Floating-Point Data Formats

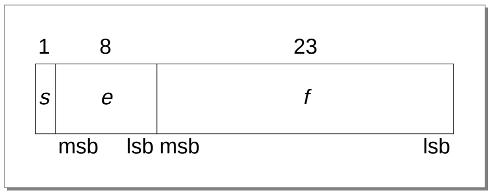
This chapter describes the data formats your PowerPC application can use to represent floating-point numbers. It begins by discussing in general the methods PowerPC Numerics uses to store and interpret floating-point values and by explaining why those methods were chosen. The chapter introduces the special values zero, NaN (Not-a-Number), and Infinity and explains why these special values are necessary. Next is an in-depth description of the numeric data formats with a discussion of how these formats represent floating-point values. At the end of the chapter, you will find a table comparing the size, range, and precision of the numeric data formats. This table can help you choose which data format is best for your application.

You should read this chapter to learn about the floating-point data formats available on PowerPC processor-based Macintosh computers and to learn more about how your computer encodes and manipulates floating-point numbers.

About Floating-Point Data Formats

The IEEE standard defines several floating-point data formats, one required and the others recommended. IEEE requires that each data format have a **sign bit** (*s*), an **exponent** field (*e*), and a **fraction** field (*f*). For each format, it lists requirements for the minimum lengths of these fields. For example, the standard describes a 32-bit single format whose exponent field must be 8 bits long and whose fraction field must be 23 bits long. Figure 2-1 shows the IEEE requirements for the single format. (In this figure, *msb* stands for *most significant bit* and *lsb* stands for *least significant bit.*)

Figure 2-1 IEEE single format



The only required data format is the 32-bit single format. A 64-bit double format is strongly recommended. The IEEE standard also describes two data formats called *single-extended* and *double-extended* and recommends that floating-point environments provide the extended format corresponding to the widest basic format (single or double) they support.

To conform to the IEEE requirements on floating-point data formats, the PowerPC Numerics environment provides three data formats: single (32 bits), double (64 bits), and double-double (128 bits). The single and double formats are implemented exactly as described in the standard. The double-double format is provided in place of the recommended double-extended format. IEEE requires that the double-extended format be at least 79 bits long with at least a 15-bit exponent. The double-double format is

128 bits long and has an 11-bit exponent. The double-double format is just what its name sounds like: two double-format numbers combined. The PowerPC assembly-language multiply-add instructions, which multiply two double-format numbers and add a third with at most one roundoff error, make implementing the double-double format much more efficient than implementing a true IEEE double-extended format. See Chapter 14, “Assembly-Language Numeric Operations,” for more information on the multiply-add instructions.

Table 2-1 shows how the three numeric data formats correspond to C variable types. For more information about data types in C, refer to Chapter 7, “Numeric Data Types in C.”

Table 2-1 Names of data types

PowerPC Numerics data format	C type
IEEE single	`float`
IEEE double	`double`
Double-double	`long double`

The IEEE standard also makes requirements about how the values in these data formats are interpreted. PowerPC Numerics follows these requirements exactly. They are described in the next section.

Interpreting Floating-Point Values

Regardless of which data format (single, double, or double-double) you use, the numerics environment uses the same basic method to interpret which floating-point value the data format represents. This section describes that method.

Every floating-point data format has a sign bit, an exponent field, and a fraction field. These three fields provide binary encodings of a sign (+ or –), an exponent, and a **significand,** respectively, of a floating-point value. The value is interpreted as

$$\pm\ \mathit{significand} \times 2^{exponent - bias}$$

where

$\pm$	is the sign stored in the sign bit (1 is negative, 0 is positive).
significand	has the form $b_0 .b_1 b_2 b_3 \ldots b_{precision-1}$ where $b_1 b_2 b_3 \ldots b_{precision-1}$ are the bits in the fraction field and b_0 is an implicit bit whose value is interpreted as described in the sections “Normalized Numbers” and “Denormalized Numbers.” The significand is sometimes called the *mantissa*.

exponent is the value of the exponent field.

bias is the bias of the exponent. The **bias** is a predefined value (127 for single format, 1023 for double and double-double formats) that is added to the exponent when it is stored in the exponent field. When the floating-point number is evaluated, the bias is subtracted to return the correct exponent. The minimum biased exponent field (all 0's) and maximum biased exponent field (all 1's) are assigned special floating-point values (described in the next several sections).

