SECTION 3
OPERAND CONVENTIONS

This section describes the conventions used for storing values in registers and memory, accessing PowerPC registers, and representing data in these registers.

3.1 Data Alignment and Memory Organization

Bytes in memory are numbered consecutively starting with zero. Each number is the address of the corresponding byte.

Memory operands can be bytes, half words, words, or double words, or, for the load/store multiple and move assist instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction.

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length. In other words, the “natural” address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is misaligned.

Operands for single-register memory access instructions have the characteristics shown in **Table 3-1**. (Although not permitted as memory operands, quad words are shown because quad-word alignment is desirable for certain memory operands.)

Table 3-1 Memory Operands

Operand	Length	ADDR[28:31] if aligned
Byte	8 bits	xxxx[1]
Half word	2 bytes	xxx0[1]
Word	4 bytes	xx00[1]
Double word	8 bytes	x000[1]
Quad word	16 bytes	0000

NOTES:
1. An “x” in an address bit position indicates that the bit can be zero or one independent of the state of other bits in the address.

The concept of alignment is also applied more generally to data in memory. For example, 12 bytes of data are said to be word-aligned if the address of the lowest-numbered byte is a multiple of four.

Some instructions require their memory operands to have certain alignments. In addition, alignment may affect performance. For single-register memory access instructions, the best performance is obtained when memory operands are aligned. Additional effects of data placement on performance are described in **SECTION 7 INSTRUCTION TIMING**.

Instructions are four bytes long and word-aligned.

3.2 Byte Ordering

There are two practical ways to order the four bytes in a word: big-endian and little-endian. The PowerPC architecture supports both these formats.

Big-endian ordering assigns the lowest address to the highest-order eight bits of the scalar. This is called big-endian because the big end of the scalar, considered as a binary number, comes first in memory.

Little-endian byte ordering assigns the lowest address to the lowest-order (rightmost) eight bits of the scalar. The little end of the scalar, considered as a binary number, comes first in memory.

Two bits in the MSR specify byte ordering: LE (little-endian mode) and ILE (exception little-endian mode). The LE bit specifies the endian mode in which the processor is currently operating, and ILE specifies the mode to be used when the system error handler is invoked. That is, when an exception occurs, the ILE bit (as set for the interrupted process) is copied into MSR[LE] to select the endian mode for the context established by the exception. For both bits, a value of zero specifies big-endian mode and a value of one specifies little-endian mode.

The default byte and bit ordering is big-endian, as shown in **Figure 3-1**. After a hard reset, the hard reset handler (using the **mtspr** instruction) can select little-endian mode for normal operation and exception processing by setting the LE and ILE bits, respectively, in the MSR.

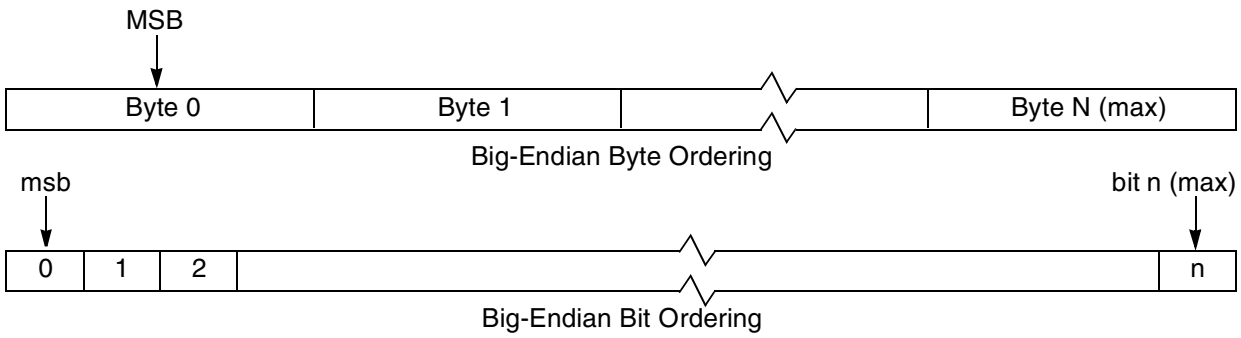

Figure 3-1 Big-Endian Byte Ordering

If individual data items were indivisible, the concept of byte ordering would be unnecessary. The order of bits or groups of bits within the smallest addressable unit of memory is irrelevant, because nothing can be observed about such order. Order matters only when scalars, which the processor and programmer regard as indivisible quantities, can be made up of more than one addressable units of memory.

