Floating-Point Instructions 10

CHAPTER 10
FLOATING-POINT INSTRUCTIONS

This chapter describes the floating-point processing capabilities of the 80960SB processor. The subjects discussed include the real number data types, the execution environment for floating-point operations, the floating-point instructions, and fault and exception handling.

NOTE

This chapter applies only to the 80960SB processor.

INTRODUCING THE 80960SB FLOATING-POINT ARCHITECTURE

The floating-point architecture used in the 80960SB processor is designed to allow a convenient implementation of the IEEE Standard 754-1985 for Binary Floating-Point Arithmetic. This hardware architecture, along with a small amount of software support, conforms to the IEEE standard and provides support for the following data structures and operations:

- Real (32-bit), long-real (64-bit), and extended-real (80-bit) floating-point number formats.
- Add, subtract, multiply, divide, square root, remainder, and compare operations
- Conversion between integer and floating-point formats
- Conversion between different floating-point formats
- Handling of floating-point exceptions, including non-numbers (NaNs)

The software to support the 80960SB floating-point architecture is needed primarily to handle conversions between real numbers and decimal strings.

In addition, the 80960SB floating-point architecture supports several functions that go beyond the IEEE standard. These functions fall into two categories:

- functions recommended in the appendix to the IEEE standard, such as copy sign and classify, and
- commonly used transcendental functions, including trigonometric, logarithmic, and exponential functions.

REAL NUMBERS AND FLOATING-POINT FORMAT

This section provides an introduction to real numbers and how they are represented in floating-point format. Readers who are already familiar with numeric processing techniques and the IEEE standard may wish to skip this section.

Real Number System

As shown at the top of Figure 10-1, the real-number system comprises the continuum of real numbers from minus infinity (-∞) to plus infinity (+∞).

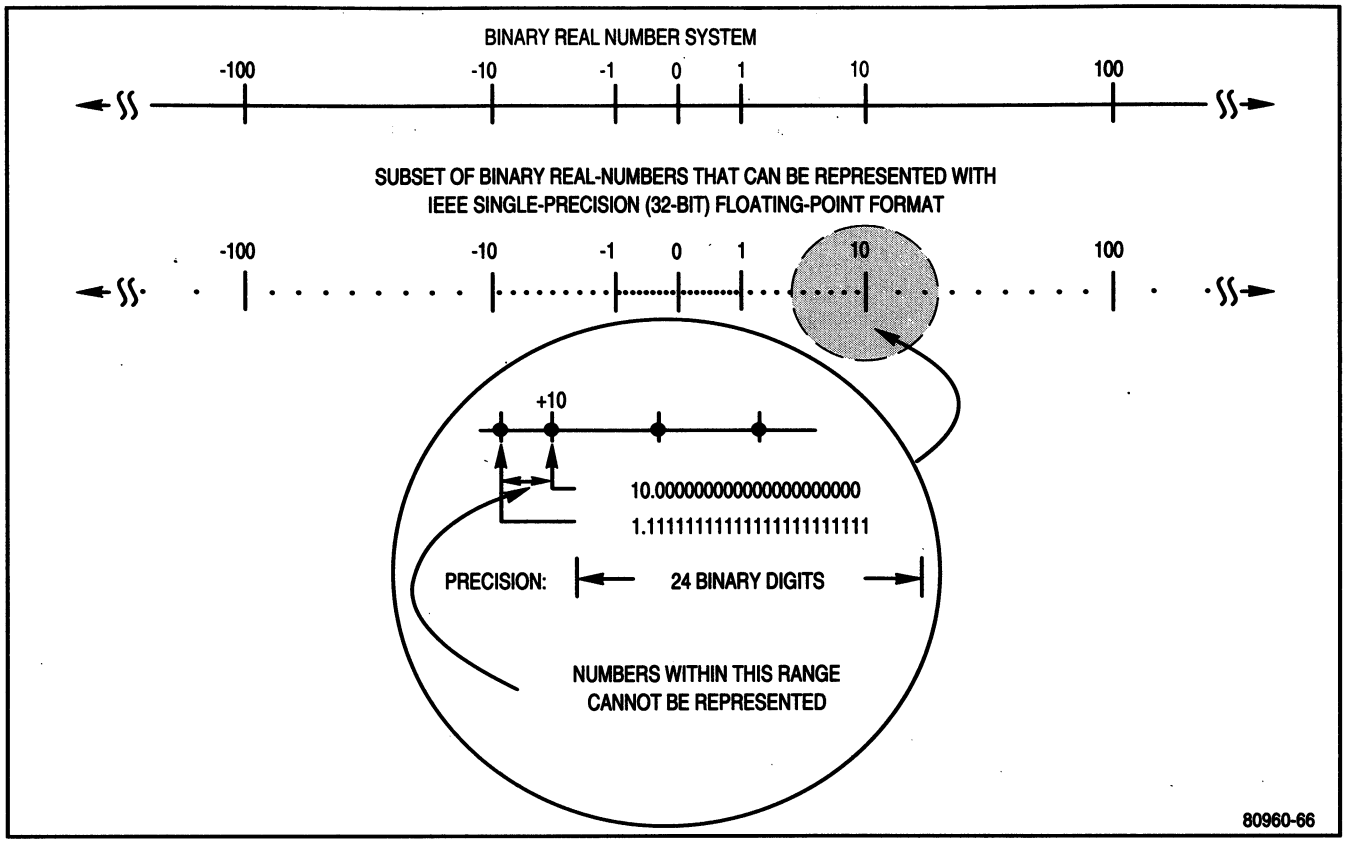

Figure 10-1: Binary Number System

Because the size and number of registers that any computer can have is limited, only a subset of the real-number continuum can be used in real-number calculations. As shown at the bottom of Figure 10-1, the subset of real numbers that a particular processor supports represents an approximation of the real number system. The range and precision of this real-number subset is determined by the format that the processor uses to represent real numbers.

Floating-Point Format

To increase the speed and efficiency of real number computations, computers or numeric processors typically represent real numbers in a binary floating-point format. In this format, a real number has three parts: a sign, a significand, and an exponent. Figure 10-2 shows the binary floating-point format that the processor uses. This format conforms to the IEEE standard.

The sign is a binary value that indicates whether the number is positive (0) or negative (1). The significand has two parts: a one-bit binary integer (also referred to as the j-bit) and a binary fraction. The j-bit is often not represented, but instead is an implied value. The exponent is a binary integer that represents the base-2 power that the significand is raised to.

