This chapter describes the operations (comparisons, arithmetic operations, and auxiliary and transcendental functions) that PowerPC Numerics allows you to perform on floating-point numbers. Numeric operations are evaluated as floating-point expressions; as such they are affected by, and might affect, the floating-point environment.

Read this chapter to find out what numeric operations are supported and how they work. For more information about how floating-point operations are evaluated in general, see Chapter 3, "Expression Evaluation." For a description of the floating-point environment, see Chapter 4, "Environmental Controls."

Comparisons

PowerPC Numerics supports the usual numeric comparisons: less than, less than or equal to, greater than, greater than or equal to, equal to, and not equal to (see Table 6-1 for a complete listing). For real numbers, these comparisons behave according to the familiar ordering of real numbers.

Comparisons With NaNs and Infinities

Numeric comparisons handle NaNs and Infinities as well as real numbers. The usual trichotomy for real numbers is extended so that, for any numeric values *a* and *b*, exactly one of the following statements is true:

- $a < b$
- $a > b$
- $a = b$
- *a* and *b* are unordered

The following rule determines which statement is true: If *a* or *b* is a NaN, then *a* and *b* are unordered; otherwise, *a* is less than, equal to, or greater than *b* according to the ordering of the real numbers, with the understanding that

$+0 = -0$ and $-\infty <$ every real number $< +\infty$

Comparison Operators

The meaning of high-level language relational operators is a natural extension of their old meaning based on trichotomy. For example, the C expression $x <= y$ is true if *x* is less than *y* or if *x* equals *y*, and is false if *x* is greater than *y* or if *x* and *y* are unordered. Note that the numeric not-equal relation means less than, greater than, or unordered.

The FPCE technical report extends the usual set of C relational operators to a set of 14 comparisons, shown in Table 6-1.

Table 6-1 Comparison symbols

Symbol	Relation	Invalid if unordered?
<	Less than	Yes
>	Greater than	Yes
<=	Less than or equal to	Yes
>=	Greater than or equal to	Yes
==	Equal to	No
!=	Not equal to (unordered, less than, or greater than)	No
!<>=	Unordered	No
<>	Less than or greater than	Yes
<>=	Not unordered (less than, equal to, or greater than)	Yes
!<=	Not less than or equal to (unordered or greater than)	No
!<	Not less than (unordered, greater than, or equal to)	No
!>=	Not greater than or equal to (unordered or less than)	No
!>	Not greater than (unordered, less than, or equal to)	No
!<>	Unordered or equal	No

Some relational operators in high-level language comparisons contain the predicate less than or greater than, but not unordered. In C, those relational operators are <, <=, >, and >= (but not == and !=). For those relations, comparisons signal invalid if the operands are unordered, that is, if either operand is a NaN. For the operators equal and nonequal, comparisons with NaN are not misleading; thus, when x or y is a NaN, the relation $x == y$ is false, which is not misleading. Likewise, when x or y is a NaN, $x != y$ returns true, again not misleading. On the other hand, when x or y is a NaN, $x < y$ being false might tempt you to conclude that $x \geq y$, so PowerPC Numerics signals invalid to help you avoid the pitfall. Table 6-1 shows the results of such comparisons in C.

The full 26 distinct comparison predicates of the IEEE standard may be obtained by logical negation of all of the operators except for == and !=, which never signal invalid. For example, $(x < y)$ and $!(x !< y)$ are logically equivalent for all possible values of a and b, but the former raises the invalid exception flag when x and y compare unordered while the latter does not.

A comparison with a signaling NaN as an operand always signals invalid, just as in arithmetic operations.

In addition to the comparison operators, there are also library functions that perform comparisons. See “Comparison Functions” in Chapter 10, “Transcendental Functions.”

Arithmetic Operations

PowerPC Numerics provides the seven arithmetic operations required by the IEEE standard for its three data types, as shown for the C language in Table 6-2 and described in the sections that follow.

Table 6-2 Arithmetic operations in C

Operation	C symbol
Add	`+`
Subtract	`-`
Multiply	`*`
Divide	`/`
Square root	`sqrt`
Remainder	`remainder`
Round-to-integer	`rint`

The language processors for the PowerPC automatically use their chosen expression evaluation methods for the normal inline operators (`+`, `-`, `*`, `/`). All the arithmetic operations produce the best possible result: the mathematically exact result, coerced to the precision and range of the evaluation format. The coercions honor the user-selectable rounding direction and handle all exceptions according to the requirements of the IEEE standard (see Chapter 4, “Environmental Controls”).

Some of the arithmetic operations are implemented in software. These operations are declared to be type `double_t`, which is defined to be type `double`.

+

You can use the + symbol to add two real numbers.

```
x + y
```

`x`	Any floating-point number.
`y`	Any floating-point number.

DESCRIPTION

The + operator performs the standard addition of two floating-point numbers.

EXCEPTIONS

When x and y are both finite and nonzero, either the result of $x + y$ is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or if an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 6-3 shows the results when one of the operands of the addition operation is a zero, a NaN, or an Infinity. In this table, x is any floating-point number.