In a numeric data format, each valid representation belongs to exactly one of these classes, which are described in the sections that follow:

- normalized numbers
- denormalized numbers
- Infinities
- NaNs (signaling or quiet)
- zeros

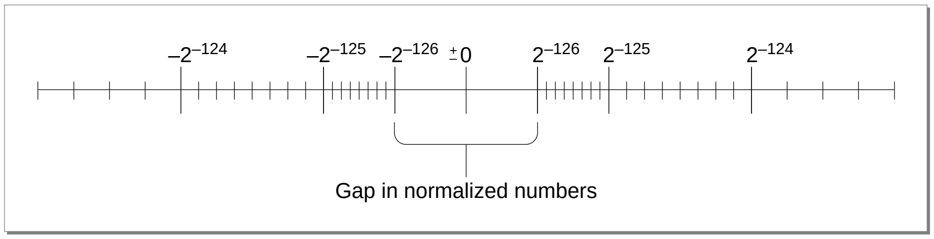
Normalized Numbers

The numeric data formats represent most floating-point numbers as **normalized numbers,** meaning that the implicit leading bit (b_0 on page 2-4) of the significand is 1. Normalization maximizes the resolution of the data type and ensures that representations are unique. Figure 2-2 shows the magnitudes of normalized numbers in single precision on the number line. The spacing of the vertical marks indicates the relative density of numbers in each binade. (A **binade** is a collection of numbers between two successive powers of 2.) Notice that the numbers get more dense as they approach 0.

Note

The figure shows only the relative density of the numbers; in reality, the density is immensely greater than it is possible to show in such a figure. For example, there are 2^{23} (8,388,608) single-precision numbers in the interval $2^{-126} \leq x < 2^{-125}$. ◆

Figure 2-2 Normalized single-precision numbers on the number line



Using only normalized representations creates a gap around the value 0, as shown in Figure 2-2. If a computer supports only the normalized numbers, it must round all tiny values to 0. For example, suppose such a computer must perform the operation $x - y$, where x and y are very close to, but not equal to, each other. If the difference between x and y is smaller than the smallest normalized number, the computer must deliver 0 as the result. Thus, for such **flush-to-zero systems,** the following statement is *not* true for all real numbers:

$x - y = 0$ if and only if $x = y$

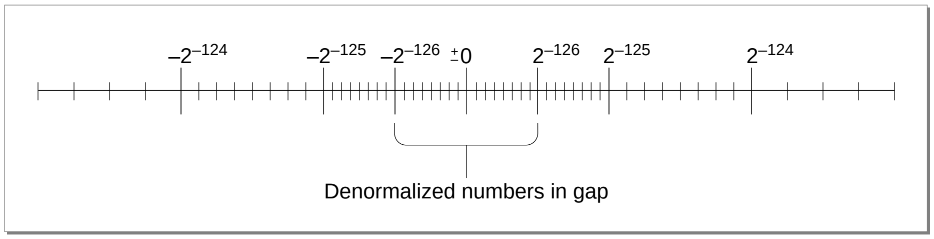
Denormalized Numbers

Instead of using only normalized numbers and allowing this small gap around 0, PowerPC processor-based Macintosh computers use **denormalized numbers,** in which the leading implicit bit (b_0 on page 2-4) of the significand is 0 and the minimum exponent is used.