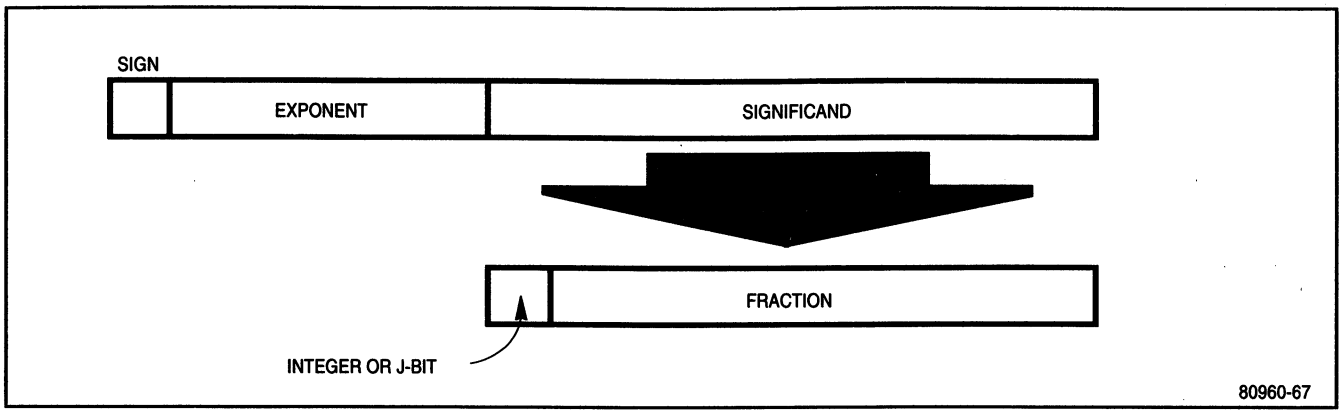

Figure 10-2: Binary Floating-Point Format

Table 10-1 shows how the real number 201.187 (in ordinary decimal format) is stored in floating-point format. The table lists a progression of real number notations that leads to the format that the 80960SB processor uses. In this format, the binary real number is normalized and the exponent is biased.

Table 10-1: Real-Number Notation

Notation	Value		
Ordinary Decimal	201.187		
Scientific Decimal	$2.011873E_{10}2$		
Scientific Binary	$1.1001001001011111E_2111$		
Scientific Binary	$1.1001001001011111E_210000110$		
32-Bit Floating Point Format (Normalized)	**Sign**	**Biased Exponent**	**Significand**
	0	10000110	1001001001011111 1. (Implied)

Normalized Numbers

In most cases, the processor represents real numbers in normalized form. This means that except for zero, the significand is always made up of an integer of 1 and a fraction as follows:

1.fff...ff

For values less than 1, leading zeros are eliminated. (For each leading zero eliminated, the exponent is decremented by one.)

Representing numbers in normalized form maximizes the number of significant digits that can be accommodated in a significand of a given width. To summarize, a normalized real number consists of a normalized significand that represents a real number between 1 and 2 and an exponent that gives the number's binary point.

Biased Exponent

The processor represents exponents in a biased form. This means that a constant is added to the actual exponent so that the biased exponent is always a positive number. The value of the biasing constant depends on the number of bits available for representing exponents in the floating-point format being used. The biasing constant is chosen so that the smallest normalized number can be reciprocated without overflow.

(The biasing constants for the various sizes of real data types that the processor supports are given in the section later in this chapter titled "Real Data Types.")

Real Number and Non-Number Encodings

The real numbers that are encoded in the floating-point format described above are generally divided into three classes: ±0, ± nonzero-finite numbers, and ±∞. Encodings for non-numbers (NaNs) are also defined. The term NaN stands for "Not a Number."

Figure 10-3 shows how the encodings for these numbers and non-numbers fit into the real number continuum. The encodings shown here are for the IEEE single-precision (32-bit) format, where the term "S" indicates the sign bit, "E" the biased exponent, and "F" the fraction (the exponent values are given in decimal).

Signed Zeros

Zero can be represented as a +0 or a -0 depending on the sign bit. Both encodings are equal in value. The sign of a zero result depends on the operation being performed and the rounding mode being used. Signed zeros have been provided to aid in implementing interval arithmetic. The sign of a zero may indicate the direction from which underflow occurred, or it may indicate the sign of an ∞ that has been inverted.

Signed, Nonzero, Finite Values

The class of signed, nonzero, finite values is divided into two groups: normalized and denormalized. The normalized finite numbers comprise all the nonzero finite values that can be encoded in a normalized real number format from zero to ∞ and zero to -∞. In the 32-bit form shown in Figure 10-3, this group of numbers includes all the numbers with biased exponents ranging from 1 to 254_{10} (unbiased, the exponent range is from -126_{10} to $+127_{10}$).

Denormalized Numbers

When real numbers become very close to zero, the normalized-number format can no longer be used to represent the numbers. This is because the range of the exponent is not large enough to compensate for shifting the binary point to the right to eliminate leading zeros.

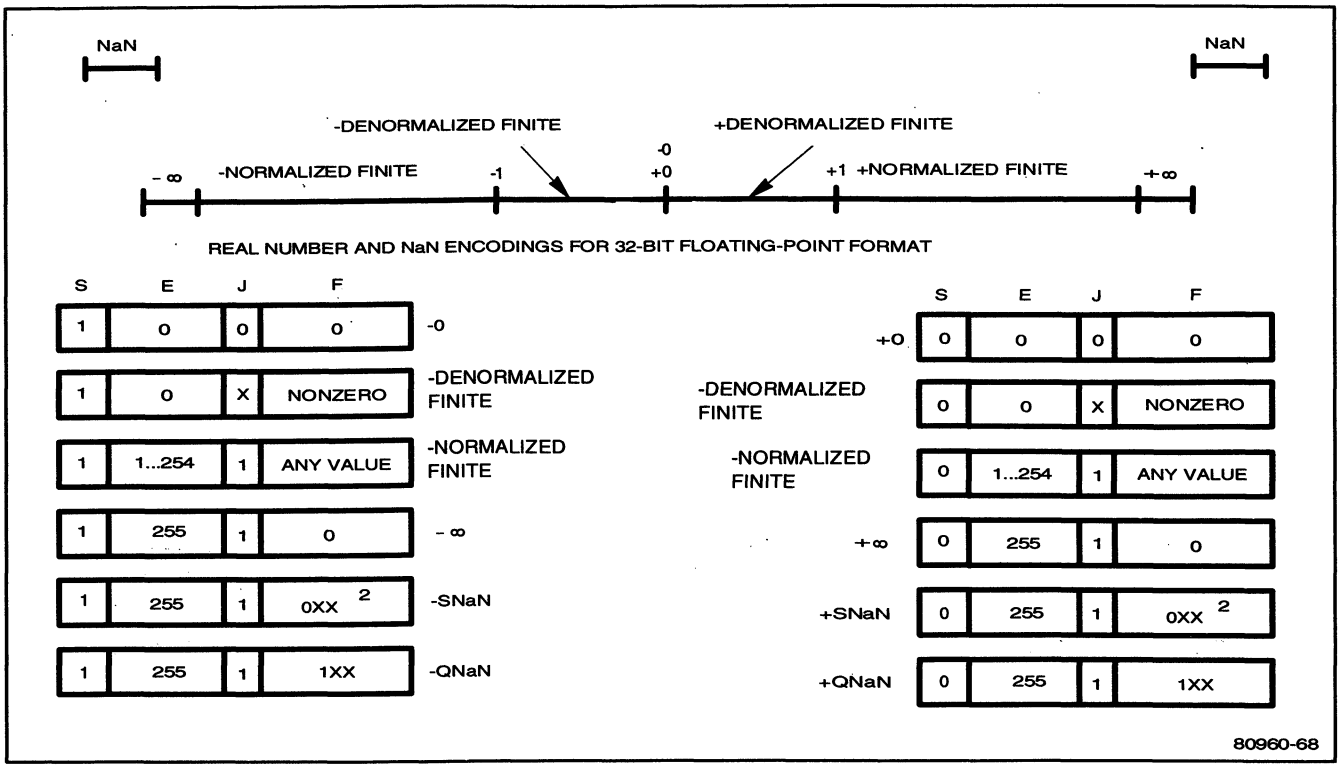

Figure 10-3: Real Numbers and NaNs

When the biased exponent is zero, smaller numbers can only be represented by making the integer bit (and perhaps other leading bits) of the significand zero. The numbers in this range are called denormalized numbers. The use of leading zeros with denormalized numbers allows smaller numbers to be represented. However, this denormalization causes a loss of precision (the number of significant bits in the fraction is reduced by the leading zeros).