Table 6-3 Special cases for floating-point addition

Operation	Result	Exceptions raised
$x + (+0)$	x	None
$x + (-0)$	x	None
$(-0) + (+0)$	+0	None
$(-0) + (-0)$	–0	None
x + NaN	NaN	None*
$x + (+\infty)$	$+\infty$	None
$x + (-\infty)$	$-\infty$	None
$+\infty + (-\infty)$	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

–

You can use the – symbol to subtract one real number from another.

```
x - y
```

x	Any floating-point number.
y	Any floating-point number.

DESCRIPTION

The – operator performs the standard subtraction of two floating-point numbers.

EXCEPTIONS

When x and y are both finite and nonzero, either the result of $x - y$ is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or if an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 6-4 shows the results when one of the operands of the subtraction operation is a zero, a NaN, or an Infinity. In this table, x is any floating-point number.

Table 6-4 Special cases for floating-point subtraction

Operation	Result	Exceptions raised
$x - (+0)$	x	None
$(+0) - x$	$-x$	None
$(+0) - (-0)$	$+0$	None
$x - (-0)$	x	None
$(-0) - x$	$-x$	None
$(-0) - (+0)$	-0	None
$(-0) - (-0)$	$+0$	None
x – NaN	NaN	None*
NaN – x	NaN	None*
$x - (+\infty)$	$-\infty$	None
$(+\infty) - x$	$+\infty$	None
$(+\infty) - (+\infty)$	NaN	Invalid
$x - (-\infty)$	$+\infty$	None
$(-\infty) - x$	$-\infty$	None
$(-\infty) - (-\infty)$	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

*

You can use the * symbol to multiply two real numbers.

```
x * y
```

x	Any floating-point number.
y	Any floating-point number.

DESCRIPTION

The * operator performs the standard multiplication of two floating-point numbers $(x \times y)$.

EXCEPTIONS

When x and y are both finite and nonzero, either the result of x * y is exact or it raises one of the following exceptions:

- inexact (if the result of x * y must be rounded or if an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 6-5 shows the results when one of the operands of the multiplication operation is a zero, a NaN, or an Infinity. In this table, x is a nonzero floating-point number.

Table 6-5 Special cases for floating-point multiplication

Operation	Result	Exceptions raised
x * +0	±0	None
x * –0	±0	None
±∞ * ±0	NaN	Invalid
x * NaN	NaN	None*
x * +∞	±∞	None
x * –∞	±∞	None
±0 * ±∞	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

/

You can use the / symbol to divide one real number by another.

```
x / y
```

x	Any floating-point number.
y	Any floating-point number.

DESCRIPTION

The / operator performs the standard division of two floating-point numbers.

EXCEPTIONS

When x and y are both finite and nonzero, either the result of x/y is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or if an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 6-6 shows the results when one of the operands of the division operation is a zero, a NaN, or an Infinity. In this table, x is any floating-point number.

Table 6-6 Special cases for floating-point division

Operation	Result	Exceptions raised
$(+0)/x$	± 0	None
$x/(+0)$	$\pm\infty$	Divide-by-zero
$(-0)/x$	± 0	None
$x/(-0)$	$\pm\infty$	Divide-by-zero
$\pm 0/\pm 0$	NaN	Invalid
x/NaN	NaN	None*
NaN/x	NaN	None*
$x/(+\infty)$	± 0	None

continued

Table 6-6 Special cases for floating-point division (continued)

Operation	Result	Exceptions raised
$(+\infty)/x$	$\pm\infty$	None
$x / (-\infty)$	± 0	None
$(-\infty)/x$	$\pm\infty$	None
$(\pm\infty)/(\pm\infty)$	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

sqrt

You can use the square root (`sqrt`) function to compute the square root of a real number.

```
double_t sqrt(double_t x);
```

`x` Any positive floating-point number.

DESCRIPTION

$\text{sqrt}(x) = \sqrt{x}$

EXCEPTIONS

When x is finite and nonzero, either the result of $\text{sqrt}(x)$ is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded)
- invalid (if x is negative)

SPECIAL CASES

Table 6-7 shows the results when the argument to the square root function is a zero, a NaN, or an Infinity, plus other special cases for the square root function. In this table, x is a finite, nonzero floating-point number.

Table 6-7 Special cases for floating-point square root

Operation	Result	Exceptions raised
sqrt(x) for $x < 0$	NaN	Invalid
sqrt(+0)	+0	None
sqrt(–0)	–0	None
sqrt(NaN)	NaN	None*
sqrt(+∞)	+∞	None
sqrt(–∞)	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

remainder, remquo, and fmod

You can use the `remainder`, `remquo`, and `fmod` functions to perform the remainder operation recommended in the IEEE standard.

```
double_t remainder (double_t x, double_t y);
double_t remquo (double_t x, double_t y, int *quo);
double_t fmod (double_t x, double_t y);
```

`x`	Any floating-point number.
`y`	Any floating-point number.
`quo`	On return, the signed lowest seven bits (in the range of –127 to +127, inclusive) of the integer value closest to the quotient x/y. This partial quotient might be of use in certain argument reduction algorithms.

