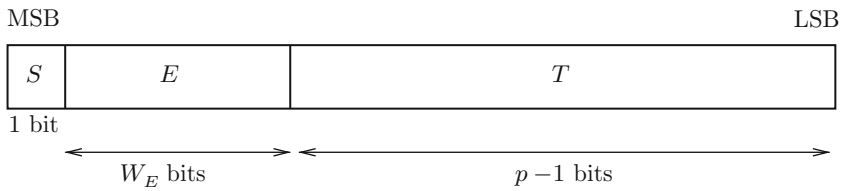

Figure 3.1: Binary interchange floating-point formats [267] (©IEEE, 2008, with permission).

Define E as the integer whose binary representation consists of the bits of the exponent field, T as the integer whose representation consists of the bits of the trailing significand field, and S as the sign bit. The binary encoding (S, E, T) should be interpreted as follows [267]:

- if $E = 2^{W_E} - 1$ (i.e., E is a string of ones) and $T \neq 0$, then a NaN, either quiet (qNaN) or signaling (sNaN), is represented. A quiet NaN is the default result of an invalid operation (e.g., $\sqrt{-5.0}$), and for most operations, a signaling NaN will signal the invalid operation exception whenever it appears as an operand (see Section 3.1.7.1);

- if $E = 2^{W_E} - 1$ and $T = 0$, then $(-1)^S \times (+\infty)$ is represented;
- if $1 \leq E \leq 2^{W_E} - 2$, then the (normal) floating-point number being represented is
$$(-1)^S \times 2^{E-b} \times \left(1 + T \cdot 2^{1-p}\right),$$
where the *bias* b is defined as $b = e_{\max} = 2^{W_E - 1} - 1$ (see Table 3.4 for the actual values);
- if $E = 0$ and $T \neq 0$, then the (subnormal) number being represented is
$$(-1)^S \times 2^{e_{\min}} \times \left(0 + T \cdot 2^{1-p}\right);$$
- if $E = 0$ and $T = 0$, then the number being represented is the signed zero $(-1)^S \times (+0)$.

Biased exponent N_e	Trailing significand $t_1 t_2 \dots t_{p-1}$	Value represented
$111\cdots1_2$	$\neq 000\cdots0_2$	NaN
$111\cdots1_2$	$000\cdots0_2$	$(-1)^s \times \infty$
$000\cdots0_2$	$000\cdots0_2$	$(-1)^s \times 0$
$000\cdots0_2$	$\neq 000\cdots0_2$	$(-1)^s \times 0.t_1 t_2 \dots t_{p-1} \times 2^{e_{\min}}$
$0 < N_e < 2^{W_E} - 1$	any	$(-1)^s \times 1.t_1 t_2 \dots t_{p-1} \times 2^{N_e - b}$

Table 3.3: How to interpret the encoding of an IEEE 754 binary floating-point number.

format	binary16	binary32	binary64	binary128
former name	N/A	single precision	double precision	N/A
storage width	16	32	64	128
$p-1$, trailing significand width	10	23	52	112
W_E, exponent field width	5	8	11	15
$b = e_{\max}$	15	127	1023	16383
$e_{\min}$	-14	-126	-1022	-16382

Table 3.4: Parameters of the encodings of binary interchange formats [267]. As stated above, in some articles and software libraries, 128-bit formats were called "quad precision." However, quad precision was not specified by IEEE 754-1985.

	Smallest positive subnormal $2^{e_{\min}+1-p}$	Smallest positive normal $2^{e_{\min}}$	Largest finite $2^{e_{\max}}(2-2^{1-p})$
binary16	2^{-14-10} $\approx 5.96 \times 10^{-8}$	2^{-14} $\approx 6.10 \times 10^{-5}$	$(2-2^{-10}) \times 2^{15}$ $= 65504$
binary32	$2^{-126-23}$ $\approx 1.401 \times 10^{-45}$	2^{-126} $\approx 1.175 \times 10^{-38}$	$(2-2^{-23}) \times 2^{127}$ $\approx 3.403 \times 10^{38}$
binary64	$2^{-1022-52}$ $\approx 4.941 \times 10^{-324}$	2^{-1022} $\approx 2.225 \times 10^{-308}$	$(2-2^{-52}) \times 2^{1023}$ $\approx 1.798 \times 10^{308}$
binary128	$2^{-16382-112}$ $\approx 6.68 \times 10^{-4966}$	2^{-16382} $\approx 3.36 \times 10^{-4932}$	$(2-2^{-112}) \times 2^{16383}$ $\approx 1.19 \times 10^{4932}$

Table 3.5: Extremal values of the IEEE 754-2008 binary interchange formats.

Datum	Sign	Biased exponent	Trailing significand
-0	1	00000000	00000000000000000000000
$+0$	0	00000000	00000000000000000000000
$-\infty$	1	11111111	00000000000000000000000
$+\infty$	0	11111111	00000000000000000000000
NaN	0	11111111	nonzero string
5	0	10000001	01000000000000000000000

Table 3.6: Binary encoding of various floating-point data in the binary32 format.

Table 3.3 summarizes this encoding. The sizes of the various fields are given in Table 3.4, and extremal values are given in Table 3.5.

Table 3.6 gives examples of the binary encoding of various floating-point values in the binary32 format. Let us now detail two examples.

Example 3.1 (Binary encoding of a normal number)**.** *Consider the binary32 number x whose binary encoding is*

sign	exponent	trailing significand
0	01101011	01010101010101010101010

- *the bit sign of x is a zero, which indicates that $x \geq 0$;*
- *the biased exponent is neither 00000000_2 nor 11111111_2, which indicates that x is a normal number. It is $01101011_2 = 107_{10}$, hence, since the bias in binary32 is 127, the actual exponent of x is $107 - 127 = -20$;*

- *by placing the hidden bit (which is a* 1*, since* x *is not subnormal) at the left of the trailing significand, we get the significand of* x*:*

$$1.01010101010101010101010_2 = \frac{5592405}{2^{22}};$$

- *hence,* x *is equal to*

$$\begin{aligned}\frac{5592405}{2^{22}} \times 2^{-20} &= \frac{5592405}{2^{42}} \\ &= 0.00000127156567941710818558931350708007812 5.\end{aligned}$$

Example 3.2 (Binary encoding of a subnormal number)**.** *Consider the binary32 number* x *whose binary encoding is*

sign	exponent	trailing significand
1	00000000	01100000000000000000000

- *the bit sign of* x *is a one, which indicates that* $x \leq 0$*;*
- *the biased exponent is* 00000000*, which indicates that* x *is a subnormal number. It is not a zero, since the significand field is not a string of zeros. Hence, the real exponent of* x *is* $e_{\min} = -126$*;*
- *by placing the hidden bit (which is a* 0*, since* x *is subnormal) at the left of the trailing significand, we get the significand of* x*:*

$$0.01100000000000000000000_2 = \frac{3}{8};$$

- *hence,* x *is equal to*

$$\begin{aligned}-\frac{3}{8} \times 2^{-126} &= -4.40810381558357815488276201458342129181999583789 \\ &\quad 3282056578188985440647229552268981933593 75 \times 10^{-39}.\end{aligned}$$

3.1.1.2 Decimal interchange format encodings

The decimal format encodings are more complex than the binary ones, for several reasons.

- *Two* encoding systems are specified, called the *decimal* and *binary* encodings: the members of the revision committee could not agree on a single encoding system. The reason for that is that the binary encoding makes a *software* implementation of decimal arithmetic easier, whereas the decimal encoding is more suited for a *hardware* implementation. And yet,

despite this problem, one must understand that the set of representable floating-point numbers is *the same* for both encoding systems, so that this additional complexity will be transparent for most users. Also, a conforming implementation must provide conversions between these two encoding systems [267, §5.5.2].

- Contrary to the binary interchange formats, the sign, exponent, and (trailing) significand fields are not fully separated: to preserve as much accuracy as possible, some information on the significand is partly encoded in what used to be the exponent field and is hence called the *combination* field.

- In the decimal formats, the representations (M, e) are not normalized (i.e., it is not required that e should be minimal), so that a decimal floating-point number may have multiple valid representations. The set of the various representations of a same number is called a *cohort*. As a consequence, we will have to explain which exponent is *preferred* for the result of an arithmetic operation.

- Even if the representation itself (that is, values of the sign, exponent, and significand) of a number x (or an infinite, or a NaN) and the type of encoding (binary or decimal) are chosen, a same number (or infinite, or NaN) can still be encoded by different bit strings. One of them will be said to be *canonical*.

Roughly speaking, the difference between the decimal and binary encodings of decimal floating-point numbers originates from a choice in the encoding of the significand. The integral significand is a nonnegative integer less than or equal to $10^p - 1$. One can encode it either in binary (which gives the binary encoding) or in decimal (which gives the decimal encoding).

Concerning the decimal encoding, in the early days of computer arithmetic, people would use binary coded decimal (BCD) encoding, where each decimal digit was encoded by four bits. That encoding was quite wasteful, since among the 16 possible values representable on four bits, only 10 were actually used. Since $2^{10} = 1024$ is very close to 10^3 (and larger), one can design a much denser encoding by encoding three consecutive decimal digits by a 10-bit *declet* [95]. Many possible ways of performing this encoding are possible. The one chosen by the standard committee for the decimal encoding of decimal numbers is given in Tables 3.10 (declet to decimal) and 3.11 (decimal to declet). It was designed to facilitate conversions: all these tables have a straightforward hardware implementation and can be implemented in three gate levels [184]. Note that Table 3.10 has 1024 possible inputs and 1000 possible outputs (hence, there is some redundancy), and Table 3.11 has 1000 possible inputs and outputs. This implies that there are 24 "noncanonical"

bit patterns,[3] which are accepted as input values but cannot result from an arithmetic operation. An encoding that contains a noncanonical bit pattern is called *noncanonical*.

Let us explain more precisely why there is no clear separation between an exponent field and a significand field (as is the case in the binary formats). Consider as an example the decimal64 format (see Table 3.2). In that format, $e_{\max} = 384$ and $e_{\min} = -383$; therefore, there are 768 possible values of the exponent. Storing all these values in binary in an exponent field would require 10 bits. Since we can store 1024 possible values in a 10-bit field, that would be wasteful. This explains why it was decided to put all the information about the exponent plus some other information in a "combination field," where will be stored:

- "classification" information: Does the datum represent a finite number, or $\pm\infty$, or a NaN (see Section 3.1.7)?
- the exponent (if the datum represents a finite number);
- the leading part of the significand (if the datum represents a finite number); more precisely, the leading decimal digit (if the *decimal* encoding is used) or 3 to 4 leading bits (if the *binary* encoding is used). The remaining significand bits/digits are stored in the trailing significand field.