When performing normalized floating-point computations, a processor normally operates on normalized numbers and produces normalized numbers as results. Denormalized numbers represent an underflow condition.

A denormalized number is computed through a technique called gradual underflow. Table 10-2 gives an example of gradual underflow in the denormalization process. Here the 32-bit format is being used, so the minimum exponent (unbiased) is -126_{10}. The true result in this example requires an exponent of -129_{10} in order to have a normalized number. Since -129_{10} is beyond the allowable exponent range, the result is denormalized by inserting leading zeros until the minimum exponent of -126_{10} is reached.

Table 10-2: Denormalization Process

Operation	Sign	Exponent*	Significand
True Result	0	-129	1.01011100...00
Denormalize	0	-128	0.101011100...00
Denormalize	0	-127	0.0101011100...00
Denormalize	0	-126	0.00101011100...00
Denormal Result	0	-126	0.00101011100...00

*Expressed as unbiased, decimal number.

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

Signed Infinities

The two infinities, $+\infty$ and $-\infty$, represent the maximum positive and negative real numbers, respectively, that can be represented in the floating-point format. Infinity is always represented by a zero fraction and the maximum biased exponent allowed in the specified format (e.g., 255_{10} for the 32-bit format).

Whereas denormalized numbers represent an underflow condition, the two infinity numbers represent the result of an overflow condition. Here, the normalized result of a computation has a biased exponent greater than the largest allowable exponent for the selected result format.

NaNs

Since NaNs are non-numbers, they are not part of the real number line. In Figure 10-3, the encoding space for NaNs in the 80960SB floating-point formats is shown above the ends of the real number line. This space includes any value with the maximum allowable biased exponent and a non-zero fraction. (The sign bit is ignored for NaNs.)

The IEEE standard defines two specific NaN values: a quiet NaN (QNaN) and a signaling NaN (SNaN). A QNaN is a NaN with the most significant fraction bit set; an SNaN is a NaN with the most significant fraction bit clear. QNaNs are allowed to propagate through most arithmetic operations without signaling an exception. SNaNs signal an invalid-operation exception whenever they appear as operands in arithmetic operations. Exceptions are discussed later in this chapter in the section titled "Exceptions and Fault Handling."

The section at the end of this chapter titled "Operations on NaNs" provides detailed information on how the processor handles NaNs.

REAL DATA TYPES

The processor supports three real-number data formats: real, long real, and extended real. These formats correspond directly to the single-precision, double-precision, and double-extended precision formats in the IEEE standard. Figure 10-4 shows these data formats and gives the resolution that each provides.

As described earlier in this chapter, the processor represents exponents in a biased format. For real values, the biasing constant is 127; for long-real values, it is 1023; and for extended-real values, it is 16383.

For the real and long-real formats, only the fraction is given for the significand. The integer is assumed to be 1 for all numbers except 0 and denormalized finite numbers. For the extended-real format, the integer is contained in bit 63, and the most-significant fraction bit is bit 62. Here, the integer is explicitly set to 1 for normalized numbers, infinities, and NaNs, and to 0 for zero and denormalized numbers. A non-zero exponent with the integer bit set to zero is a reserved encoding, which will result in a floating reserved-encoding exception being signaled.

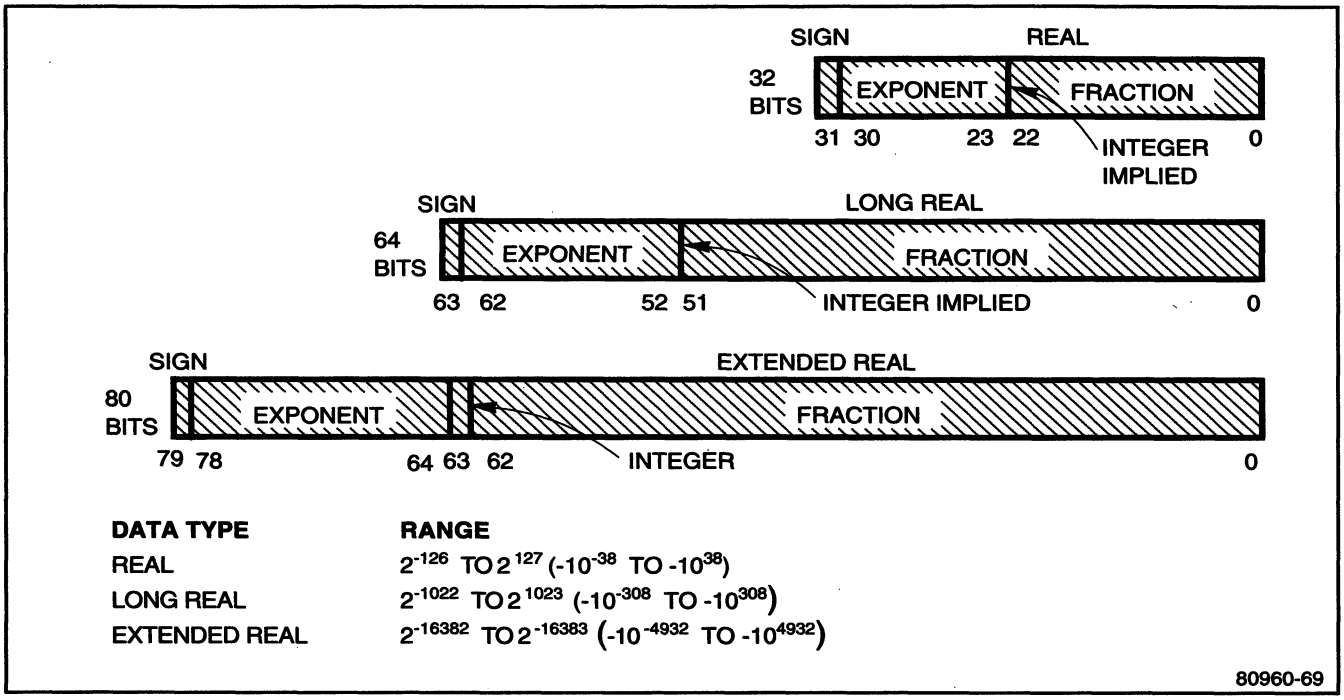

Figure 10-4: Real-Number Formats

Table 10-3 shows the encodings for all the classes of real numbers (i.e., zero, denormalized finite, normalized finite, and ∞) and NaNs, for each of the three real data-types.