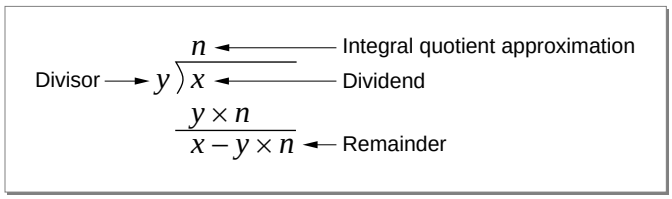
DESCRIPTION

The IEEE remainder (rem) operation returns the result of the following computation.

$r = x \text{ rem } y = x - y \times n$

where n is the integer nearest the exact value of the quotient x/y. This expression can be found even in the conventional integer-division algorithm, shown in Figure 6-1.

Figure 6-1 Integer-division algorithm



Whenever $|n - x/y| = 1/2$, n is even.

If the value of r is 0, the sign of r is that of x.

The rem operation is always exact.

The IEEE rem operation differs from other commonly used remainder and modulo operations. It returns a remainder of the smallest possible magnitude, and it always returns an exact remainder. Other remainder functions can be constructed from the IEEE remainder function by appropriately adding or subtracting y.

EXCEPTIONS

When x and y are finite, nonzero floating-point numbers in single or double format, the result of x rem y is exact.

SPECIAL CASES

Table 6-8 shows the results when one of the arguments to the rem operation is a zero, a NaN, or an Infinity. In this table, x is a finite, nonzero floating-point number.

Table 6-8 Special cases for floating-point remainder

Operation	Result	Exceptions raised
+0 rem x	+0	None
x rem (+0)	NaN	Invalid
–0 rem x	–0	None
x rem (–0)	NaN	Invalid
x rem NaN	NaN	None*
NaN rem x	NaN	None*
x rem +∞	x	None
+∞ rem x	NaN	Invalid
x rem –∞	x	None
–∞ rem x	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = remainder(5, 3);    /* z = –1. */
/* 5 rem 3 = 5 – 3 × 2 = –1 because 1 < 5/3 < 2 and because
   5/3 = 1.66666... is closer to 2 than to 1, quo is taken to
   be 2. */

z = remainder(43.75, 2.5); /* z = –1.25. */
/* 43.75 rem 2.5 = 43.75 – 2.5 × 18 = –1.25 because
   17 < 43.75/2.5 < 18 and because 43.75/2.5 = 17.5 is
   equally close to both 17 and 18, quo is taken to be the
   even quotient, 18. */

z = remainder(43.75, +INFINITY); /* z = 43.75 */
/* 43.75 rem ∞ = 43.75 – 0 × ∞ = 43.75 because 43.75 / ∞ = 0,
   quo is taken to be 0. */
```

rint

You can use the round-to-integer operation (`rint` function) to round a number to the nearest integer in the current rounding direction.

```
double_t rint(double_t x);
```

x Any floating-point number.

DESCRIPTION

The `rint` function rounds its argument to an integer in the current rounding direction. The available rounding directions are upward, downward, to nearest (default), and toward zero. With the default rounding direction, if the argument is equally near two integers, the even integer is used as the result.

In each floating-point data type, all values of sufficiently great magnitude are integers. For example, in single format, all numbers whose magnitudes are at least 2^{23} are integers. This means that $+\infty$ and $-\infty$ are already integers and return exact results.

The `rint` function performs the round-to-integer arithmetic operation described in the IEEE standard. For other C functions that perform rounding to integer, see Chapter 9, "Conversion Functions."

EXCEPTIONS

When x is finite and nonzero, either the result of $\mathrm{rint}(x)$ is exact or it raises the following exception

- inexact (if x is not an integer)

SPECIAL CASES

Table 6-9 shows the results when the argument to the round-to-integer operation is a zero, a NaN, or an Infinity.

Table 6-9 Special cases for floating-point round-to-integer

Operation	Result	Exceptions raised
rint(+0)	+0	None
rint(–0)	–0	None
rint(NaN)	NaN	None*
rint(+∞)	+∞	None
rint(–∞)	–∞	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

Table 6-10 shows some example results of `rint`, given different rounding directions.

Table 6-10 Examples of `rint`

	Current rounding direction			
Example	To nearest	Toward 0	Downward	Upward
`rint(1.5)`	2	1	1	2
`rint(2.5)`	2	2	2	3
`rint(-2.2)`	–2	–2	–3	–2

Auxiliary Functions

The IEEE standard defines a number of recommended functions (called *auxiliary functions*) that are generally useful in numerical programming. The recommended functions supported by PowerPC Numerics are

- copysign(x, y) : copy the sign
- fabs(x) : absolute value
- logb(x) : binary exponent
- nan functions: NaN generators
- nextafter functions

- scalb(x) : binary scaling

The auxiliary functions are provided in the C library MathLib. For more information about these functions, see Part 2.

Transcendental Functions

PowerPC Numerics provides several basic mathematical functions in addition to the auxiliary functions recommended in the IEEE standard. These functions include

- logarithms
- exponentials
- two important financial functions
- trigonometric functions
- a random number generator
- error and gamma functions

For information about the transcendental functions supported, see Part 2.