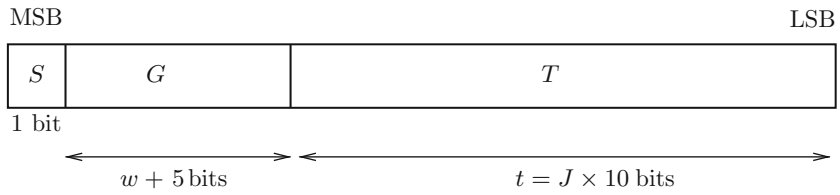

Figure 3.2: Decimal interchange floating-point formats [267] (©IEEE, 2008, with permission).

The widths of the various fields are given in Table 3.7. It is important to note that in this table the bias b is related to the *quantum exponent* (see Section 2.1.1), which means that if e is the exponent of x, if $q = e - p + 1$ is its quantum exponent, then the biased exponent E is

$$E = q + b = e - p + 1 + b.$$

The floating-point format illustrated in Figure 3.2, with a 1-bit sign, a $(w + 5)$-bit *combination* field, and a $t = (J \times 10)$-bit *trailing significand* field must be interpreted as follows [267]:

[3] Those of the form $01x11x111x$, $10x11x111x$, or $11x11x111x$.

	decimal32	decimal64	decimal128
storage width	32	64	128
$t = 10J$, trailing significand width	20	50	110
$w + 5$, combination field width	11	13	17
$b = E - (e - p + 1)$, bias	101	398	6176

Table 3.7: Width (in bits) of the various fields in the encodings of the decimal interchange formats of size up to 128 *bits [267].*

- if the most significant five bits of G (numbered from the left, G_0 to G_4) are all ones, then the datum being represented is a NaN (see Section 3.1.7.1). Moreover, if G_5 is 1, then it is an sNaN, otherwise it is a qNaN. In a canonical encoding of a NaN, the bits G_6 to G_{w+4} are all zeros;

- if the most significant five bits of G are 11110, then the value being represented is $(-1)^S \times (+\infty)$. Moreover, the canonical encodings of infinity have bits G_5 to G_{w+4} as well as trailing significand T equal to 0;

- if the most significant four bits of G, i.e., G_0 to G_3, are not all ones, then the value being represented is a finite number, equal to

$$(-1)^S \times 10^{E-b} \times C. \tag{3.1}$$

  Here, the value $E - b$ is the *quantum exponent* (see Section 2.1.1), where b, the *exponent bias*, is equal to 101, 398, and 6176 for the decimal32, decimal64, and decimal128 formats, respectively. E and C are obtained as follows.

  1. If the *decimal encoding* is used for the significand, then the least significant w bits of the biased exponent E are made up of the bits G_5 to G_{w+4} of G, whereas the most significant two bits of E and the most significant two digits of C are obtained as follows:
     - if the most significant five bits $G_0G_1G_2G_3G_4$ of G are of the form $110xx$ or $1110x$, then the leading significand digit C_0 is $8 + G_4$ (which equals 8 or 9), and the leading biased exponent bits are G_2G_3;
     - if the most significant five bits of G are of the form $0xxxx$ or $10xxx$, then the leading significand digit C_0 is $4G_2 + 2G_3 + G_4$ (which is between 0 and 7), and the leading biased exponent bits are G_0G_1.

The $p-1=3J$ decimal digits $C_1,\ldots,C_{p-1}$ of C are encoded by T, which contains J declets encoded in densely packed decimal (see Tables 3.10 and 3.11). Note that if the most significant five bits of G are 00000, 01000, or 10000, and $T=0$, then the significand is 0 and the number represented is $(-1)^S\times(+0)$.

Table 3.8 summarizes these rules.

2. If the *binary encoding* is used for the significand, then
   - if G_0G_1 is 00, 01, or 10, then E is made up of the bits G_0 to G_{w+1}, and the binary encoding of the significand C is obtained by prefixing the last 3 bits of G (i.e., $G_{w+2}G_{w+3}G_{w+4}$) to T;
   - if G_0G_1 is 11 and G_2G_3 is 00, 01 or 10, then E is made up of the bits G_2 to G_{w+3}, and the binary encoding of the significand C is obtained by prefixing $100G_{w+4}$ to T.

   Remember that the maximum value of the integral significand is $10^p-1=10^{3J+1}-1$. If the value of C computed as above is larger than that maximum value, then the value used for C will be zero [267], and the encoding will not be canonical. Table 3.9 summarizes these rules.

A decimal software implementation of IEEE 754-2008, based on the binary encoding of the significand, is presented in [116, 117]. Interesting information on decimal arithmetic can be found in [121]. A decimal floating-point multiplier that uses the decimal encoding of the significand is presented in [192].

Example 3.3 (Finding the encoding of a decimal number assuming decimal encoding of the significands)**.** *Consider the number*

$$x=3.141592653589793\times 10^0=3141592653589793\times 10^{-15}.$$

This number is exactly representable in the decimal64 format. Let us find its encoding, assuming decimal encoding of the significands.

- *First, the sign bit is* 0*;*
- *since the quantum exponent is* -15*, the biased exponent will be* 383 *(see Table 3.7), whose* 10*-bit binary representation is* 0101111111_2*. One should remember that the exponent is not directly stored in an exponent field, but combined with the most significant digit of the significand in a* combination field G*. Since the leading significand digit is* 3*, we are in the case*

G	T	Datum being represented
$11111\,0xxx\cdots x$	any	qNaN
$11111\,1xxx\cdots x$	any	sNaN
$11110\,xxxx\cdots x$	any	$(-1)^S \times (+\infty)$
$110xx\cdots$ or $1110x\cdots$	$T_0T_1\cdots T_{10J-1}$	$(-1)^S \times 10^{\overbrace{G_2G_3G_5G_6\cdots G_{w+4}}^{\text{binary}}-b} \times \underbrace{(8+G_4)C_1C_2\cdots C_{p-1}}_{\text{decimal}}$ with $C_{3j+1}C_{3j+2}C_{3j+3}$ deduced from $T_{10j}T_{10j+1}T_{10j+2}\cdots T_{10j+9}$ for $0 \le j < J$ using Table 3.10.
$0xxxx\cdots$ or $10xxx\cdots$	$T_0T_1\cdots T_{10J-1}$	$(-1)^S \times 10^{\overbrace{G_0G_1G_5G_6\cdots G_{w+4}}^{\text{binary}}-b} \times \underbrace{(4G_2+2G_3+G_4)C_1C_2\cdots C_{p-1}}_{\text{decimal}}$ with $C_{3j+1}C_{3j+2}C_{3j+3}$ deduced from $T_{10j}T_{10j+1}T_{10j+2}\cdots T_{10j+9}$ for $0 \le j < J$ using Table 3.10.

Table 3.8: Decimal encoding of a decimal floating-point number (IEEE 754-2008).

G	T	Datum being represented
$11111\,0xxx\cdots x$	any	qNaN
$11111\,1xxx\cdots x$	any	sNaN
$11110\,xxxx\cdots x$	any	$(-1)^S \times (+\infty)$
$00xxx\cdots$ **or** $01xxx\cdots$ **or** $10xxx\cdots$	$T_0T_1\cdots T_{10J-1}$	$(-1)^S \times 10^{\overbrace{G_0G_1G_2\cdots G_{w+1}}^{\text{binary}}-b} \times \underbrace{G_{w+2}G_{w+3}G_{w+4}T_0T_1\cdots T_{10J-1}}_{\text{binary}}$ if $G_{w+2}G_{w+3}G_{w+4}T_0T_1\cdots T_{10J-1} \leq 10^p-1$, otherwise $(-1)^S \times (+0)$.
$1100xxx\cdots$ **or** $1101xxx\cdots$ **or** $1110xxx\cdots$	$T_0T_1\cdots T_{10J-1}$	$(-1)^S \times 10^{\overbrace{G_2G_3G_4\cdots G_{w+3}}^{\text{binary}}-b} \times \underbrace{100G_{w+4}T_0T_1\cdots T_{10J-1}}_{\text{binary}}$ if $100G_{w+4}T_0T_1\cdots T_{10J-1} \leq 10^p-1$, otherwise $(-1)^S \times (+0)$.

Table 3.9: Binary encoding of a decimal floating-point number (IEEE 754-2008).

$b_6 b_7 b_8 b_3 b_4$	d_0	d_1	d_2
$0xxxx$	$4b_0 + 2b_1 + b_2$	$4b_3 + 2b_4 + b_5$	$4b_7 + 2b_8 + b_9$
$1\ 0\ 0xx$	$4b_0 + 2b_1 + b_2$	$4b_3 + 2b_4 + b_5$	$8 + b_9$
$1\ 0\ 1xx$	$4b_0 + 2b_1 + b_2$	$8 + b_5$	$4b_3 + 2b_4 + b_9$
$1\ 1\ 0xx$	$8 + b_2$	$4b_3 + 2b_4 + b_5$	$4b_0 + 2b_1 + b_9$
$1\ 1\ 1\ 0\ 0$	$8 + b_2$	$8 + b_5$	$4b_0 + 2b_1 + b_9$
$1\ 1\ 1\ 0\ 1$	$8 + b_2$	$4b_0 + 2b_1 + b_5$	$8 + b_9$
$1\ 1\ 1\ 1\ 0$	$4b_0 + 2b_1 + b_2$	$8 + b_5$	$8 + b_9$
$1\ 1\ 1\ 1\ 1$	$8 + b_2$	$8 + b_5$	$8 + b_9$

Table 3.10: Decoding the declet $b_0 b_1 b_2 \cdots b_9$ of a densely packed decimal encoding to three decimal digits $d_0 d_1 d_2$ [267] (©IEEE, 2008, with permission).

$d_0^0\ d_1^0\ d_2^0$	$b_0\ b_1\ b_2$	$b_3\ b_4\ b_5$	b_6	$b_7\ b_8\ b_9$
$0\ 0\ 0$	$d_0^1\ d_0^2\ d_0^3$	$d_1^1\ d_1^2\ d_1^3$	0	$d_2^1\ d_2^2\ d_2^3$
$0\ 0\ 1$	$d_0^1\ d_0^2\ d_0^3$	$d_1^1\ d_1^2\ d_1^3$	1	$0\ 0\ d_2^3$
$0\ 1\ 0$	$d_0^1\ d_0^2\ d_0^3$	$d_2^1\ d_2^2\ d_1^3$	1	$0\ 1\ d_2^3$
$0\ 1\ 1$	$d_0^1\ d_0^2\ d_0^3$	$1\ 0\ d_1^3$	1	$1\ 1\ d_2^3$
$1\ 0\ 0$	$d_2^1\ d_2^2\ d_0^3$	$d_1^1\ d_1^2\ d_1^3$	1	$1\ 0\ d_2^3$
$1\ 0\ 1$	$d_1^1\ d_1^2\ d_0^3$	$0\ 1\ d_1^3$	1	$1\ 1\ d_2^3$
$1\ 1\ 0$	$d_2^1\ d_2^2\ d_0^3$	$0\ 0\ d_1^3$	1	$1\ 1\ d_2^3$
$1\ 1\ 1$	$0\ 0\ d_0^3$	$1\ 1\ d_1^3$	1	$1\ 1\ d_2^3$

Table 3.11: Encoding the three consecutive decimal digits $d_0 d_1 d_2$, each of them being represented in binary by four bits (e.g., d_0 is written in binary $d_0^0 d_0^1 d_0^2 d_0^3$), into a 10-bit declet $b_0 b_1 b_2 \cdots b_9$ of a densely packed decimal encoding [267] (©IEEE, 2008, with permission).