Table 10-3: Real Numbers and NaN Encodings

Class		Sign	Biased Exponent	Integer[1]	Fraction
Positive	+∞	0	11...11	1	00...00
	+Normals	0 . . 0	11...10 . . 00...01	1 . . 1	11...11 00...00
	+Denormals	0 . . 0	00...00 . . 00...00	0 . . 0	11...11 . . 00..01
	+Zero	0	00...00	0	00..00
Negative	-Zero	1	00...00	1	00...00
	-Denormals	1 . . 1	00...00 . . 00...00	0 . . 0	00...01 11...11
	-Normals	1 . . . 1	00...01 . . . 11...10	1 . . . 1	00...00 . . . 11..11
	-∞	1	11...11	1	00..00
NaN	SNaN	X	11...11	1	0X...XX[2]
	QNaN	X	11...11	1	1X...XX
	Real:		←——— 8 bits ———→		← 23 bits→
	Long Real:		←——— 11 bits ———→		← 52 bits→
	Extended Real:		←——— 15 bits ———→		← 63 bits→

[1]Integer is implied for long real and real formats and is not stored.
[2]Fraction for SNaN must be non-zero.

EXECUTION ENVIRONMENT FOR FLOATING-POINT OPERATIONS

An important feature of the 80960SB processor is that the floating-point processing capabilities have been integrated into the execution environment of the processor. Operations on floating-point numbers are carried out using the same registers that are used for ordinals and integers. In addition, four floating-point registers have been provided for extended-precision floating-point arithmetic. The following sections describe how floating-point operations are handled in the processor's execution environment.

Registers

All of the registers in the processor's execution environment, (i.e., global, local, and floating point) can be used for floating-point operations. When using global or local registers, real values (i.e., 32 bits) are contained in one register; long-real values (i.e., 64 bits) are contained in two successive registers; and extended-real values (i.e., 80 bits) are contained in three successive registers.

Figure 10-5 shows how the three forms of the real data type are encoded when stored in global and local registers. Note that long-real values must be aligned on even-numbered register boundaries (e.g., g0, g2, ...). Extended-real values must be aligned on register boundaries that are an integral multiple of four (e.g., g0, g4, ...).

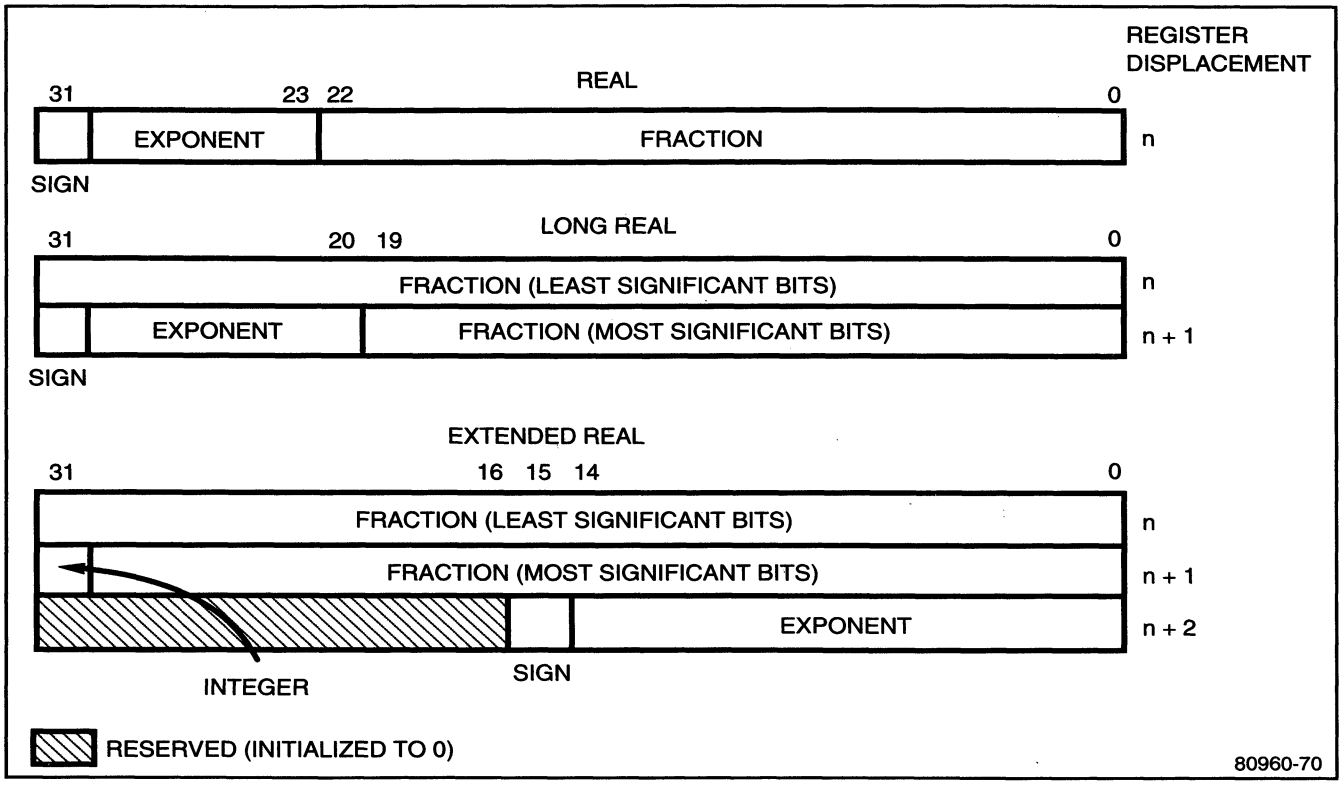

Figure 10-5: Storage of Real Values in Global and Local Registers

Real values in the floating-point registers are always in the extended-real format. When a real or long-real value is moved from global or local registers to a floating-point register, the processor automatically reformats it for the extended-real format.

Loading and Storing Floating-Point Values

Floating-point values are loaded from memory into global or local registers using the load (**ld**), load long (**ldl**), and load triple (**ldt**) instructions. Likewise, floating-point values in global or local registers are stored in memory using the store (**st**), store long (**stl**), and store triple (**stt**) instructions.

Loading a floating-point value into a floating-point register requires two steps (two instructions). First, a floating-point value must be loaded from memory into one or more global or local registers. Then, the value must be moved to the floating-point register using a move real (**movr**), move long-real (**movrl**), or move extended-real (**movre**) instruction.

A similar two-step procedure is required to store a value from a floating-point register into memory. The value must first be moved into one or more global or local registers (using a **movr**, **movrl**, or **movre** instruction), then stored in memory.

This two-step method for moving values from memory into floating-point registers and vice versa may seem a little cumbersome; however, in practice it generally is not. Floating-point registers are most often used to store and accumulate intermediate results of computations. The contents of these registers are not normally stored in memory.

For example, the following instruction

```
divr r3, r4, fp2
```

causes the real value in local register r4 to be divided by the value in r3, with the extended-real result stored in floating-point register fp2. Here, a move operation from the local registers to the floating-point registers is not required, since it is implicit in the divide operation.

Moving Floating-Point Values

Either the move instructions (**mov**, **movl**, or **movt**) or the move-real instructions (**movr**, **movrl**, or **movre**) can be used to move real values among global and local registers. The move real instructions are generally used to convert a real value from one format to another or for moving real values between the global or local registers and floating-point registers. The move instructions are used to move real values while keeping them in the same format.

When using the **movr** and **movrl** instructions to move floating-point numbers between the global or local registers and the floating-point registers, the processor automatically converts values from real and long-real format, respectively, into the extended-real format and vice versa.

For example, the following instruction

```
movr g3, fp1
```

causes a 32-bit, real value in global register g3 to be converted to 80-bit, extended-real format and placed in floating-point register fp1.