Note
Some references use the term **subnormal numbers** instead of *denormalized numbers.* ◆

Figure 2-3 illustrates the relative magnitudes of normalized and denormalized numbers in single precision. Notice that the denormalized numbers have the same density as the numbers in the smallest normalized binade. This means that the roundoff error is the same regardless of whether an operation produces a denormalized number or a very small normalized number. As stated previously, without denormalized numbers, operations would have to round tiny values to 0, which is a much greater roundoff error.

Figure 2-3 Denormalized single-precision numbers on the number line



To put it another way, the use of denormalized numbers makes the following statement true for all real numbers:

$x - y = 0$ if and only if $x = y$

Another advantage of denormalized numbers is that error analysis involving small values is much easier without the gap around zero shown in Figure 2-2 (Demmel 1984).

The computer determines that a floating-point number is denormalized (and therefore that its implicit leading bit is interpreted as 0) when the biased exponent field is filled with 0's and the fraction field is nonzero.

Table 2-2 shows how a single-precision value A_0 becomes progressively denormalized as it is repeatedly divided by 2, with rounding to nearest. This process is called **gradual underflow.** In the table, values $A_2 \ldots A_{25}$ are denormalized; A_{25} is the smallest positive denormalized number in single format. Notice that as soon as the values are too small to be normalized, the biased exponent value becomes 0.

Table 2-2 Example of gradual underflow

Variable or operation	Value	Biased exponent	Comment
A_0	$1.100\ 1100\ 1100\ 1100\ 1100\ 1101 \times 2^{-125}$	2	
$A_1 = A_0/2$	$1.100\ 1100\ 1100\ 1100\ 1100\ 1101 \times 2^{-126}$	1	
$A_2 = A_1/2$	$0.110\ 0110\ 0110\ 0110\ 0110\ 0110 \times 2^{-126}$	0	Inexact*
$A_3 = A_2/2$	$0.011\ 0011\ 0011\ 0011\ 0011\ 0011 \times 2^{-126}$	0	Exact result
$A_4 = A_3/2$	$0.001\ 1001\ 1001\ 1001\ 1001\ 1010 \times 2^{-126}$	0	Inexact*
	.		
	.		
	.		
$A_{23} = A_{22}/2$	$0.000\ 0000\ 0000\ 0000\ 0000\ 0011 \times 2^{-126}$	0	Exact result
$A_{24} = A_{23}/2$	$0.000\ 0000\ 0000\ 0000\ 0000\ 0010 \times 2^{-126}$	0	Inexact*
$A_{25} = A_{24}/2$	$0.000\ 0000\ 0000\ 0000\ 0000\ 0001 \times 2^{-126}$	0	Exact result
$A_{26} = A_{25}/2$	0.0	0	Inexact*

* Whenever division returns an inexact tiny value, the exception bit for underflow is set to indicate that a low-order bit has been lost.

Infinities

An **Infinity** is a special bit pattern that can arise in one of two ways:

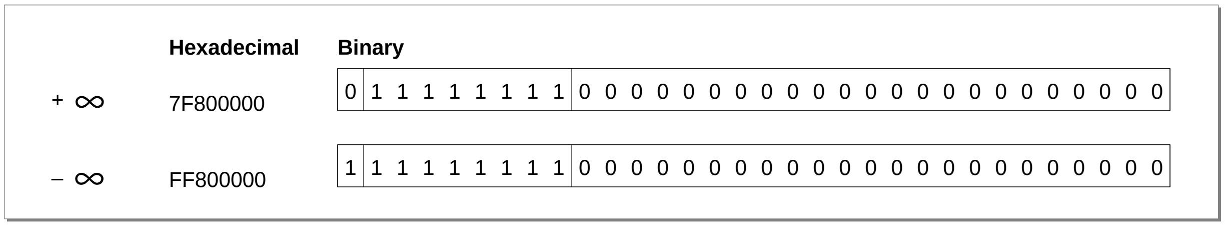
- When an operation (such as $1/0$) should produce a mathematical infinity, the result is an Infinity.
- When an operation attempts to produce a number with a magnitude too great for the number's intended floating-point data type, the result might be a value with the largest possible magnitude or it might be an Infinity (depending on the current rounding direction).