> *"if the most significant five bits of G are of the form $0xxxx$ or $10xxx$, then the leading significand digit C_0 is $4G_2 + 2G_3 + G_4$ (which is between 0 and 7), and the leading biased exponent bits are G_0G_1."*

Hence,

- *G_0 and G_1 are the leading biased exponent bits, namely 0 and 1;*
- *G_2, G_3, and G_4 are the binary encoding of the first significand digit 3, i.e., $G_2 = 0$, and $G_3 = G_4 = 1$; and*
- *the bits G_5 to G_{12} are the least significant bits of the biased exponent, namely 01111111.*

- *Now, the trailing significand field T is made up of the five declets of the densely packed decimal encoding of the trailing significand 141592653589793:*
  - *the 3-digit chain 141 is encoded by the declet 0011000001, according to Table 3.11;*
  - *592 is encoded by the declet 1010111010;*
  - *653 is encoded by the declet 1101010011;*
  - *589 is encoded by the declet 1011001111;*
  - *793 is encoded by the declet 1110111011.*
- *Therefore, the encoding of x is*

$$\underbrace{0}_{\text{sign}}\ \underbrace{0101101111111}_{\text{combination field}}\ \ldots$$

$$\underbrace{00110000011010111010110101001110110011111110111011}_{\text{trailing significand field}}.$$

Example 3.4 (Finding an encoding of a decimal number assuming binary encoding of the significands)**.** *Consider the number*

$$x = 3.141592653589793 \times 10^0 = 3141592653589793 \times 10^{-15}.$$

(This is the same number as in Example 3.3, but now we consider binary encoding, in the decimal64 format.) The sign bit will be zero. Since 3141592653589793 is a 16-digit integer that does not end with a 0, the quantum exponent can only be -15; therefore, the biased exponent E will be $398-15 = 383$, whose binary representation is 101111111. The binary representation of the integral significand of x is

$$10\,\underbrace{1100101001010000110000101000100101011011010010000 1}_{t = 50 \text{ bits (trailing significand)}}.$$

The length of this bit string is 52, which is less than $t + 4 = 54$, hence we are not in the case

> *"if G_0G_1 is 11 and G_2G_3 is 00, 01 or 10, then E is made up of the bits G_2 to G_{w+3}, and the binary encoding of the significand C is obtained by prefixing $100G_{w+4}$ to T,"*

which means that we are in the case

> *"if G_0G_1 is 00, 01, or 10, then E is made up of the bits G_0 to G_{w+1}, and the binary encoding of the significand C is obtained by prefixing the last 3 bits of G (i.e., $G_{w+2}G_{w+3}G_{w+4}$) to T."*

Therefore, $G_0G_1 \dots G_9 = 0101111111$, $G_{10}G_{11}G_{12} = 010$ and T is made up with the 50 rightmost bits of t, resulting in an encoding of x as

$$T = 11001010010100001100001010001001010110110100100001.$$

Example 3.5 (Finding the value of a decimal floating-point number from its encoding, assuming decimal encoding of the significand)**.** *Consider the decimal32 number x whose encoding is*

$$\underbrace{1}_{\text{sign}} \quad \underbrace{11101101101}_{\text{combination field } G} \quad \underbrace{01101001101111000011}_{\text{trailing significand field } T}.$$

- *Since the bit sign is 1, we have $x \leq 0$;*
- *since the most significant four bits of G are not all ones, x is not an infinity or a NaN;*
- *by looking at the most significant four bits of G, we deduce that we are in the case*

  > *if the most significant five bits $G_0G_1G_2G_3G_4$ of G are of the form $110xx$ or $1110x$, then the leading significand digit C_0 is $8 + G_4$ (which equals 8 or 9), and the leading biased exponent bits are G_2G_3.*

  Therefore, the leading significand bit C_0 is $8 + G_4 = 9$, and the leading biased exponent bits are 10. The least significant bits of the exponent are 101101; therefore, the biased exponent is $10101101_2 = 173_{10}$. Hence, the (unbiased) quantum exponent of x is $173 - 101 = 72$;
- *the trailing significand field T is made up of two declets, 0110100110 and 1111000011. According to Table 3.10,*
  - *the first declet encodes the 3-digit chain 326;*
  - *the second declet encodes 743.*
- *Therefore, x is equal to*

$$-9326743 \times 10^{72} = -9.326743 \times 10^{78}.$$

Example 3.6 (Finding the value of a decimal floating-point number from its encoding, assuming binary encoding of the significand)**.** *Consider the decimal32 number x whose encoding is*

$$\underbrace{1}_{\text{sign}} \quad \underbrace{11101101101}_{\text{combination field } G} \quad \underbrace{01101001101111000011}_{\text{trailing significand field } T}.$$

(It is the same bit string as in Example 3.5, but now we consider binary encoding.)

- *Since the bit sign is* 1*, we have $x \leq 0$;*
- *since the most significant four bits of G are not all ones, x is not an infinity or a NaN;*
- *since $G_0G_1 = 11$ and $G_2G_3 = 10$, we are in the case*

  > *if G_0G_1 is* 11 *and G_2G_3 is* 00*,* 01*, or* 10*, then E is made up of the bits G_2 to G_{w+3}, and the binary encoding of the significand C is obtained by prefixing $100G_{w+4}$ to T.*

  Therefore, the biased exponent E is $10110110_2 = 182_{10}$, which means that the quantum exponent of x is $182 - 101 = 81$, and the integral significand of x is

  $$10010110100110111100001 1_2 = 9870275_{10}.$$

- *Therefore, x is equal to*

  $$-9870275 \times 10^{81} = -9.870275 \times 10^{87}.$$

3.1.1.3 Larger formats

The IEEE 754-2008 standard also specifies larger interchange formats for widths of at least 128 bits that are multiples of 32 bits. Their parameters are given in Table 3.12, and examples are given in Tables 3.13 and 3.14. This allows one to define "big" (yet, fixed) precisions. A format is fully defined from its radix (2 or 10) and size: the various parameters (precision, $e_{\min}$, $e_{\max}$, bias, etc.) are derived from them, using the formulas given in Table 3.12. Hence, binary1024 or decimal512 will mean the same thing on all platforms.

3.1.1.4 Extended and extendable precisions

Beyond the interchange formats, the IEEE 754-2008 standard partially specifies the parameters of possible *extended precision* and *extendable precision* formats. These formats are optional, and their binary encoding is not specified.

Parameter	Binary$k$ format (k is a multiple of 32)	Decimal$k$ format (k is a multiple of 32)
k	≥ 128	≥ 32
p	$k - \lfloor 4\log_2(k) \rceil + 13$	$9 \times \frac{k}{32} - 2$
t	$p - 1$	$(p-1) \times 10/3$
w	$k - t - 1$	$k - t - 6$
$e_{\max}$	$2^{w-1} - 1$	$3 \times 2^{w-1}$
$e_{\min}$	$1 - e_{\max}$	$1 - e_{\max}$
b	$e_{\max}$	$e_{\max} + p - 2$

Table 3.12: Parameters of the interchange formats. $\lfloor u \rceil$ is u rounded to the nearest integer, t is the trailing significand width, w is the width of the exponent field for the binary formats, and the width of the combination field minus 5 for the decimal formats, and b is the exponent bias [267], (©IEEE, 2008, with permission).

Format	p	t	w	$e_{\min}$	$e_{\max}$	b
binary256	237	236	19	-262142	$+262143$	262143
binary1024	997	996	27	-67108862	$+67108863$	67108863

Table 3.13: Parameters of the binary256 and binary1024 interchange formats deduced from Table 3.12. Variables p, t, w, $e_{\min}$, $e_{\max}$, and b are the precision, the trailing significant field length, the exponent field length, the minimum exponent, the maximum exponent, and the exponent bias, respectively.

- An *extended precision format* extends a basic format with a wider precision and range, and is either language defined or implementation defined. The constraints on these wider precisions and ranges are listed in Table 3.15. The basic idea behind these formats is that they should be used to carry out intermediate computations, in order to return a final result in the associated basic formats. The wider precision makes it possible to get a result that will generally be more accurate than that obtained with the basic formats only, and the wider range will drastically limit the cases of "apparent under/overflow" (that is, cases where there is an underflow or overflow in an intermediate result, whereas the final value would have been representable). An example of extended precision format, still of importance, is the 80-bit "double-extended format" (radix 2, precision 64, extremal exponents -16382 and 16383) specified by the IA-32 instruction set.

Format	p	t	$w+5$	$e_{\max}$	b
decimal256	70	230	25	+1572864	1572932
decimal512	142	470	41	+103079215104	103079215244

Table 3.14: Parameters of the decimal256 and decimal512 interchange formats deduced from Table 3.12. $e_{\min}$ (not listed in the table) equals $1 - e_{\max}$. Variables p, t, w, $e_{\min}$, $e_{\max}$, and b are the precision, the combination field length, the exponent field length, the minimum exponent, the maximum exponent, and the exponent bias, respectively.

	Extended formats associated with:				
Parameter	binary32	binary64	binary128	decimal64	decimal128
$p \geq$	32	64	128	22	40
$e_{\max} \geq$	1023	16383	65535	6144	24576
$e_{\min} \leq$	−1022	−16382	−65534	−6143	−24575

Table 3.15: Extended format parameters in the IEEE 754-2008 standard [267] (©IEEE, 2008, with permission).

- An *extendable precision format* is a format whose precision and range are defined under user or program control. The standard says that language standards supporting extendable precision shall allow users to specify p and $e_{\max}$ (or, possibly, p only with constraints on $e_{\max}$), and define $e_{\min} = 1 - e_{\max}$.

3.1.1.5 Little-endian, big-endian

The IEEE 754 standard specifies how floating-point data are encoded, but only as a sequence of bits. How such a sequence of bits is ordered in the memory depends on the platform. In general, the bits are grouped into bytes, and these bytes are ordered according to what is called the *endianness* of the platform. On big-endian platforms the most significant byte has the lowest address. This is the opposite on little-endian platforms.

Some architectures, such as IA-64, ARM, and PowerPC are *bi-endian*, i.e., they may be either little-endian or big-endian depending on their configuration.

For instance, the binary64 number that is closest to $-7.0868766365730135 \times 10^{-268}$ is encoded by the sequence of bytes 11 22 33 44 55 66 77 88 on little-endian platforms and by 88 77 66 55 44 33 22 11 on big-endian platforms.

When a format fits into several words (for instance, a 128-bit format on a 64-bit processor), the order of the words does not necessarily follow the endianness of the platform.