Going the opposite direction, the instruction

```
movrl fp0, r4
```

causes an extended-real value in floating-point register fp0 to be converted to 64-bit, long-real format and placed in local registers r4 and r5.

The **movre** instruction moves 80-bit, extended-real values between registers, without format conversion. When this instruction is used to move a value from three global or local registers to a floating-point register, the processor extracts the 80-bit value from the three word extended-real format. When moving a value from a floating-point register to global or local registers, the processor inserts the 80-bit value into the three registers in the three-word format.

Arithmetic Controls

The arithmetic controls are used extensively to control the arithmetic and faulting properties of floating-point operations. Table 10-4 shows the bits in the arithmetic controls that are used in floating-point operations. (The placement of these bits in the arithmetic controls is shown in Figure 2-3.)

The condition code flags are used to indicate the results of comparisons of real numbers, just as they are for integers and ordinals.

The arithmetic status field is used to record results from the classify real (**classr** and **classrl**) and remainder real (**remr** and **remrl**) instructions. These instructions are discussed later in this chapter.

The floating-point flags indicate exceptions to floating-point operations. Here, the term exception refers to a potentially undesirable operation (such as dividing a number by zero) or an undesirable result (such as underflow). The flags provide a means of recording the occurrence of specific exceptions.

The floating-point masks provide a method of inhibiting the processor from invoking a fault handler when an exception is detected.

Use of the floating-point flag and mask bits are discussed later in this chapter in the section titled "Exceptions and Fault Handling."

Table 10-4: Arithmetic Controls Used in Floating-Point Operations

Arithmetic Control Bits	Function
0 - 2	Condition code
3 - 6	Arithmetic status field
8	Integer overflow flag
12	Integer overflow mask
16	Floating overflow flag
17	Floating underflow flag
18	Floating invalid-operation flag
19	Floating zero-divide flag
20	Floating inexact flag
24	Floating overflow mask
25	Floating underflow mask
26	Floating invalid-operation mask
27	Floating zero-divide mask
28	Floating inexact mask
29	Normalizing mode flag
30 - 31	Rounding control

Normalizing Mode

The normalizing-mode flag specifies whether the processor operates in normalizing mode (set) or not (clear).

Normalizing mode is the most common mode of operation. Here, the processor operates on valid floating-point operands, regardless of whether they are normalized or denormalized values.

When the processor is not operating in normalizing mode, it signals a reserved-encoding exception whenever it encounters a denormalized floating-point value as a source operand. In either mode, denormalized numbers are produced if the underflow exception is masked.

There are no flag or mask bits in the arithmetic controls for this exception. When a reserved-encoding exception is detected, the processor generates a floating reserved-encoding fault and leaves the destination operand unchanged (i.e., no result is stored).

The unnormalized mode of operation is provided to allow unnormalized arithmetic to be simulated with software. Here, a fault handler routine can be used to perform unnormalized arithmetic whenever a reserved-encoding exception is signaled.

Rounding Control

Often the infinitely precise result of an arithmetic operation cannot be encoded exactly in the format of the destination operand. For example, the following value has a 24-bit fraction. The least-significant bit of this fraction (the underlined bit) cannot be encoded exactly in the real (32-bit) format:

$$1.0001\ 0000\ 1000\ 0011\ 1001\ 0\underline{1}1E_2\ 101$$

The processor must then round the result to one of the following two values:

$$1.0001\ 0000\ 1000\ 0011\ 1001\ 011E_2\ 101$$

$$1.0001\ 0000\ 1000\ 0011\ 1001\ 100E_2\ 101$$

A rounded result is called an inexact result. When an inexact result is produced, the floating-point inexact flag bit in the arithmetic controls is set.

The processor rounds results according to the destination format (real, long real, or extended real) and the setting of the rounding-mode flags of the arithmetic controls. Four types of rounding are allowed, as described in Table 10-5.

Table 10-5: Rounding Methods

Rounding Mode	Description	Value
Round up (toward +∞)	Rounded result is close to but no less than the infinitely precise result	00
Round down (toward -∞)	Rounded result is close to but no greater than the infinitely precise result	01
Round toward zero (Truncate)	Rounded result is close to but no greater in absolute value than the infinitely precise result	10
Round to nearest (even)	Rounded result is close to the infinitely precise result. If two values are equally close, the result is the even value (i.e., the one with the least-significant bit of zero)	11

When the infinitely precise result is between the largest positive finite value allowed in a particular format and +∞, the processor rounds the result as shown in Table 10-6.

Table 10-6: Rounding of Special Case Large Positive Numbers

Rounding Mode	Description	Value
Round up (toward +∞)	+∞	00
Round down (toward -∞)	Maximum, positive finite value	01
Round toward zero (Truncate)	Maximum, positive finite value	10
Round to nearest (even)	+∞	11

When the infinitely precise result is between the largest negative finite value allowed in a particular format and -∞, the processor rounds the result as shown in Table 10-7.

Table 10-7: Rounding of Special Case Large Negative Numbers

Rounding Mode	Description	Value
Round up (toward +∞)	Maximum, negative finite value	00
Round down (toward -∞)	-∞	01
Round toward zero (Truncate)	Maximum, negative finite value	10
Round to nearest (even)	-∞	11

The rounding modes have no effect on comparison operations, operations that produce exact results, or operations that produce NaN results.

The floating-point instructions allow a result to be stored in a shorter destination than the source operands. For example, the instruction

```
addr fp1, fp2, g5
```

produces a real (32-bit) result from two extended-real (80-bit) source operands. In all such operations, only one rounding error occurs: the error that occurs when rounding the infinitely precise result to the size of the destination format.

Technically, an operation which computes a narrow result from wide operands is in violation of the IEEE standard. However, systems that are designed to conform to the IEEE standard do not need to use this capability of the processor.

INSTRUCTION FORMAT

The instruction format for floating-point instructions is the same as for the other processor instructions. When programming in assembly language, an assembly language statement begins with an instruction mnemonic and is followed by from one to three operands. For example, the multiply-real instruction **mulr** might be used as follows:

```
mulr r8, r9, fp3
```

Here, real operands in local registers r8 and r9 are multiplied together and the result is stored in floating-point register fp3.

From the machine level point of view, all floating-point instructions use the REG format. Refer to Appendix B for details on the REG format instructions.

INSTRUCTION OPERANDS

Operands for floating-point instructions can be either floating-point literals or registers. The processor recognizes two encodings for floating-point literals: 0.0 and 1.0

All of the registers in the processor's execution environment (global registers g0 through g15, local registers r0 through r15, and floating-point registers fp0 through fp3) can be used as operands in floating-point instructions. (Of course, registers g15, r0, r1, and r2 would generally not be used for storing floating-point numbers, since they are reserved for stack management functions.)

When global or local registers are specified as operands, the instruction mnemonic (or opcode) determines how the values in these registers are interpreted. For example, there are two floating-point divide instructions: divide real (**divr**) and divide long real (**divrl**). When using the **divr** instruction, the processor assumes that global- or local-register operands contain real (32-bit) values. When using the **divrl** instruction, global- or local-register operands are assumed to contain long-real (64-bit) values.

With either instruction, floating-point registers (containing extended-real values) can also be used as operands.

Using floating-point registers as operands allows mixed format or mixed precision arithmetic to be performed with either real and extended-real values or long-real and extended-real values. Mixed-format operations with real and long-real values are not supported.