These bit patterns (as well as NaNs, introduced next) are recognized in subsequent operations and produce predictable results. The Infinities, one positive and one negative, generally behave as suggested by the theory of limits. For example:

- Adding 1 to $+\infty$ yields $+\infty$.
- Dividing -1 by $+0$ yields $-\infty$.
- Dividing 1 by $-\infty$ yields -0.

The computer determines that a floating-point number is an Infinity if its exponent field is filled with 1's and its fraction field is filled with 0's. So, for example, in single format, if the sign bit is 1, the exponent field is 255 (which is the maximum biased exponent for the single format), and the fraction field is 0, the floating-point number represented is $-\infty$ (see Figure 2-4).

Figure 2-4 Infinities represented in single precision



NaNs

When a numeric operation cannot produce a meaningful result, the operation delivers a special bit pattern called a **NaN (Not-a-Number).** For example, zero divided by zero, $+\infty$ added to $-\infty$, and $\sqrt{-1}$ yield NaNs. A NaN can occur in any of the numeric data formats (single, double, and double-double), but generally, system-specific **integer types** (non-numeric types exclusively for integer values) have no representation for NaNs.

NaNs propagate through arithmetic operations. Thus, the result of 3.0 added to a NaN is the same NaN. If two operands of an operation are NaNs, the result is one of the NaNs. NaNs are of two kinds: **quiet NaNs,** the usual kind produced by floating-point operations, and **signaling NaNs.**

When a signaling NaN is encountered as an operand of an arithmetic operation, the invalid-operation exception is signaled and a quiet NaN is the delivered result. Signaling NaNs are not created by any numeric operations, but you might find it useful to create signaling NaNs manually. For example, you might fill uninitialized memory with signaling NaNs so that if one is ever encountered in a program, you will know that uninitialized memory is accessed.

A NaN may have an associated code that indicates its origin. These codes are listed in Table 2-3. The NaN code is the 8th through 15th most significant bits of the fraction field.

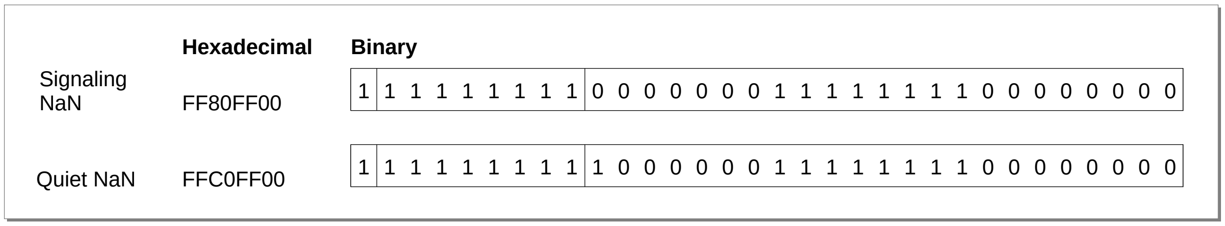
Table 2-3 NaN codes

Decima l	Hexadecima l	Meaning
1	0x01	Invalid square root, such as $\sqrt{-1}$
2	0x02	Invalid addition, such as $(+\infty) + (-\infty)$
4	0x04	Invalid division, such as $0/0$
8	0x08	Invalid multiplication, such as $0 \times \infty$
9	0x09	Invalid remainder or modulo, such as x rem 0
17	0x11	Attempt to convert invalid ASCII string
21	0x15	Attempt to create a NaN with a zero code
33	0x21	Invalid argument to trigonometric function (such as cos, sin, tan)
34	0x22	Invalid argument to inverse trigonometric function (such as acos, asin, atan)
36	0x24	Invalid argument to logarithmic function (such as log, $\log 10$)
37	0x25	Invalid argument to exponential function (such as exp, expm1)
38	0x26	Invalid argument to financial function (compound or annuity)
40	0x28	Invalid argument to inverse hyperbolic function (such as acosh, asinh)
42	0x2A	Invalid argument to gamma function (gamma or lgamma)