Endianness must be taken into account by users who want to exchange data in binary between different platforms.

3.1.2 Attributes and rounding

The IEEE 754-2008 standard defines *attributes* as parameters, attached to a program block, that specify some of its numerical and exception semantics. The availability of *rounding direction attributes* is mandatory, whereas the availability of *alternate exception-handling attributes*, *preferred width attributes*, *value-changing optimization attributes*, and *reproducibility attributes* is recommended only. Language standards must provide for constant specification of the attributes, and should also allow for dynamic-mode specification of them.

3.1.2.1 Rounding direction attributes

IEEE 754-2008 requires that the following operations and functions (among others), called "General-computational operations" in [267], be correctly rounded:

- arithmetic operations: addition, subtraction, multiplication, division, fused multiply-add (FMA);
- unary functions: square root, conversion from a supported format to another supported format.

Also, most conversions from a variable in a binary format to a decimal character sequence (typically for printing), or from a decimal character sequence to a binary format, must be correctly rounded: see Section 3.1.5.

Let us now describe the various *directed rounding attributes*.

- The roundTowardPositive attribute corresponds to what was called the round-toward $+\infty$ mode in IEEE 754-1985. The rounding function (see Section 2.2) is RU.
- The roundTowardNegative attribute corresponds to what was called the round-toward $-\infty$ mode in IEEE 754-1985. The rounding function is RD.
- The roundTowardZero attribute corresponds to what was called the round-toward-zero mode in IEEE 754-1985. The rounding function is RZ.

Concerning rounding to nearest, the situation is somewhat different. IEEE 754-1985 had one round-to-nearest mode only, named *round-to-nearest*

even. The IEEE 754-2008 standard specifies two *rounding direction attributes to nearest*, which differ in the way of handling the case when an exact result is halfway between two consecutive floating-point numbers:

- roundTiesToEven attribute: if the two nearest floating-point numbers bracketing the exact result are equally near, the one whose least significant significand digit is even is delivered. This corresponds to the *round-to-nearest-even mode* of IEEE 754-1985 (in binary) and IEEE 854-1987[4];
- roundTiesToAway attribute: in the same case as above, the value whose magnitude is larger is delivered.

For instance, in the decimal64 format ($p = 16$), if the exact result of some arithmetic operation is 1.2345678901234565, then the result returned should be 1.234567890123456 with the roundTiesToEven attribute, and 1.234567890123457 with the roundTiesToAway attribute.

There is another important issue with rounding to nearest: In radix-β, precision-p arithmetic, a number of absolute value greater than or equal to $\beta^{e_{\max}}(\beta - \frac{1}{2}\beta^{-p+1})$ will be rounded to infinity (with the appropriate sign). This of course is not what one would infer from a naive understanding of the words *round to nearest*, but the advantage is clear: when the result of an arithmetic operation is a normal number (including the largest one, $\Omega = \beta^{e_{\max}}(\beta - \beta^{-p+1})$), we know that the relative error induced by that operation is small. If huge numbers were rounded to the floating-point value that is really closest to them (namely, $\pm\Omega$), we would have no bound on the relative error induced by an arithmetic operation whose result is $\pm\Omega$.

The standard requires that an implementation (be it binary or decimal) provide the roundTiesToEven and the three directed rounding attributes. A *decimal* implementation must also provide the roundTiesToAway attribute (this is not required for binary implementations).

Having roundTiesToEven as the default rounding direction attribute is mandatory for binary implementations and recommended for decimal implementations. Whereas roundTiesToEven has several advantages (see [342]), roundTiesToAway is useful for some accounting calculations. This is why it is required for radix-10 implementations only, the main use of radix 10 being financial calculations. For instance, the European Council Regulation No. 1103/97 of June 17th 1997 on certain provisions relating to the introduction of the Euro sets out a number of rounding and conversion rules. Among them,

[4]The case where these floating-point numbers both have an odd least significant significand digit (this can occur in precision 1 only, possibly when converting a number such as 9.5 into a decimal string for instance) has been forgotten in the standard, but for the next revision, it has been proposed—See http://speleotrove.com/misc/IEEE754-errata.html—to deliver the one larger in magnitude.

> If the application of the conversion rate gives a result which is exactly half-way, the sum shall be rounded up.

3.1.2.2 Alternate exception-handling attributes

It is recommended (but not required) that language standards define means for programmers to be able to associate alternate exception-handling attributes with a block. The alternate exception handlers provide lists of exceptions (invalid operation, division by zero, overflow, underflow, inexact, all exceptions) and specify what should be done when each of these exceptions is signaled. If no alternate exception-handling attribute is associated with a block, the exceptions are treated as explained in Section 3.1.6 (default exception handling).

3.1.2.3 Preferred width attributes

Consider an expression of the form

$$((a+b)\times c+(d+e))\times f,$$

where a, b, c, d, e, and f are floating-point numbers, represented in the same radix, but possibly with different formats. Variables a, b, c, d, e, and f are *explicit*, but during the evaluation of that expression, there will also be *implicit* variables; for instance, the result r_1 of the calculation of $a+b$, and the result r_2 of the calculation of $r_1 \times c$. When more than one format is available on the system under consideration, an important question arises: *In which format should these intermediate values be represented?* That point was not very clear in IEEE 754-1985. Many choices are possible for the "destination width" of an implicit variable. For instance:

- one might prefer to always have these implicit variables in the largest format provided in hardware. This choice will generally lead to more accurate computations (although it is quite easy to construct counterexamples for which this is not the case);
- one might prefer to clearly specify a destination format. This will increase the portability of the program being written;
- one might require the implicit variables to be of the same format as the operands (and, if the operands are of different formats, to be of the widest format among the operands). This also will improve the portability of programs and will ease the use of smart algorithms such as those presented in Chapters 4, 5, and 11.

The standard recommends (but does not require) that the following preferredWidthNone and preferredWidthFormat attributes should be defined by language standards.

preferredWidthNone attribute: When the user specifies a preferredWidthNone attribute for a block, the destination width of an operation is the maximum of the operand widths.

preferredWidthFormat attributes: When the user specifies a preferredWidthFormat attribute for a block, the destination width is the maximum of the width of the preferredWidthFormat and the operand widths.

3.1.2.4 Value-changing optimization attributes

Some optimizations (e.g., generation of FMAs, use of distributive and associative laws) can enhance performance in terms of speed, and yet seriously hinder the portability and reproducibility of results. Therefore, it makes sense to let the programmer decide whether to allow them or not. The value-changing optimization attributes are used in this case. The standard recommends that language standards should clearly define what is called the "literal meaning" of the source code of a program (that is, the order of the operations and the destination formats of the operations). By default, the implementations should preserve the literal meaning. Language standards should define attributes for allowing or disallowing value-changing optimizations such as:

- applying relations such as $x \cdot y + x \cdot z = x \cdot (y + z)$ (distributivity), or $x + (y + z) = (x + y) + z$ (associativity);
- using FMAs for replacing, e.g., an expression of the form $a \cdot b + c \cdot d$ by FMA$(a, b, c \cdot d)$;
- using larger formats for storing intermediate results.

3.1.2.5 Reproducibility attributes

The standard requires that conforming language standards should define ways of expressing when reproducible results are required. To get reproducible results, the programs must be translated into an unambiguous sequence of reproducible operations in reproducible formats. As explained in the standard [267], when the user requires reproducible results:

- the execution behavior must preserve what the standard calls the *literal meaning* of the source code[5];
- conversions from and to external character strings must not bound the value of the maximum precision H (see Section 3.1.5) of these strings;

[5]This implies that the language standards must specify what that literal meaning is: order of operations, destination formats of operations, etc.

- when the reproducibility of some operation is not guaranteed, the user must be warned;
- only default exception handling is allowed.

3.1.3 Operations specified by the standard

3.1.3.1 Arithmetic operations and square root

The IEEE 754-2008 standard requires that addition, subtraction, multiplication, FMA, division, and square root of operands of any supported format be provided, with correct rounding (according to the supported rounding direction attributes) to any of the supported formats with same radix.

In other words, it not only mandates support of these basic operations with identical input and output formats (*homogeneous* operations in the standard's terminology), but also with different input and output formats (*formatOf*-operations in the standard). The hardware typically supports the homogeneous case, which is the most common, and heterogeneous operations can be provided at low cost in software [400].

When the sum or difference of two numbers is exactly zero, the returned result is zero, with a $+$ sign in the round-to-nearest, round-toward-zero, and round-toward $+\infty$ modes, and with a $-$ in the round-toward $-\infty$ mode, except for $x + x$ and $x - (-x)$ with x being ± 0, in which case the result has the same sign as x. As noticed by Boldo et al. [51], this means that $x + 0$ cannot be blindly replaced by x: when x is -0, assuming round-to-nearest, $+0$ must be returned. Compiler designers should be aware of such subtleties (see Section 6.2.3.4).

Concerning square root, the result is defined and has a positive sign for all input values greater than or equal to zero, with the exception[6] that $\sqrt{-0} = -0$.

3.1.3.2 Remainders

The remainder must also be provided, but only in homogeneous variants. There are several different definitions of remainders [62]; here is the one chosen for the standard. If x is a finite floating-point number and y is a finite, nonzero floating-point number, then the remainder $r = x$ REM y is defined as

1. $r = x - y \times n$, where n is the integer nearest to the exact value x/y;

[6] This rule (which may help in implementing complex functions [317]) may seem strange, but the most important point is that any sequence of exact computations on real numbers will give the correct result, even when $\sqrt{-0}$ is involved. Also let us recall that -0 is regarded as a null value, not a negative number.

2. if x/y is an odd multiple of $1/2$ (i.e., there are two integers nearest to x/y), then n is even;

3. if $r = 0$, its sign is that of x.

A consequence of this definition is that remainders are always exactly representable, which implies that the result returned does not depend on the rounding function.

The result of x REM ∞ is x.

3.1.3.3 Preferred exponent for arithmetic operations in the decimal format

Let $Q(x)$ be the quantum exponent of a floating-point number x. Since some numbers in the decimal format have several possible representations (as mentioned in Section 3.1.1.2, the set of their representations is a *cohort*), the standard specifies for each operation which exponent is preferred for representing the result of a calculation. The rule to be followed is:

- if the result of an operation is inexact, the cohort member of smallest exponent is used;
- if the result of an operation is exact, then if the result's cohort includes a member with the preferred exponent (see below), that member is returned; otherwise, the member with the exponent closest to the preferred exponent is returned.

The preferred quantum exponents for the most common operations are:

- $x + y$ and $x - y$: $\min(Q(x), Q(y))$;
- $x \times y$: $Q(x) + Q(y)$;
- x/y: $Q(x) - Q(y)$;
- FMA(x, y, z) (i.e., $xy + z$ using an FMA): $\min(Q(x) + Q(y), Q(z))$;
- $\sqrt{x}$: $\lfloor Q(x)/2 \rfloor$.