SUMMARY OF FLOATING-POINT INSTRUCTIONS

The processor's floating-point instructions consist of all instructions for which at least one operand is a real data type.

These instructions can be divided into the following groups:

- Data Movement
- Data-Type Conversion
- Basic Arithmetic
- Comparison and Classification
- Trigonometric
- Logarithmic and Exponential

The following sections give a brief overview of the instructions in each group. Detailed descriptions of the operations of these instructions are given in Chapter 9.

Data Movement

As has been described earlier in this chapter, the non-floating-point load and store instructions are used to move real values between registers and memory. Once in registers, the non-floating-point move instructions (**mov**, **movl**, and **movt**) are used to move real values between global and local registers without format conversion; whereas, the floating-point move instructions (**movr**, **movrl**, and **movre**) are used to move real values between global and local registers and floating-point registers.

The copy-sign-real-extended (**cpysre**) and copy-reverse-sign-real-extended (**cpyrsre**) instructions provide a means of copying the sign of one extended-real value to another, if one of the values is in a floating-point register. This operation is best performed on real and long-real values using the bit instructions **chkbit** and **alterbit**.

Data-Type Conversion

Two types of data-type conversions are provided: conversion from one floating-point format to another (e.g., real to extended real) and conversion between integer and real.

Conversion between floating-point formats is handled in either of two ways: explicitly by move instructions or implicitly by using the floating-point registers as operands in instructions.

As described earlier in this chapter, the **movr** instruction implicitly converts values from real to extended real, and vice versa, when moving values between global or local registers and floating-point registers. Likewise, the **movrl** instruction implicitly converts values from long real to extended real, and vice versa.

Conversion between real and long-real formats requires the use of both instructions. For example, the following two instructions convert a real value in global register g6 to a long-real value contained in g6 and g7, using a floating-point register for intermediate storage of the value:

```
movr g6, fp1
movrl fp1, g6
```

Implicit format conversion is also provided through the arithmetic, trigonometric, logarithmic, and exponential instructions. For example, the instruction

```
addr r4, r5, fp2
```

adds two real values together and produces an extended-real result.

The following six instructions allow conversion between integers and reals:

cvtir	convert integer to real
cvtilr	convert long integer to long real
cvtri	convert real to integer
cvtril	convert real to long integer
cvtzri	convert truncated real to integer
cvtzril	convert truncated real to long integer

Both the **cvtir** and **cvtilr** instructions can be used to convert an integer to an extended-real value by specifying that the result be placed in a floating-point register.

The convert real-to-integer instructions round off the real value to the nearest integer or long-integer value. For the **cvtri** and **cvtril** instructions, the rounding mode determines the direction the real number is rounded. For the convert truncated real-to-integer instructions (**cvtzri** and **cvtzril**), rounding is always toward zero. The latter two instructions are provided to allow efficient implementation of FORTRAN-like truncation semantics.

Extended-real values can be converted to integers by using a floating-point register as a source operand in either of the convert real-to-integer instructions.

Converting long-real values to integers requires two instructions, as in the following example:

```
movrl g6, fp3
cvtzri fp3, g6
```

The first instruction moves the long-real value to a floating-point register. The second instruction converts the extended-real value to an integer.

Basic Arithmetic

The following instructions perform the basic arithmetic operations specified in the IEEE standard:

addr	add real
addrl	add long real
subr	subtract real
subrl	subtract long real
mulr	multiply real
mulrl	multiply long real
divr	divide real
divrl	divide long real
remr	remainder real
remrl	remainder long real
roundr	round real
roundrl	round long real
sqrtr	square root real
sqrtrl	square root long real

The round instructions round the floating-point operand to its nearest integral (i.e., integer) value, based on the current rounding mode. These instructions perform a function similar to the convert real-to-integer instructions except that the result is in floating-point format.

Comparison, Branching, and Classification

Comparison of floating-point values differs from comparison of integers or ordinals because with floating-point values there are four, rather than the usual three, mutually exclusive relationships: less than, equal to, greater than, and unordered.

The unordered relationship is true when at least one of the two values being compared is a NaN. This additional relationship is required because, by definition, NaNs are not numbers, so they cannot have greater than, equal, or less than relationships with other floating-point values.

The following instructions are provided for comparing floating-point values:

cmpr	compare real
cmprl	compare long real
cmpor	compare ordered real
cmporl	compare ordered long real

All of these instructions set the condition code flags in the arithmetic controls to indicate the results of the comparison. With the compare instructions (**cmpr** and **cmprl**), the condition code flags are set to 000_2 for the unordered condition. With the compare ordered instructions (**cmpor** and **cmporl**), the condition code flags are set to 000_2 and an invalid-operation exception is signaled for the unordered condition.

Two branch instructions (**bo** and **bno**) allow conditional branching to be performed on an ordered or unordered condition, respectively. With these instructions, the processor checks the condition code flags for unordered (000_2) or ordered (111_2) and branches accordingly.

The classify-real instructions (**classr** and **classrl**) provide a means of determining the class of a floating-point value (i.e., zero, denormalized finite, normalized finite, ∞, SNaN, or QNaN). The result of this operation is stored in the arithmetic status field of the arithmetic controls.

Trigonometric

The following instructions provide four common trigonometric functions:

sinr	sine real
sinrl	sine long real
cosr	cosine real
cosrl	cosine long real
tanr	tangent real
tanrl	tangent long real
atanr	arctangent real
atanrl	arctangent long real

The arctangent instructions facilitate conversion from rectangular to polar coordinates.

Pi

The processor uses the following value for π in its computations:

$$\pi = 0.f * 2^e$$

where:

$$f = C90FDAA2\ 2168C234\ C_{16}$$

$$e = 2 \text{ if significand is } 0.f$$

(The spaces in the fraction above indicate 32-bit boundaries.)

This value has a 66-bit mantissa, which is 2 bits more than is allowed in the significand of an extended-real value. (Since 66 bits is not an even number of hex digits, two additional zeros have been added to the value so that it can be represented in a hexadecimal format. The least-significant hex digit (C_{16}) is thus 1100_2, where the two least significant bits represent bits 67 and 68 of the mantissa.)

If the results of computations that explicitly use π are to be used in the sine, cosine, or tangent instructions, the full 66-bit fraction for π should be used. This insures that the results are consistent with the argument-reduction algorithms that these instructions use. Using a rounded version of π can cause slight inaccuracies in result values, which if propagated through several calculations, might result in meaningless results.

A common method of representing the full 66-bit fraction of π is to store it as the sum of numbers. For example, the following two long-real values added together give the value for π shown above with the full 66-bit fraction:

$$\pi = high\pi + low\pi$$

where:

$$high\pi = 400921FB\ 54400000_{16}$$

$$low\pi = 3DD0B461\ 1A600000_{16}$$

Here *high*π gives the most significant 33 bits of π and *low*π gives the least significant 33 bits. Similar versions of π can also be written in the extended-real format.

When using this two-part π value in an algorithm, parallel computations should be performed on each part, with the results kept separate. When all the computations are complete, the two results can be added together to form the final result.

Logarithmic, Exponential, and Scale

The following instructions provide three different logarithmic functions, an exponential function, and a scale function:

logbnr	log binary real
logbnrl	log binary long real
logr	log real
logrl	log long real
logepr	log epsilon real
logeprl	log epsilon long real
expr	exponent real
exprl	exponent long real
scaler	scale real
scalerl	scale long real

These instructions are described in detail in Chapter 9. The following is a brief description of their functions.