Note
The PowerPC processor always returns 0 for the NaN code. ◆

The computer determines that a floating-point number is a NaN if its exponent field is filled with 1's and its fraction field is nonzero. The most significant bit of the fraction field distinguishes quiet and signaling NaNs. It is set for quiet NaNs and clear for signaling NaNs. For example, in single format, if the sign field has the value 1, the exponent field has the value 255, and the fraction field has the value 65,280, then the number is a signaling NaN. If the sign is 1, the exponent is 255, and the fraction field has the value 4,259,584 (which means the fraction field has a leading 1 bit), the value is a quiet NaN. Figure 2-5 illustrates these examples.

Figure 2-5 NaNs represented in single precision



Zeros

Each floating-point format has two representations for zero: $+0$ and -0. Although the two zeros compare as equal $(+0) = -0$, their behaviors in IEEE arithmetic are slightly different.

Ordinarily, the sign of zero does not matter except (possibly) for a function discontinuous at zero. Though the two forms are numerically equal, a program can distinguish $+0$ from -0 by operations such as division by zero or by performing the numeric copysign function.

The sign of zero obeys the usual sign laws for multiplication and division. For example, $(+0) \times (-1) = -0$ and $1/(-0) = -\infty$. Because extreme negative underflows yield -0, expressions like $1/x^3$ produce the correct sign for ∞ when x is tiny and negative. Addition and subtraction produce -0 only in these cases:

- $(-0) - (+0)$ yields -0
- $(-0) + (-0)$ yields -0

When rounding downward, with x finite,

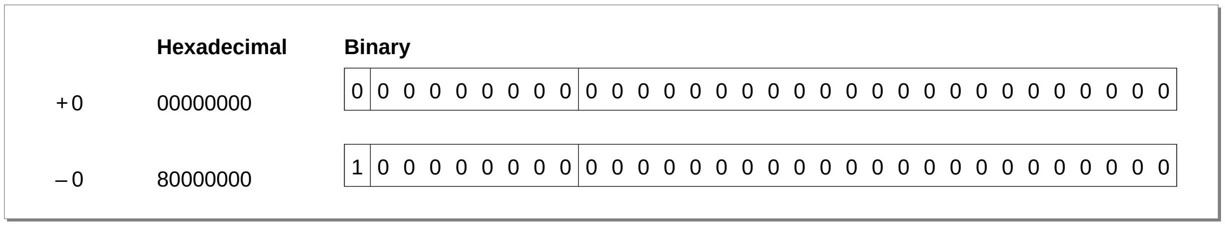
- $x - x$ yields -0
- $x + (-x)$ yields -0

The square root of -0 is -0.

The sign of zero is important in complex arithmetic (Kahan 1987).

The computer determines that a floating-point number is 0 if its exponent field and its fraction field are filled with 0's. For example, in single format, if the sign bit is 0, the exponent field is 0, and the fraction field is 0, the number is +0 (see Figure 2-6).

Figure 2-6 Zeros represented in single precision



Formats

This section shows the three numeric data formats: single, double, and double-double. These are pictorial representations and might not reflect the actual byte order in any particular implementation.

Each of the diagrams on the following pages is followed by a table that gives the rules for evaluating the number. In each field of each diagram, the leftmost bit is the most significant bit (msb) and the rightmost is the least significant bit (lsb). Table 2-4 defines the symbols used in the diagrams.