3.1.3.4 scaleB and logB

When designing fast software for evaluating elementary functions, or for efficiently scaling variables (for instance, to write robust code for computing functions such as $\sqrt{x^2 + y^2}$), it is sometimes very useful to have functions $x \cdot \beta^n$ and $\lfloor \log_\beta |x| \rfloor$, where β is the radix of the floating-point system, n is an integer, and x is a floating-point number. This is the purpose of the functions scaleB and logB:

- scaleB(x, n) is equal to $x \cdot \beta^n$, correctly rounded[7] (following the rounding direction attribute);
- when x is finite and nonzero, logB(x) equals $\lfloor \log_\beta |x| \rfloor$. When the output format of logB is a floating-point format, logB(NaN) is NaN, logB$(\pm\infty)$ is $+\infty$, and logB(± 0) is $-\infty$.

3.1.3.5 Miscellaneous

The standard defines many useful operations, see [267]. Some examples are

- nextUp(x), which returns the smallest floating-point number in the format of x that is greater than x;
- maxNum(x, y), which returns the maximum of x and y (the next revision plans to replace the current minimum/maximum operations to solve an issue as explained in Section 3.1.7.1);
- class(x), which tells whether x is a signaling NaN, a quiet NaN, $-\infty$, a negative normal number, a negative subnormal number, -0, $+0$, a positive subnormal number, a positive normal number, or $+\infty$.

3.1.4 Comparisons

It must be possible to compare two floating-point numbers, in all formats specified by the IEEE 754-2008 standard, even if their formats differ, provided that they have the same radix. This can be done either by means of a condition code identifying one of the four mutually exclusive following conditions: *less than*, *equal*, *greater than*, and *unordered*; or as a Boolean response to a predicate that gives the desired comparison. The *unordered* condition arises when at least one of its operands is a NaN: a NaN compares unordered with everything *including itself*. A consequence of this is that the test

$$x \neq x$$

returns **true** when x is a NaN. As pointed out by Kahan [318], this provides a way of checking if a floating-point datum is a NaN in languages that lack an instruction for doing that (assuming the test is not optimized out). The other tests involving a NaN will return **false**. Hence, the test

$$x \leq y$$

is not always equivalent to the test

$$\text{not}(x > y).$$

[7] In most cases, $x \cdot \beta^n$ is exactly representable so that there is no rounding at all, but requiring correct rounding is the simplest way of defining what should be returned if the result is outside the normal range.

If at least one of the two operands is a NaN, the first test will return *false* whereas the second one will return *true*.

Also, the test $+0 = -0$ must return *true*.

Again, users and especially compiler designers should be aware of these subtleties.

As mentioned above, floating-point data represented in different formats specified by the standard must be comparable, but only *if these formats have the same radix*; the standard does not require that comparing a decimal and a binary number should be possible without a preliminary conversion. Such mixed-radix comparisons appear extremely rarely in programs written by good programmers and, at the time the standard was released, it seemed very tricky to implement them without preliminary conversion. Performing correct comparisons, however, is presumably easier than what was believed (see [73, 40]).

3.1.5 Conversions to/from string representations

Concerning conversions between an external decimal or hexadecimal character sequence and an internal binary or decimal format, the requirements of IEEE 754-2008 are much stronger than those of IEEE 754-1985. They are described as follows.

1. Conversions between an external decimal character sequence and a supported decimal format: Input and output conversions are correctly rounded (according to the applicable rounding direction).

2. Conversions between an external hexadecimal character sequence and a supported binary format: Input and output conversions are also correctly rounded (according to the applicable rounding direction). They have been specified to allow any binary number to be represented exactly by a finite character sequence.

3. Conversions between an external decimal character sequence and a supported binary format: first, for each supported binary format, define a value p_{10} as the minimum number of decimal digits in the decimal external character sequence that allows for an error-free write-read cycle, as explained in Section 4.9. Table 3.16, which gives the value of p_{10} from the various basic binary formats of the standard, is directly derived from Table 4.2.

   The standard requires that there should be an implementation-defined value H, preferably unbounded, and in any case larger than or equal to 3 plus the largest value of p_{10} for all supported binary formats, such that the conversions are correctly rounded to and from external decimal character sequences with any number of significant digits between 1

format	binary32	binary64	binary128
p_{10}	9	17	36

Table 3.16: Minimum number of decimal digits in the decimal external character sequence that allows for an error-free write-read cycle, for the various basic binary formats of the standard. See Section 4.9 for further explanation.

and H. This implies that these conversions must always be correctly rounded if H is unbounded.

For output conversions, if the external decimal format has more than H significant digits, then the binary value is correctly rounded to H decimal digits and trailing zeros are appended to fill the output format. For input conversions, if the external decimal format has more than H significant digits, then the internal binary number is obtained by first correctly rounding the value to H significant digits (according to the applicable rounding direction), then by correctly rounding the resulting decimal value to the target binary format (with the applicable rounding direction). In the directed rounding directions, these rules allow intervals to be respected (interval arithmetic is dealt with in Chapter 12).

More details are given in the standard [267].

3.1.6 Default exception handling

The IEEE 754-2008 standard supports the five exceptions already listed in Section 2.5. Let us examine them.

3.1.6.1 Invalid operation

This exception is signaled each time there is no satisfactory way of defining the numeric result of some operation. The default result of such an operation is a quiet NaN (see Section 3.1.7.1), and it is recommended that its payload contains some diagnostic information. The operations that lead to an invalid operation exception are:

- an operation on a signaling NaN (see Section 3.1.7.1), for most operations;
- a multiplication of the form $0 \times \infty$ or $\infty \times 0$;
- an FMA of the form FMA$(0, \infty, x)$ (i.e., $0 \times \infty + x$) or FMA$(\infty, 0, x)$, unless x is a quiet NaN (in that last case, whether the invalid operation exception is signaled is implementation defined);
- additions/subtractions of the form $(-\infty) + (+\infty)$ or $(+\infty) - (+\infty)$;

- FMAs that lead to the subtraction of infinities of the same sign (e.g., FMA$(+\infty, -1, +\infty)$;
- divisions of the form $0/0$ or ∞/∞;
- remainder$(x, 0)$, where x is not a NaN;
- remainder(∞, y), where y is not a NaN;
- $\sqrt{x}$ where $x < 0$;
- conversion of a floating-point number x to an integer, where x is $\pm\infty$, or a NaN, or when the result would lie outside the range of the chosen integer format;
- comparison using unordered-signaling predicates (called in the standard compareSignalingEqual, compareSignalingGreater, compareSignalingGreaterEqual, compareSignalingLess, compareSignalingLessEqual, compareSignalingNotEqual, compareSignalingNotGreater, compareSignalingLessUnordered, compareSignalingNotLess, and compareSignalingGreaterUnordered), when the operands are unordered;
- logB(x) where x is NaN or ∞;
- logB(0) when the output format of logB is an integer format (when it is a floating-point format, the value to be returned is $-\infty$).

3.1.6.2 Division by zero

The words "division by zero" are misleading, since this exception is signaled whenever an *exact* infinite result is obtained from an operation on *finite* operands. The most frequent case, of course, is the case of a division by zero, but this can also appear, e.g., when computing the logarithm of zero or the arctanh of 1. An important case is logB(0) when the output format of logB is a floating-point format. The result returned is infinity, with the correct sign.

3.1.6.3 Overflow

Let us call an *intermediate* result what would have been the rounded result if the exponent range were unbounded. The overflow exception is signaled when the absolute value of the intermediate result is finite and strictly larger than the largest finite number $\Omega = (\beta - \beta^{1-p}) \cdot \beta^{e_{\max}}$, or equivalently, when it would have an exponent strictly larger than $e_{\max}$. When an overflow occurs, the result returned depends on the rounding direction attribute:

- it will be $\pm\infty$ with the two "round-to-nearest" attributes, namely roundTiesToEven and roundTiesToAway, with the sign of the intermediate result;
- it will be $\pm\Omega$ with the roundTowardZero attribute, with the sign of the intermediate result;
- it will be $+\Omega$ for a positive intermediate result and $-\infty$ for a negative one with the roundTowardNegative attribute;
- it will be $-\Omega$ for a negative intermediate result and $+\infty$ for a positive one with the roundTowardPositive attribute.

Furthermore, the overflow flag is raised and the inexact exception is signaled.

It is important to understand three consequences of these rules:

- as we have already seen, with the two "round-to-nearest" attributes, if the absolute value of the exact result of an operation is greater than or equal to
$$\beta^{e_{\max}} \cdot \left(\beta - \frac{1}{2}\beta^{1-p}\right) = \Omega + \frac{1}{2}\,\mathrm{ulp}(\Omega),$$
then an infinite result is returned, which is not what one could expect from a naive interpretation of the words "round to nearest";
- "overflow" is *not* equivalent to "infinite result returned";
- with the roundTowardZero attribute, "overflow" is *not* equivalent to "$\pm\Omega$ is returned": if the absolute value of the exact result of some operation is larger than or equal to Ω, and strictly less than $\beta^{e_{\max}}$, then $\pm\Omega$ is returned, and yet there is no overflow.

3.1.6.4 Underflow

The underflow exception is signaled when a nonzero result whose absolute value is strictly less than $\beta^{e_{\min}}$ is computed.

- For *binary formats*, unfortunately, there is some ambiguity in the standard.[8] See Section 2.1.3 for more explanation. The underflow can be signaled either *before rounding*, that is, when the absolute value of the exact result is nonzero and strictly less than $2^{e_{\min}}$, or *after rounding*, that is, when the absolute value of a nonzero result computed as if the exponent range were unbounded is strictly less than $2^{e_{\min}}$. In rare cases, this can make a difference, for instance, when computing
$$\mathrm{FMA}\left(-2^{e_{\min}}, 2^{-p-1}, 2^{e_{\min}}\right)$$

[8] This problem was already there in the 1985 version of the standard. The choice of not giving a clearer specification in IEEE 754-2008 probably results from the desire to keep existing implementations conforming to the standard.

in rounding to nearest, an underflow will be signaled if this is done before rounding, but not if it is done after rounding.

- For *decimal formats*, there is no ambiguity and the underflow result is signaled *before rounding*, i.e., when the absolute value of the exact result is nonzero and strictly less than $10^{e_{\min}}$.

The result is always correctly rounded: the choice (in the binary case) of how the underflow is detected (that is, before or after rounding) has no influence on the result delivered.

In case of underflow, if the result is inexact, then the underflow flag is raised and the inexact exception is signaled. If the result is exact, then the underflow flag is *not* raised. This might sound strange, but this was an adroit choice of the IEEE working group: the major use of the underflow flag is for warning that the result of some operation might not be very accurate—in terms of relative error. Thus, raising it when the operation is *exact* would be a needless warning. *This should not be thought of as an extremely rare event*: indeed, Theorem 4.2 shows that with any of the two round-to-nearest rounding direction attributes, whenever an addition or subtraction underflows, it is performed exactly.