The log binary instructions compute the IEEE recommended function *logb (X)*. The result is an integral value that is the binary log of X.

The log instructions compute the function *Y * log (X)*, where the log of X is the base-2 logarithm.

The log epsilon instructions compute the function *Y * log (X + 1)*, where the log of X + 1 is a base-2 logarithm.

The exponent instructions compute the value $2^X - 1$.

The scale instructions perform a multiplication of a floating-point value by a power of 2.

Arithmetic Versus Nonarithmetic Instructions

The floating-point instructions can be divided into two groups: arithmetic and nonarithmetic. Arithmetic instructions are those that are sensitive to real values, meaning that they distinguish among NaN, ∞, normalized finite, denormalized finite, and zero values.

All but five of the floating-point instructions are arithmetic. The five nonarithmetic instructions are move-real extended (**movre**), copy-sign real extended (**cpysre**), copy-reversed-sign real extended (**cpyrsre**), and classify real (**classr** and **classrl**). These nonarithmetic instructions are insensitive to real values and cannot generate floating-point exceptions or faults.

This distinction between arithmetic and nonarithmetic instructions is important because floating-point exceptions and faults can be signaled only during the execution of arithmetic instructions.

OPERATIONS ON NaNs

As was described earlier in this chapter, the processor supports two types of NaNs: QNaN and SNaN. An SNaN is any NaN value with its most-significant fraction bit set to 0 and at least one other fraction bit set to 1. (If all the fraction bits are set to 0, the value is an ∞.) A QNaN is any NaN value with the most-significant fraction bit set to 1. The sign bit of a NaN is not interpreted.

In general, when a QNaN is used in one or more arithmetic floating-point instructions, it is allowed to propagate through a computation. An SNaN on the other hand causes a floating invalid-operation exception to be signaled.

The floating invalid-operation exception has a flag and a mask bit associated with it in the arithmetic controls. The mask bit determines how the processor handles an SNaN value. If the floating invalid-operation mask bit is set, the SNaN is converted to a QNaN by setting the most significant fraction bit of the value to a 1. The result is then stored in the destination and the floating invalid-operation flag is set. If the invalid operation mask is clear, a floating invalid-operation fault is signaled and no result is stored in the destination.

When the result is a QNaN, the format of the result is as shown in Table 10-8, depending on the form of the source operands.

Table 10-8: Format of QNaN Results

Source Operands	QNaN Result
Only one operand is NaN, destination is same width	QNaN version of NaN source
Only one operand is NaN, destination is longer	QNaN version of NaN source, with fraction extended with zeros
Only one operand is NaN, destination is shorter	QNaN version of NaN source, with fraction truncated
Both operands are NaNs	QNaN version of source whose fraction field has greatest magnitude, with fraction extended or truncated as described above

In some cases, a QNaN result is returned when none of the source operands are NaNs. Here, a standard QNaN is returned. The significand for the standard QNaN is as follows:

1.1000...00

(For real and long-real destinations, the integer bit will be an implied 1.)

Other than the rules specified above, software is free to use the other bits of a NaN for any purpose.

EXCEPTIONS AND FAULT HANDLING

Occasionally, a floating-point instruction can result in an exception being signaled. The processor recognizes six floating-point exceptions:

- Floating Reserved Encoding
- Floating Invalid Operation
- Floating Zero Divide
- Floating Overflow
- Floating Underflow
- Floating Inexact

These exceptions can be divided into two categories:

1. Situations in which one or more source operands are inappropriate for an operation and would cause an exception to be signaled.
2. Situations in which the result of an operation is exceptional.

The reserved encoding, invalid operation, and division-by-zero exceptions fall in the first category; the overflow, underflow, and inexact exceptions fall in the second category.

Except for the floating reserved-encoding exception, each of these exceptions has a flag and a mask bit associated with it in the arithmetic controls. When an exception condition occurs, the processor performs one of the following operations:

- If the mask bit for the exception is set, the flag for the exception is set and instruction execution continues, substituting a default value in place of the result.
- If the mask bit for the exception is clear, the flag for the exception is not set and a floating-point arithmetic fault is raised. The processor then stores diagnostic information in the fault information area and diverts instruction execution to a fault handler.

Since the floating reserved-encoding exception does not have a flag or mask bit, it always results in a fault.

NOTE

The floating-point exception flags are "sticky," which means that the processor does not implicitly clear them while carrying out floating-point operations. They may be cleared by software.

Fault Handler

As is described in Chapter 6, when a floating-point fault is signaled, the processor calls a single fault handler. This fault handler determines how to handle the specific fault subtype by interpreting the floating-point exception flags and the information in the fault record.

Floating-Reserved-Encoding Exception

A reserved-encoding exception occurs as a result of either of the following two conditions:

- When a reserved encoding is used as an operand in a floating-point instruction, or
- When a denormalized value is used as an operand in a floating-point instruction and the normalizing-mode bit in the arithmetic controls is clear.

The first condition is rare. It can only occur if a program presents an extended-real value to the processor that has a zero j-bit (integer part) and a non-zero biased exponent.

The second condition was discussed earlier in this chapter in the section titled "Normalizing Mode." This condition is also rare, since the vast majority of programs run with the normalizing mode enabled.

There is neither a flag nor a mask bit for this exception. When a reserved-encoding exception occurs, the processor raises a floating-reserved-encoding fault and does not store a result.

Floating-Invalid-Operation Exception

The invalid-operation exception indicates that one of the source operands is inappropriate for the type of operation being performed. The following conditions cause this exception to be signaled:

- Any arithmetic operation on an SNaN
- Addition of infinities of unlike sign
- Subtraction of infinities of like sign
- Multiplication of zero by ∞
- Division of zero by zero or ∞ by ∞
- Remainder of x by y, if y is zero or x is ∞
- Square root of a negative, nonzero value
- Conversion of a NaN from floating-point format to integer format
- Sine, cosine, or tangent of ∞
- y * log (x), if:
  - x is negative and nonzero,
  - y is zero and x is ∞,
  - y and x are zero, or
  - y is ∞ and x is 1
- Log epsilon of (y, x), if y is ∞ and x is 0
- Compare ordered, if a source operand is a NaN

When a floating-invalid-operation exception occurs and its mask is set, the following occurs:

- When the result is a floating-point value, the standard QNaN value is stored in the destination and the floating-invalid-operation flag is set.
- If one of the operands is a NaN and the result is a floating-point value, a NaN is stored in the destination and the floating-invalid operation flag is set. (A discussion of how the processor handles NaNs was provided earlier in this chapter in the section titled "Operations on NaNs.")
- When the result is an integer, the maximum negative integer is stored in the destination and the floating-invalid-operation flag is set.

When the mask is clear, no result is stored; the floating-invalid-operation flag is not set; and the floating-invalid-operation fault is signaled.

Floating-Zero-Divide Exception

The floating-zero-divide exception is signaled when an exact non-finite result would be produced from finite operands. (Note that a different exception, overflow, is signaled when an infinite result is produced inexactly from finite operands.) The most common example of this exception is a division operation, where the divisor is zero and the dividend is a nonzero, finite value.

When the floating-zero-divide mask is set: a correctly signed ∞ is stored in the destination and the floating-zero-divide flag is set. When the mask is clear, no result is stored; the floating-zero-divide flag is not set; and a floating-zero-divide fault is signaled.