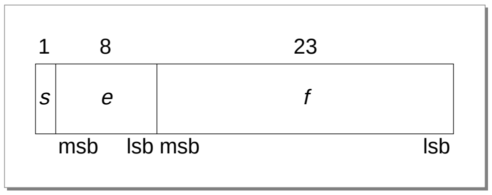
Table 2-4 Symbols used in format diagrams

Symbol	Description
v	Value of number
s	Sign bit
e	Biased exponent (*exponent* + *bias*)
f	Fraction (*significand* without leading bit)

Single Format

The 32-bit **single format** is divided into three fields having 1, 8, and 23 bits (see Figure 2-7).

Figure 2-7 Single format



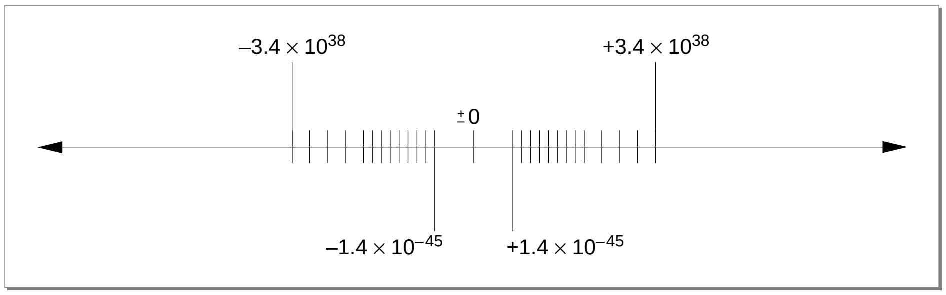
The interpretation of a single-format number depends on the values of the exponent field (*e*) and the fraction field (*f*), as shown in Table 2-5.

Table 2-5 Values of single-format numbers (32 bits)

If biased exponent *e* is:	And fraction *f* is:	Then value *v* is:	And the class of *v* is:
$0 < e < 255$	(any)	$v = (-1)^s \times 2^{(e-127)} \times (1.f)$	Normalized
$e = 0$	$f \neq 0$	$v = (-1)^s \times 2^{(-126)} \times (0.f)$	Denormalized
$e = 0$	$f = 0$	$v = (-1)^s \times 0$	Zero
$e = 255$	$f = 0$	$v = (-1)^s \times \infty$	Infinity
$e = 255$	$f \neq 0$	v is a NaN	NaN

Figure 2-8 shows the range and density of the real numbers that can be represented as single-format floating-point numbers using normalized and denormalized values. The vertical marks indicate the relative density of the numbers that can be represented. As explained in the section "Normalized Numbers" on page 2-5, the number of representable values gets more dense closer to 0.

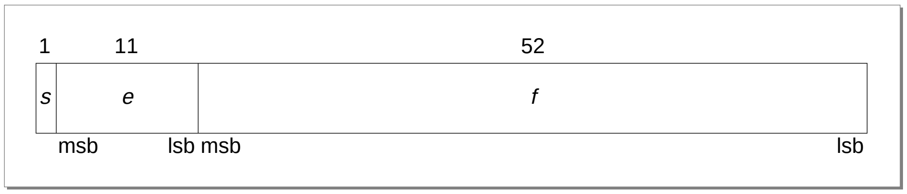
Figure 2-8 Single-format floating-point numbers on the real number line



Double Format

The 64-bit **double format** is divided into three fields having 1, 11, and 52 bits (see Figure 2-9).

Figure 2-9 Double format



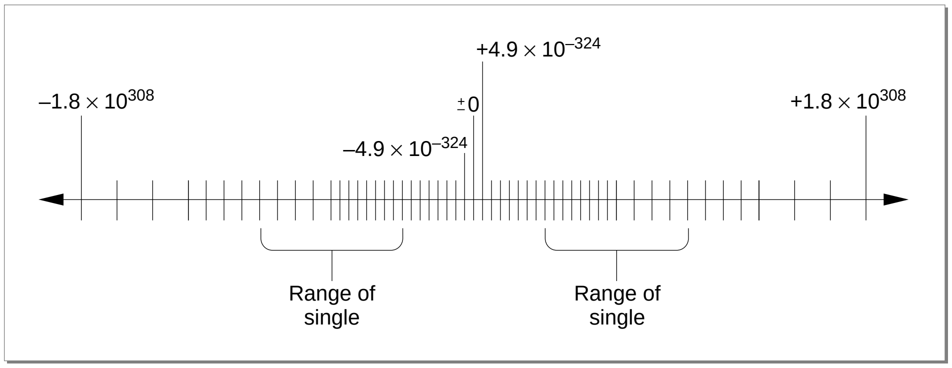
The interpretation of a double-format number depends on the values of the exponent field (*e*) and the fraction field (*f*), as shown in Table 2-6.