3.1.6.5 Inexact

If the result of an operation differs from the exact result, then the inexact exception is signaled. The correctly rounded result is returned.

3.1.7 Special values

3.1.7.1 NaN: Not a Number

The standard defines two types of NaNs:

- *signaling NaNs* (sNaNs) do not appear, in default mode, as the result of arithmetic operations. Except for the sign-bit operations (such as copy, negate, and absolute value), decimal re-encoding operations, and some conversions, they signal the invalid operation exception whenever they appear as operands. For instance, they can be used for uninitialized variables;

- *quiet NaNs* (qNaNs) propagate through almost all operations[9] without signaling exceptions. They can be used for debugging and diagnostic purposes. As stated above, for operations that deliver a floating-point result, the default exception handling is that a quiet NaN is returned

[9] There are a few exceptions to this rule (but not with the basic arithmetic operations): for instance it is recommended that $\mathrm{pow}(+1, y)$ should be 1 even if y is a quiet NaN (this partial function being constant).

whenever an invalid operation exception occurs; this rule is valid for all operations of IEEE 754-2008, but the new revision plans to introduce new operations that will not follow it, as explained below.

For example, qNaN $\times$ 8, sNaN + 5, and $\sqrt{-2}$ all give qNaN.

An issue with the various rules on NaNs is that from a signaling NaN, one may get a non-NaN result. Thus users who wish to detect the use of uninitialized variables via signaling NaNs should test the invalid operation exception instead of the result. Moreover, these rules make the current operations that return the minimum or maximum of two data[10] non-associative, which is an issue in some contexts, such as parallel processing; for instance:

- $\mathrm{minNum}(1, \mathrm{minNum}(1, \mathrm{sNaN})) \rightarrow \mathrm{minNum}(1, \mathrm{qNaN}) \rightarrow 1$;
- $\mathrm{minNum}(\mathrm{minNum}(1, 1), \mathrm{sNaN}) \rightarrow \mathrm{minNum}(1, \mathrm{sNaN}) \rightarrow \mathrm{qNaN}$.

The next revision plans to replace these operations by ones that will be associative, thus with specific rules.[11]

We have seen in Section 3.1.1 that in the binary interchange formats, the least significant $p - 2$ bits of a NaN are not defined, and in the decimal interchange formats, the trailing significand bits of a NaN are not defined. These bits can be used for encoding the *payload* of the NaN, i.e., some information that can be transmitted through the arithmetic operation for diagnostic purposes. To preserve this diagnostic information, for an operation with quiet NaN inputs and a quiet NaN result, the returned result should be one of these input NaNs.

3.1.7.2 Arithmetic of infinities and zeros

The arithmetic of infinities and zeros follows the intuitive rules. For instance, $-1/(-0) = +\infty$, $-5/(+\infty) = -0$, $\sqrt{+\infty} = +\infty$ (the only somewhat counter intuitive property is $\sqrt{-0} = -0$). This very frequently allows one to get sensible results even when an underflow or an overflow has occurred. And yet, one should be cautious. Consider for instance, assuming round-to-nearest (with any choice in case of a tie), the computation of

$$f(x) = \frac{x}{\sqrt{1 + x^2}},$$

for $\sqrt{\Omega} < x \leq \Omega$, where Ω is the largest finite floating-point number. The computation of x^2 will return an infinite result; hence, the computed value of $\sqrt{1 + x^2}$ will be $+\infty$. Since x is finite, by dividing it by an infinite value we

[10] These operations regard qNaN as missing data, but not sNaN.

[11] This means that in the case where a NaN is not regarded as missing data, qNaN must propagate even when the partial function is constant. For instance, the minimum function on (qNaN,$-\infty$) will return qNaN instead of $-\infty$.

will get $+0$. Therefore, the computed value of $f(x)$, for x large enough, will be $+0$, whereas the exact value of $f(x)$ is extremely close to 1. This shows, for critical applications, the need either to prove in advance that an overflow/underflow cannot happen, or to check, by testing the appropriate flags, that an overflow/underflow has not happened.

3.1.8 Recommended functions

The standard recommends (but does not require) that the following functions should be correctly rounded: e^x, $e^x - 1$, 2^x, $2^x - 1$, 10^x, $10^x - 1$, $\ln(x)$, $\log_2(x)$, $\log_{10}(x)$, $\ln(1+x)$, $\log_2(1+x)$, $\log_{10}(1+x)$, $\sqrt{x^2+y^2}$, $1/\sqrt{x}$, $(1+x)^n$, x^n, $x^{1/n}$ (n is an integer), $\sin(\pi x)$, $\cos(\pi x)$, $\arctan(x)/\pi$, $\arctan(y/x)/\pi$, $\sin(x)$, $\cos(x)$, $\tan(x)$, $\arcsin(x)$, $\arccos(x)$, $\arctan(x)$, $\arctan(y/x)$, $\sinh(x)$, $\cosh(x)$, $\tanh(x)$, $\sinh^{-1}(x)$, $\cosh^{-1}(x)$, $\tanh^{-1}(x)$ (note that the power function x^y is not in this list). This was not the case with the previous version (754-1985) of the standard.

See Section 10.5 for an introduction to the various issues linked with the correct rounding of transcendental functions.

3.2 On the Possible Hidden Use of a Higher Internal Precision

The processor being used may offer an internal precision that is wider than the precision of the variables of a program (a typical example is the double-extended format available on Intel platforms when we use the x87 instruction set, when the variables of the program are binary32 or binary64 floating-point numbers). This may sometimes have strange side effects, as we will see in this section.

Consider the C program (Program 3.1).

```
#include <stdio.h>

int main(void)
{
  double a = 1848874847.0;
  double b = 19954562207.0;
  double c;
  c = a * b;
  printf("c = %20.19e\n", c);
  return 0;
}
```

Program 3.1: A C program that might induce a double rounding.

Table 3.17 gives some results returned by this program, depending

on the compilation options. In order to really test the arithmetic of the machine, it is important that the compiler does not optimize the multiplication by performing it at compile time (one controls even less what occurs at compile time); by default, GCC does not do such an optimization. Note that the double-precision number closest to the exact product is `3.689348814741911424e+19`.

Switches on the GCC command line	Output
no switch (default)	c = 3.689348814741911424e+19
-mfpmath=387	c = 3.689348814741910323e+19

Table 3.17: Results returned by Program 3.1 on a Linux/Debian 64-bit Intel Xeon CPU platform, with GCC 7.2.0, depending on the compilation options. On a 64-bit platform, the default is to use the SSE registers.

What happened? The exact value of `a*b` is 36893488147419107329, whose binary representation is

$$\overbrace{\underbrace{10000000000000000000000000000000000000000000000000}_{53\text{ bits}}10000000000}^{64\text{ bits}}01$$

On the processor used, with the `-mfpmath=387` switch, the product is first rounded to the precision of the `x87` registers (namely, "double-extended" precision), which gives (in binary)

$$\overbrace{\underbrace{10000000000000000000000000000000000000000000000000}_{53\text{ bits}}10000000000}^{64\text{ bits}} \times 4$$

Then, that intermediate value is rounded to the binary64 destination format, which gives (using the round-to-nearest-even rounding mode)

$$\underbrace{10000000000000000000000000000000000000000000000000}_{53\text{ bits}} \times 2^{13}$$
$$= 36893488147419103232_{10},$$

whereas the product `a*b` correctly rounded to the nearest binary64 number is

$$\underbrace{10000000000000000000000000000000000000000000000001}_{53\text{ bits}} \times 2^{13}$$
$$= 36893488147419111424_{10}.$$

In general, by default,[12] the product is directly stored in the 64-bit Streaming SIMD Extension (SSE) registers. In that case, it is directly rounded to binary64, so that we get the expected result.

The problem we have faced here is called "double rounding." In this example, it appears during a multiplication, but it may also appear during another arithmetic operation. Another example (still with binary64 input values) is the addition of

$$9223372036854775808.0 = 2^{63}$$

and

$$1024.25.$$

Such examples are not so rare that they can be neglected. Assuming binary64 variables and an Intel "double-extended" internal format, if the chosen compilation switches (or the default mode) do not prevent the problem from occurring, the double rounding problem occurs when the binary expansion of the exact result of some operation is of the form

$$2^k \times \overbrace{1.xxxxx\cdots xx0}^{53\text{ bits}}\ \overbrace{10000000000}^{11\text{ bits}}\ 0\ \overbrace{xxxxxxxxxxxxxxxxxxxxx\cdots}^{\text{at least one 1 somewhere}}$$

or

$$2^k \times \overbrace{1.xxxxx\cdots xx1}^{53\text{ bits}}\ \overbrace{01111111111}^{11\text{ bits}}\ 1\ \overbrace{xxxxxxxxxxxxxxxxxxxxx\cdots}^{\text{at least one 0 somewhere}}.$$

Assuming equal probabilities of occurrence for the zeros and ones in the binary expansion of the result of an arithmetic operation,[13] the probability of a double rounding is $2^{-12} = 1/4096$, which means that without care with the compilation options, double roundings will occur in any computation of significant size. We must emphasize that this might be a problem with certain very specific algorithms (such as those presented in Chapter 4, see [409] for a discussion on that topic), but with most calculations, it will be unnoticed.

The possible, sometimes hidden, use of a larger internal precision may also lead to a management of overflow and underflow that is sometimes difficult to predict. Consider Program 3.2, due to Monniaux [423].

Compiled with GCC 4.9.2 under Linux on an Intel(R) Xeon(R) CPU E5-2695 v2 processor, the program returns $+\infty$. If we add the `-mfpmath=387` command line option, we get `1e+308`. What happened? Although in binary64 arithmetic, with the round-to-nearest ties-to-even (i.e., the default) rounding function, the multiplication `v * v` should return $+\infty$, in the program compiled with the `-mfpmath=387` option, the implicit variable representing this

[12] But it is the default on recent systems only.

[13] Which is not very realistic but suffices to get a rough estimate of the frequency of occurrences of double roundings.

```
#include <stdio.h>
int main(void)
{
  double v = 1E308;
  double x = (v * v) / v;
  printf("%g\n",x);
  return 0;
}
```

Program 3.2: This example is due to David Monniaux [423]. Compiled with GCC under Linux, we get `1e+308` *with the command line option* `-mfpmath=387` *and* $+\infty$ *without it.*

product was actually stored in a "double-extended" precision register of the `x87` instruction set of the processor. And since the product `v * v` is much below the overflow threshold in double-extended precision, the stored value was not $+\infty$, but the double-extended number closest to the exact product.

It is important to note that, in this case, the result obtained is *very accurate*, which is not so surprising: in most cases, using a larger internal precision for intermediate calculations leads to better calculations. What matters then is not to forbid the behavior, but to allow programmers to decide if they want all intermediate calculations to be performed in the format of the operands or in a format they clearly specify (which enhances portability and provability), or if they prefer these intermediate calculations to be performed in a wider format whenever available (typically, the largest format available in hardware, which in general improves the accuracy of the results). A tradeoff is to be found between portability, accuracy, and (frequently) speed. Choosing which among these criteria is the most important should be the programmer's task, not the compiler's. This is a reason for the introduction of the preferred width attributes in IEEE 754-2008 (see Section 3.1.2.3).