Floating-Overflow Exception

The overflow exception occurs when the infinitely precise result of a floating-point instruction exceeds the largest allowable finite value for the specified destination format. For example, if the destination format is real (32 bits), overflow occurs when the infinitely precise result falls outside the range $-1.0 * 2^{128}$ to $1.0 * 2^{128}$ (exclusive), where 128 is the unbiased exponent of the result. For long-real (64 bits) values, the overflow threshold range is $-1.0 * 2^{1024}$ to $1.0 * 2^{1024}$; for extended-real (80 bits) values, it is $-1.0 * 2^{16384}$ to $1.0 * 2^{16384}$.

The floating-overflow exception cannot occur on a conversion from floating-point format to integer format (although an integer overflow exception can occur).

Floating-Overflow Mask Set

When the floating-overflow mask is set, the largest representable real number or ∞ is stored in the destination and the floating-overflow flag is set. The current rounding mode determines the result. See Tables 10-6 and 10-7.

Floating-Overflow Mask Clear

When the mask is clear: no result is stored in the destination and the floating-overflow flag is not set. Instead, the processor stores the result in extended-real format in the fault information area. The fraction of the extended-real value is rounded to the instruction's destination precision. For example, if the destination operand's format is real (32 bits), the extended-real fraction is rounded to 23 bits, with the 40 least-significant bits filled with zeros.

If the exponent exceeds the range of the extended-real format (16383 unbiased), then the exponent is divided by 2^{24576} and a flag (bit 1 of the fault flags byte or override flags byte) is set in the fault information area to indicate that the exponent has been bias adjusted. After this fault information is stored, a floating-overflow fault is signaled.

When using the scale instructions (**scaler** or **scalerl**), massive overflow can occur, where the infinitely precise result is too large to be represented, even with a bias-adjusted exponent. Here, a properly signed ∞ or *dst* format's largest representable number is stored in the fault record.

Floating-Underflow Exception

An underflow condition occurs when the infinitely precise result of a floating-point instruction is less than the smallest possible normalized, finite value for the specified destination format. For example, for the real (32-bit) format, underflow occurs when an infinitely precise result falls in the range $-1.0 * 2^{-126}$ to $1.0 * 2^{-126}$ (exclusive), where -126 is the unbiased exponent. For long-real (64 bits) values, the underflow threshold range is $-1.0 * 2^{-1022}$ to $1.0 * 2^{-1022}$; for extended-real (80 bits) values, it is $-1.0 * 2^{-16382}$ to $1.0 * 2^{-16382}$.

When a floating-underflow condition occurs, the setting of the floating-underflow mask determines how the processor handles the condition.

Refer to the section later in this chapter titled "Floating-Point Underflow Condition" for more information on the interaction of the floating underflow and inexact exceptions.

Floating-Underflow Mask Set

If the mask is set when an underflow condition occurs, the processor goes ahead and denormalizes the result. Then if the result is exact, it is stored in the destination and the floating-underflow exception is not signaled, nor is the floating-underflow flag set. If, on the other hand, the denormalized result is inexact, the floating-underflow flag is set and the processor goes on to handle the inexact condition as described in the next section.

Floating-Underflow Mask Clear

If the floating-underflow mask is clear when an underflow condition occurs, no result is stored in the destination and the floating-underflow flag is not set. Instead, the processor stores the result in extended-real format in the fault information area, with the fraction of the extended-real value rounded to the instruction's destination precision. For example, if the destination precision is real (23-bit fraction), the 40 least-significant bits of the fraction are set to 0.

If the exponent of the value stored is less than the minimum allowable value in the extended-real format (-16,382 unbiased), then the exponent is multiplied by 2^{24576} and a flag (bit 1 of the fault or override flags byte) is set in the fault information area to indicate that the exponent has been bias adjusted. After this information is stored, a floating-underflow fault is signaled.

The scale instructions can cause massive underflow to occur, where the infinitely precise result is too small to be represented, even with a bias-adjusted exponent. Here, the infinitely precise result is denormalized and then rounded according to the rounding mode. The result of rounding produces either a properly signed zero or the smallest denormalized number, which is then stored in the fault record.

Floating-Inexact Exception

The floating-inexact exception occurs when an infinitely precise result cannot be encoded in the format specified for the destination operand. Either of the following two conditions can cause an inexact exception to be signaled:

- When a result is rounded and the result is not exact
- When overflow occurs and the floating-overflow mask is set

If the floating-inexact mask is set when an inexact condition occurs and an unmasked overflow or underflow condition does not occur, the rounded result is stored in the destination and the floating-inexact flag is set. The current rounding mode determines the method used to round the result.

If the floating-inexact mask is clear when an inexact condition occurs, the floating-inexact flag is not set and one of the following operations is carried out:

- If only the inexact condition has occurred, or an inexact condition has occurred along with masked overflow or underflow, the processor stores the rounded result in the specified destination, then raises a floating-inexact fault.
- If the inexact condition occurs along with unmasked overflow or underflow, no result is stored in the destination. Instead, the processor stores the result in extended-real format in the fault information area, as described for the floating overflow and underflow exceptions, then raises a fault with the floating-inexact bit in fault sub-type set.

Refer to the following section for more information on the interaction of the floating underflow and inexact exceptions.

Floating-Point Underflow Condition

Two aspects of underflow are important in numeric processing: the "tininess" of a number and "loss of accuracy." A result is tiny when it is nonzero and its exponent is between $\pm 2^{Emin}$, where E_{min} is the smallest unbiased exponent allowed in the destination format. For example, if the destination format is long-real (64-bit format), a result is tiny if it is nonzero and in the range of $+1 * 2^{-1022}$ to $-1 * 2^{-1022}$. The ability to detect a tiny result is important because such a result may cause an exception to be signaled in a later operation (e.g., overflow on a division).

Loss of accuracy occurs when a tiny result is approximated as part of the denormalization process so that it will fit into the destination format.

In the 80960SB processor, tininess is detected after rounding as an underflow condition. Loss of accuracy is detected as an inexact condition.

The algorithm in Figure 10-6 shows how the processor responds to these two conditions, when a floating-point operation produces a tiny result.

An important point to note in this algorithm is that if the underflow mask is set, an underflow exception is signaled only if the denormalized result is inexact. If the denormalized number is exact, no flags are set and no faults are signaled.

```
generate infinitely precise result  # exponent and significand;
   if exponent < underflow threshold
      then
         if underflow fault mask clear
            then
               goto underflow fault handler;
               exit algorithm;
            else generate denormalized number
               if denormalized significand equals infinitely precise significand
                  then
                    store denormalized result in destination;
                    # no underflow is signaled;
                  else
                    set underflow flag in AC;
                    if inexact fault mask is clear
                       then
                         goto inexact fault handler;
                         exit algorithm;
                       else
                         set inexact flag in AC;
                         store denormalized result in destination;
                    end if;
               end if;
         end if;
      else
         if infinitely precise result is inexact
            then
               if inexact fault mask is clear
                  then
                    goto inexact fault handler;
                    exit algorithm;
                  else
                    set inexact flag in AC;
                    store normalized result in destination;
               end if;
            else
              store normalized result in destination;
          end if;
       end if;
       exit algorithm
```

Figure 10-6: Interaction of Floating Underflow and Inexact Exceptions