Table 2-6 Values of double-format numbers (64 bits)

If biased exponent *e* is:	And fraction *f* is:	Then value *v* is:	And the class of *v* is:
$0 < e < 2047$	(any)	$v = (-1)^s \times 2^{(e-1023)} \times (1.f)$	Normalized
$e = 0$	$f \neq 0$	$v = (-1)^s \times 2^{(-1022)} \times (0.f)$	Denormalized
$e = 0$	$f = 0$	$v = (-1)^s \times 0$	Zero
$e = 2047$	$f = 0$	$v = (-1)^s \times \infty$	Infinity
$e = 2047$	$f \neq 0$	v is a NaN	NaN

Figure 2-10 shows the range and density of the real numbers that can be represented as double-format floating-point numbers using normalized and denormalized values. The vertical marks indicate the relative density of the numbers that can be represented. As explained in the section “Normalized Numbers” on page 2-5, the number of representable values gets more dense closer to 0.

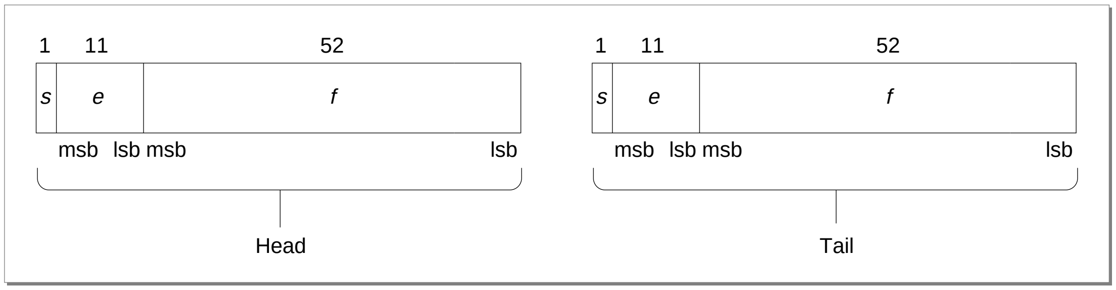
Figure 2-10 Double-format floating-point values on the real number line



Double-Double Format

The 128-bit **double-double format** is made up of two double-format numbers (see Figure 2-11).

Figure 2-11 Double-double format



The value of a double-double number is the sum of its head and tail components. These two components are both double numbers, and therefore the value of each component is determined as shown in Table 2-6. It is recommended that the tail's exponent be at least 54 less than the head's exponent. Numeric operations that produce double-double results always produce numbers in this form.

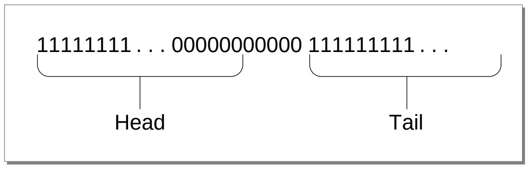
IMPORTANT

It is possible, but not recommended, to create a double-double format that does not follow this form. If you do not follow this form when creating a double-double number, the results are unpredictable. ▲

The requirement that the tail's exponent be at least 54 less than the head's exponent guarantees that the significand of the tail is more or less concatenated to the significand of the head (which is 53 bits long) when the two values are added together. For example, if the head component's exponent is 2^{200}, the tail component's exponent can be no greater than 2^{146}, so that in the value represented by this double-double format number, the head represents the first 53 binary digits and the tail represents the remaining digits.