3.3 Revision of the IEEE 754-2008 Standard

At the time we are writing this book, the IEEE 754-2008 standard is under revision. The next version should be released in 2018. The goal of the revision committee is mainly to deal with the errata and improve clarity; hence the standard will be almost unchanged. Since the next version is not yet adopted, we cannot be definite about its contents. However, the main changes are expected to be:

- the replacement of the current Min/Max operations by new ones (see Section 3.1.7.1);
- recommended "augmented" addition and multiplication operations, which would more or less return the same results as the 2Sum and

Fast2Sum algorithms (described in Chapter 4): the sum or product x of two floating-point numbers would be expressed as a "double word," that is, a pair of floating-point numbers x_h and x_ℓ such that $x_h = \mathrm{RN}(x)$ and $x = x_h + x_\ell$. It is likely that in these operations, the RN function will be round to nearest ties-to-zero.

3.4 Floating-Point Hardware in Current Processors

Virtually all recent computers are able to support the IEEE 754 standard efficiently through a combination of hardware and software. This section reviews the hardware that can be found in mainstream systems. It should be understood that the line between hardware and software support is drawn by hardware vendors depending on the demand of the target market. To take just a few examples,

- IBM supports decimal floating-point in hardware [184, 548] (with the decimal encoding), while Intel provides well-tuned software implementations [117] (with the binary encoding). These encodings are presented in Section 3.1.1.2.
- Elementary functions are usually implemented in software, but some GPUs offer a limited set of hardware implementations [470].
- For the basic operations, the standard mandates rounding *to any supported format, from any combination of operand formats*. However, hardware usually only supports *homogeneous* operations, i.e., operations that have the same input and output format. The mixed-format operations are considered rare enough to be implemented in software only [400].
- One notable exception to the previous statement is the mixed-format FMA of the Kalray processors, where the addend and the result are binary64 while the two multiplicands are binary32 [83, 84]. Binary16/binary32 mixed-format operations are also appearing in GPUs. These are discussed in more detail in Section 7.8.2.

3.4.1 The common hardware denominator

Current processors for desktop computers offer hardware binary64 operators for floating-point addition, subtraction, and multiplication, and at least hardware assistance for division and square root. Peak performance is typically between 2 and 8 binary64 floating-point operations per clock cycle for $+$, $-$, and $\times$, with much slower division and square root [469, 271].

However, most processors go beyond this common denominator and offer larger precision and/or faster operators. The following sections detail these extensions.

3.4.2 Fused multiply-add

The *fused multiply-add* (FMA) instruction evaluates an expression of the form $a \times b + c$ with one rounding only, that is, if $\circ$ is the rounding function, the returned result is $\circ(a \times b + c)$ (see Section 2.4). The FMA has been the main floating-point operation in the most recent instruction sets (IBM POWER/PowerPC, HP/Intel IA-64, Kalray MPPA). It has also been added to older instruction sets:

- Intel IA-32 with AMD's SSE5 and Intel's AVX extensions, as detailed below;
- ARM since ARMv7;
- SPARC with HAL/Fujitsu SPARC64 VI[14];
- MIPS with the Loongson processor family.

These operations are all compatible with the FMA defined by IEEE 754-2008. As far as this operator is concerned, IEEE 754-2008 standardized already existing practice.

In terms of instruction set, the FMA comes in two variants:

- an FMA4 instruction specifies 4 registers a, b, c, and d, and stores in d the result $\circ(a \cdot b + c)$.
- an FMA3 instruction specifies only 3 registers: the destination d has to be one of the input registers, which is thus overwritten.

In each instruction set, there is actually a family of instructions that includes useful variations such as fused multiply-subtract.

FMA latency. In general, the FMA is pipelined and has a latency slightly larger than that of a floating-point addition or multiplication alone. For illustration, the FMA latency was 4 cycles in Itanium2 and 7 cycles on Power6. In a recent ARM implementation [399], the 7-cycle FMA is replaced with an 8-cycle one, built by connecting an adder and a multiplier (suitably modified). This enables additions and multiplications in 4 cycles instead of 7.

The chaotic introduction of the FMA in the IA-32-compatible processors. In 2007, AMD was the first to announce FMA support with its SSE5 extension [2]. Intel followed in 2008 by announcing an FMA4. Later this year, it switched to FMA3 with a different extension: AVX. Meanwhile, AMD

[14] Warning! The instructions called `FMADDs` and so on from SPARC64 V, which share the same name and the same encoding with SPARC64 VI, are *not* real FMA instructions as they perform two roundings. [208, page 56].

switched to an FMA4 compatible with Intel's first announce, and its first Bulldozer processors in 2005 supported this FMA4. Later processors introduced FMA3 instructions compatible with Intel's. Intel's Haswell was its first processor supporting an FMA3.

As of 2017, the situation has settled, and processors from both vendors support the same FMA3 instructions in the AVX instruction set extension.

3.4.3 Extended precision and 128-bit formats

As we have already seen in Sections 3.1.1 and 3.2, the legacy x87 instructions of the IA-32 instruction set can operate on a *double-extended* precision format with 64 bits of significand and 15 bits of exponent. The corresponding floating-point operators can be instructed to round to single, double, or double-extended precision.

The IA-64 instruction set also defines several double-extended formats, including a 80-bit format compatible with IA-32 and a 82-bit format with a 64-bit significand and a 17-bit exponent. The two additional exponent bits are designed to avoid intermediate "spurious" overflows in certain computations on 80-bit operands (a typical example is the evaluation of $\sqrt{a^2+b^2}$).

Some instruction sets (SPARC, POWER, z/Architecture) have instructions operating on binary128 data. As far as we know, however, the internal data-paths are still optimized for 64-bit operations, and these instructions require several passes over 64-bit operators.

3.4.4 Rounding and precision control

Traditionally, the rounding precision (e.g., binary32, binary64, double-extended if available) and the rounding direction attributes used to be specified via a global status/control register. In IA-32, this was called `FPSR` for *Floating-Point Status Register*, or `MXCSR` for SSE/AVX extensions. Such a global register defines the behavior of the floating-point instructions.

However, recent processors are designed to execute, at a given time, many floating-point instructions launched out of order from several hardware threads. Keeping track of changes to the control/status word would be very costly in this context. Therefore, the prevailing choice in the recent years has been to ensure that all the instructions in flight share the same value of the status/control word. To achieve this, any instruction that changes the value of the control word must first wait until all in-flight instructions have completed execution.

In other words, before launching any new floating-point instruction with a new value of the control register, all current floating-point instructions have to terminate with the previous value. This can stall the processor for tens of cycles.

Unfortunately, some applications, such as interval arithmetic (see [428] and also Section 12.3.2), need frequent rounding direction changes. This performance issue could not be anticipated in 1985, when processor architectures were not yet pipelined. It also affects most processor instruction sets designed in the 1980s and 1990s.

To address this performance issue, more recent instruction sets make it possible to change the rounding direction attribute on a per-instruction basis without any performance penalty. Technically, the rounding direction attribute is defined in the instruction word, not in a global control register. This feature was for instance introduced in the HP/Intel IA-64 [118] instruction set and the Sun Microsystems' VIS extension to the SPARC instructions set [579]. It was introduced in the IA-32 instruction set by the AVX-512 extension[15] [272, Sec. 15.6.4]. Note that in all these cases, it is still possible to read the rounding precision and rounding direction from a global control/status register. In highly parallel processors such as Graphics Processing Units, rounding is also typically defined on a per-instruction basis.

The rounding direction specification in the IEEE 754-1985 standard (and hence in the language standards that were later on adapted to implement it) reflected the notion of a global status word. This meant in practice that per-instruction rounding specification could not be accessed from current high-level languages in a standard, portable way.

The IEEE 754-2008 standard corrected this, but the change will take time to percolate in programming languages and compilers. This issue will be addressed in more detail in Chapter 6, Languages and Compilers.

3.4.5 SIMD instructions

Most recent instruction sets also offer single instruction, multiple data (SIMD) instructions. An SIMD instruction applies the same operation to a vector of data, producing a vector of results. These vectors are kept in wide registers, for instance 128- to 512-bit registers in the IA-32 instruction set extensions, and up to 2048 bits for the ARM Scalable Vector Extension (SVE).

Such wide registers can store vectors of 8-bit, 16-bit, 32-bit, or 64-bit integers. SIMD instructions operating on vectors of small integers are often referred to as *multimedia instructions*. Indeed, in image processing, the color of a pixel may be defined by three 8-bit integers giving the intensity of the red, green, and blue components; in audio processing, sound samples are commonly digitized on 16 bits.

A wide register can also be considered as a vector of 16-bit, 32-bit, or 64-bit floating-point numbers.

[15] These instructions also suppress ("silent") exceptions, which induce similar problems in case of out-of-order execution (although their correct management in hardware must be implemented anyway).

Examples of vector instruction sets include AltiVec for the POWER/PowerPC family, and for the IA-32 instruction set, 3DNow!/MMX (64-bit vector), then SSE to SSE5 (128-bit vector), then AVX/AVX2 (256-bit vector) and AVX-512 (512-bit vector).

Each of these extensions comes with too many new instructions to be detailed here: not only arithmetic operations, but also data movement inside a vector, and complex operations such as scalar products, or sums of absolute values of differences.

At the time of writing this book, all new IA-32-compatible processors for workstation and personal computers implement the AVX extension (some low-power chips such as Intel's Atom are still limited to SSE3). AVX defines sixteen 256-bit registers, each of which can be considered either as a vector of eight binary32 numbers, or as a vector of four binary64 numbers. AVX instructions include the FMA and are fully IEEE 754-2008 compliant.[16]

3.4.6 Binary16 (half-precision) support

The 16-bit formats were introduced for graphics and gaming; the binary16 format of IEEE 754-2008 actually standardized existing practice. Although it was not considered as a basic format by the IEEE 754-2008 standard, binary16 is increasingly being used for computing. This is true in graphics, but also in the field of machine learning. In particular, it has been shown that certain convolution neural networks (CNN) can work with precisions as low as 8 bits. This justifies the use of the binary16 format, which provides a 11-bit significand.

This is leading to extensive binary16 arithmetic support in newer instruction sets such as ARMv8, Kalray Coolidge, and NVIDIA Volta.

3.4.7 Decimal arithmetic

At the time of writing this book, only high-end processors from IBM (POWER and zSeries) include hardware decimal support, using the binary encoding.

The latency of decimal operations is much larger than that of binary operations. Moreover, it depends on the operand values [184, 548], as illustrated by Table 3.18. This is due to several factors. The encoding itself is more complex (see Section 3.1.1.2). Since numbers are not necessarily normalized, significand alignment and rounding rules are also much more complex than in the binary case (see for instance the preferred exponent rules Section 3.1.3.3). Finally, the core of decimal units in current IBM processors is a pipelined 36-digit adder [548], and multiplication and division must iterate over it.

[16] The performance of AVX instructions can be degraded on some computations involving subnormals.

Cycles	decimal64 operands	decimal128 operands
addition/subtraction	12 to 28	16 to 31
multiplication	16 to 55	17 to 104
division	16 to 119	17 to 193

Table 3.18: Execution times in cycles of decimal operations on the IBM z10, from [548].

3.4.8 The legacy x87 processor

The Intel 8087 co-processor was a remarkable achievement when it was conceived. More than thirty years later, the floating-point instructions it defined are still available in IA-32. However, they are showing their age.

- There are only 8 floating-point registers, and their organization as a stack leads to data movement inefficiencies.
- The hidden use of extended precision entails a risk of double rounding (see Section 3.2).
- The dynamic rounding precision can introduce bugs in modern software, which is almost always made up of several components (dynamic libraries, plug-ins). For instance, the following bug in Mozilla's Javascript engine was discovered in 2006: if the rounding precision was reduced to single precision by a plug-in, then the `js_dtoa` function (double-to-string conversion) could overwrite memory, making the application behave erratically, e.g., crash. The cause was the loop exit condition being always false due to an unexpected floating-point error.[17]
- The x87 FPSR register defines the rounding precision (the significand size) but not the exponent size, which is always 15 bits. Even when instructed to round to single precision, the floating-point unit will signal overflows or underflows only for numbers out of the *double-extended* exponent range. True binary32 or binary64 overflow/underflow detection is performed only when writing the content of a floating-point register to memory. This two-step overflow/underflow detection can lead to subtle software problems, just like double rounding. It may be avoided only by writing all the results to memory, unless the compiler can prove in advance that there will be no overflows.

The SSE and AVX instructions were designed more recently. As they do not offer extended precision, they may result in less accurate results than the legacy x87 instructions. However, in addition to their obvious performance

[17] CVE-2006-6499 / https://bugzilla.mozilla.org/show_bug.cgi?id=358569.

advantage due to SIMD execution, they are fully IEEE 754-1985 (SSE) and IEEE 754-2008 (AVX) compliant. They permit better reproducibility (thanks to the static rounding precision) and portability with other platforms.

All these reasons push towards deprecating the use of the venerable x87 instructions. For instance, GCC uses SSE2 instructions by default on all 64-bit variants of GNU/Linux (since 64-bit capable processors all offer the SSE2 extension). The legacy x87 unit is still available to provide higher precision for platform-specific kernels that may require it, for instance some elementary function implementations.

3.5 Floating-Point Hardware in Recent Graphics Processing Units

Graphics processing units (GPUs), initially highly specialized for integer computations, quickly evolved towards more and more programmability and increasingly powerful arithmetic capabilities.

Binary floating-point units appeared in 2002-2003 in the GPUs of the two main vendors, ATI (with a 24-bit format in the R300 series) and NVIDIA (with a 32-bit format in the NV30 series). In both implementations, addition and multiplication were incorrectly rounded: according to a study by Collange et al. [109], instead of rounding the exact sum or product, these implementations typically rounded a $p + 2$-bit result to the output precision of p bits.

Still, these units fueled interest in GPUs for general-purpose computing (GPGPU), as the theoretical floating-point performance of a GPU is up to two orders of magnitude times that of a conventional processor (at least in binary32 arithmetic). At the same time, programmability was also improved, notably to follow the evolution to version 10 of Microsoft's DirectX application programming interface. Specific development environments also appeared: first NVIDIA's C-based CUDA, soon followed by the Khronos Group's OpenCL.

Between 2007 and 2009, both ATI (now AMD) and NVIDIA introduced new GPU architectures with, among other things, improved floating-point support.

Currently, most GPUs support the binary32 and binary64 formats, with correct rounding in the four legacy rounding modes for basic operations and FMA, with subnormals [631]. It is worth mentioning that they also include hardware acceleration of some elementary functions [470]. The latest notable evolution of floating-point in GPUs has been the introduction of half precision (binary16) arithmetic to accelerate machine learning applications. Subnormals are fully supported in binary16.

The remaining differences with mainstream microprocessors, in terms of floating-point arithmetic, are mostly due to the highly parallel execution model of GPUs.

- Current GPUs do not raise floating-point exceptions, and there are no internal status flags to check for them.

- The rounding mode is part of the instruction opcode (thus statically determined at compile time) rather than stored in a global status register.

3.6 IEEE Support in Programming Languages

The IEEE 754-1985 standard was targeted mainly at processor vendors and did not focus on programming languages. In particular, it did not define bindings (i.e., how the IEEE 754 standard is to be implemented in the language), such as the mapping between native types of the language and the formats of IEEE 754 and the mapping between operators/functions of the language and the operations defined by IEEE 754. The IEEE 754-1985 standard did not even deal with what a language should specify or what a compiler is allowed to do. This has led to many misinterpretations, with users often thinking that the processor will do exactly what they have written in the programming language. Chapter 6 will survey in more detail floating-point issues in mainstream programming languages.

For instance, it is commonly believed that the `double` type of the ISO C language must correspond everywhere to the binary64 binary format of the IEEE 754 standard, but this property is "implementation-defined," and behaving differently is not a bug. Indeed the *destination* (as defined by the IEEE 754 standard) does not necessarily correspond to the C floating-point type associated with the value. This is the reason why an implementation using double-extended is valid.

The consequences are that one can get different results on different platforms. But even when dealing with a single platform, one can also get unintuitive results, as shown in Goldberg's article with the appendix *Differences Among IEEE 754 Implementations* [214] or in Chapter 6 of this book.

The bottom line is that the reader should be aware that a language will not necessarily follow standards as he or she might expect. Implementing the algorithms given in this book may require special care in some environments (languages, compilers, platforms, and so on). This book (in Chapter 6) will give some examples, but with no claim of exhaustiveness.

The IEEE 754-2008 standard clearly improves the situation, mainly in its clauses 10 (*Expression evaluation*) and 11 (*Reproducible floating-point results*) – see Section 3.1.2.5. For instance, it deals with the double-rounding problem (observed on x87, described in Section 3.4.8): "Language standards should disallow, or provide warnings for, mixed-format operations that would cause implicit conversion that might change operand values." However, it may take time until these improvements percolate from the IEEE 754-2008 standard into languages and their compilers.

3.7 Checking the Environment

Checking a floating-point environment (for instance, to make sure that a compiler optimization option is compliant with one of the IEEE standards) may be important for critical applications. Circuit or software manufacturers frequently use formal proofs to ensure that their arithmetic algorithms are correct [426, 534, 535, 536, 241, 242, 117, 6]. Also, when the algorithms used by some environment are known, it is possible to design test vectors that allow one to explore every possible branching. Typical examples are methods for making sure that every element of the table of a digit-recurrence division or square root algorithm [186] is checked. For some functions, one can find "hardest-to-round" values that may constitute good input values for tests. This can be done using Hensel lifting for multiplication, division and square root [484], or using some techniques briefly presented in Chapter 10 for the transcendental functions.

Checking the environment is more difficult for the end user, who generally does not have any access to the algorithms that have been used. When we check some environment as a "black box" (that is, without knowing the code, or the algorithms used in the circuits) there is no way of being absolutely sure that the environment will *always* follow the standards. Just imagine a buggy multiplier that always returns the right result but for one pair of input operands. The only way of detecting this would be to check all possible inputs, which would be extremely expensive in the binary32 format, and totally impossible in the binary64, decimal64, binary128 or decimal128 formats. This is not pure speculation: in binary32/single precision arithmetic, the divider of the first version of the Pentium circuit would produce an incorrect quotient with probability around 2.5×10^{-11} [108, 437, 183], assuming random inputs.

Since the early 1980s, various programs have been designed for determining the basic parameters of a floating-point environment and assessing its quality. We present some of them below. Most of these programs are merely of historical importance: thanks to the standardization of floating-point arithmetic, the parameters of the various widely available floating-point environments are not so different.

3.7.1 MACHAR

MACHAR was a program, written in FORTRAN by W. Cody [106], whose purpose was to determine the main parameters of a floating-point format (radix, "machine epsilon," etc.). This was done using algorithms similar to the one we give in Section 4.1.1 for finding the radix β of the system. Today, it is interesting for historical purposes only.

In their book on elementary functions, Cody and Waite [107] also gave methods for estimating the quality of an elementary function library. Their methods were based on mathematical identities such as

$$\sin(3x) = 3\sin(x) - 4\sin^3(x). \tag{3.2}$$

These methods were useful at the time they were published. And yet, they can no longer be used with current libraries. Recent libraries are either correctly rounded or have a maximal error very close to $\frac{1}{2}\,\mathrm{ulp}$. Hence, *they are far more accurate than the methods that are supposed to check them.*

3.7.2 Paranoia

Paranoia [329] is a program originally written in Basic by Kahan, and translated to Pascal by B.A. Wichmann and to C by Sumner and Gay in the 1980s, to check the behavior of floating-point systems. It finds the main properties of a floating-point system (such as its precision and its exponent range), and checks if underflow is gradual, if the arithmetic operations are properly implemented, etc. It can be obtained at http://www.netlib.org/paranoia/.

Today, Paranoia is essentially of historical importance. It can be useful as a debugging tool for someone who develops his or her own floating-point environment.

3.7.3 UCBTest

UCBTest can be obtained at http://www.netlib.org/fp/ucbtest.tgz. It is a collection of programs whose purpose is to test certain difficult cases of the IEEE floating-point arithmetic. Paranoia is included in UCBTest. The "difficult cases" for multiplication, division, and square root (i.e., almost hardest-to-round cases: input values for which the result of the operation is very near a breakpoint of the rounding mode) are constructed using algorithms designed by Kahan, such as those presented in [484].

3.7.4 TestFloat

J. Hauser designed an excellent software implementation of the IEEE 754 floating-point arithmetic. The package is named SoftFloat and can be downloaded at http://www.jhauser.us/arithmetic/SoftFloat.html (at the time of writing these lines, the most recent version was released in August 2017). He also designed a program, TestFloat, aimed at testing whether a system conforms to IEEE 754. TestFloat compares results returned by the system to results returned by SoftFloat. It is accessible from the SoftFloat page.

3.7.5 Miscellaneous

SRTEST is a FORTRAN program written by Kahan for checking implementation of SRT [186, 187] division algorithms. It can be accessed on Kahan's web page, at https://people.eecs.berkeley.edu/~wkahan/srtest/. Some useful software, written by Beebe, can be found at http://www.math.utah.edu/~beebe/software/ieee/. MPCHECK is a program written by Revol, Pélissier, and Zimmermann. It checks mathematical function libraries (for correct rounding, monotonicity, symmetry, and output range). It can be downloaded at https://members.loria.fr/PZimmermann/mpcheck/.