Note that the difference between the exponent values may be greater than 54 and that the head and the tail can have different signs. To continue with the example, suppose the tail's exponent is 2^{140} instead of 2^{146}. The binary number represented would be as shown in Figure 2-12.

Figure 2-12 Double-double format number example



The head represents the binary places 2^{200} down to 2^{147}. The tail represents the binary places 2^{140} down to 2^{87}. The zeros between the head and the tail are necessary to represent the binary places 2^{146} to 2^{141}. This particular number has 112 units of precision—53 units from the head, 53 from the tail, and 6 units between the head and the tail. The double-double format always has at least 107 bits of precision, and if the tail's exponent is more than 54 less than the head's exponent, it has even greater precision.

If the value of the head component is a normalized number, then the value of the double-double number is the sum of the head and the tail. In the recommended form, if the head is not a normalized number (meaning it is denormalized, 0, NaN, or Infinity), the head contains the value of the double-double number, and the tail contains 0. This way, when you add the head and the tail, you still get the value of the head.

Although the precision of the double-double format is much greater than that of the double format, the range of the two formats is the same. However, because the double-double format is implemented in software, this format is much slower to use than the double format. Because of this, you should always use the double format unless you need the extra precision provided by the double-double format.

Range and Precision of Data Formats

Table 2-7 shows the precision, range, and memory usage for each numeric data format. You can use this table to compare the data formats and choose which one is needed for your application. Typically, choosing a data format requires that you determine the tradeoffs between

- fixed-point or floating-point form
- precision
- range
- memory usage
- speed

In the table, decimal ranges are expressed as rounded, two-digit decimal representations of the exact binary values. The speed of a given data format varies depending on the particular implementation of PowerPC Numerics. (See Chapter 5, "Conversions," for information on aspects of conversion relating to precision.)

Table 2-7 Summary of PowerPC Numerics data formats

	Single	**Double**	**Double-double**
Size (bytes:bits)	4:32	8:64	16:128
Range of binary exponents			
Minimum	–126	–1022	–1022
Maximum	127	1023	1023
Significand precision			
Bits	24	53	≥ 107
Decimal digits	7–8	15–16	≥ 32
Decimal range (approximate)			
Maximum positive	$3.4 \times 10^{+38}$	$1.8 \times 10^{+308}$	$1.8 \times 10^{+308}$
Minimum positive norm	1.2×10^{-38}	2.2×10^{-308}	2.2×10^{-308}
Minimum positive denorm	1.4×10^{-45}	4.9×10^{-324}	4.9×10^{-324}
Maximum negative denorm	-1.4×10^{-45}	-4.9×10^{-324}	-4.9×10^{-324}
Maximum negative norm	-1.2×10^{-38}	-2.2×10^{-308}	-2.2×10^{-308}
Minimum negative	$-3.4 \times 10^{+38}$	$-1.8 \times 10^{+308}$	$-1.8 \times 10^{+308}$

For example, in single format, the largest representable number is composed as follows:

significand	$=$	$(2 - 2^{-23})$
	$=$	$1.11111111111111111111111_2$
exponent	$=$	127
value	$=$	$(2 - 2^{-23}) \times 2^{127}$
	$\approx$	3.403×10^{38}

The smallest positive normalized number representable in single format is made up as follows:

significand	$=$	1
	$=$	$1.00000000000000000000000_2$
exponent	$=$	-126
value	$=$	1×2^{-126}
	$\approx$	1.175×10^{-38}

For denormalized numbers, the smallest positive value representable in the single format is made up as follows:

significand	$=$	2^{-23}
	$=$	$0.00000000000000000000001_2$
exponent	$=$	-126
value	$=$	$2^{-23} \times 2^{-126}$
	$\approx$	1.401×10^{-45}