For a device in which the smallest addressable unit is the 64-bit double word, there is no question of the order of bytes within double words. All transfers of individual scalars between registers and system memory are of double words. A subset of the 64-bit scalar (for example, a byte) is not addressable in memory. As a result, to access any subset of the bits of a scalar, the entire 64-bit scalar must be accessed, and when a memory location is read, the 64-bit value returned is the 64-bit value last written to that location.

For PowerPC processors, the smallest addressable memory unit is the byte (8 bits), and scalars are composed of one or more sequential bytes. When a 32-bit scalar is moved from a register to memory, it occupies four consecutive byte addresses, and a decision must be made regarding the order of these bytes in these four addresses.

3.2.1 Structure Mapping Examples

The following C programming example contains an assortment of scalars and one character string. The value presumed to be in each structure element is shown in hexadecimal in the comments and are used below to show how the bytes that comprise each structure element are mapped into memory.

```
struct {
          int       a;       /* 0x1112_1314                 word           */
          double    b;       /* 0x2122_2324_2526_2728       doubleword     */
          char *    c;       /* 0x3132_3334                 word           */
          char      d[7];    /* 'A','B','C','D','E','F','G' array of bytes */
          short     e;       /* 0x5152                      halfword       */
          int       f;       /* 0x6162_6364                 word           */
} s;
```

Note that the C structure mapping introduces padding (skipped bytes) in the map in order to align the scalars on their proper boundaries — four bytes between *a* and *b*, one byte between *d* and *e*, and two bytes between *e* and *f*. Both big- and little-endian mappings use the same amount of padding.

3.2.1.1 Big-Endian Mapping

The big-endian mapping of a structure *S* is shown in **Figure 3-2**. Addresses are shown in hexadecimal at the left of each double word and in small figures below each byte. The content of each byte, as shown in the preceding C programming example, is shown in hexadecimal as characters for the elements of the string.

00	11 00	12 01	13 02	14 03	04	05	06	07
08	21 08	22 09	23 0A	24 0B	25 0C	26 0D	27 0E	28 0F
10	31 10	32 11	33 12	34 13	‘A’ 14	‘B’ 15	‘C’ 16	‘D’ 17
18	‘E’ 18	‘F’ 19	‘G’ 1A	1B	51 1C	52 1D	1E	1F
20	61 20	62 21	63 22	64 23				

Figure 3-2 Big-Endian Mapping of Structure *S*

3.2.1.2 Little-Endian Mapping

Figure 3-3 shows the structure, *S*, using little-endian mapping. Double words are laid out from right to left.

07	06	05	04	11 03	12 02	13 01	14 00
21 0F	22 0E	23 0D	24 0C	25 0B	26 0A	27 09	28 08
‘D’ 17	‘C’ 16	‘B’ 15	‘A’ 14	31 13	32 12	33 11	34 10
1F	1E	51 1D	52 1C	1B	‘G’ 1A	‘F’ 19	‘E’ 18
				61 23	62 22	63 21	64 20

Figure 3-3 Little-Endian Mapping of Structure *S*

3.2.2 Data Memory in Little-Endian Mode

This section describes how data in memory is stored and accessed in little-endian mode.

3.2.2.1 Aligned Scalars

For load and store instructions, the effective address is computed as specified in the instruction descriptions in **SECTION 4 ADDRESSING MODES AND INSTRUCTION SET SUMMARY**. The effective address is modified as shown in **Table 3-2** before it is used to access memory.

Table 3-2 EA Modifications

Data Width (Bytes)	EA Modification
8	No change
4	XOR with 0b100
2	XOR with 0b110
1	XOR with 0b111

The modified EA is passed to the main memory and the specified width of the data is transferred between a GPR or FPR and the addressed memory locations (as modified). The effective address modification makes it appear to the processor that individual aligned scalars are stored as little-endian, when in fact they are stored as big-endian but in different bytes within double words from the order in which they are stored in big-endian mode.

Taking into account the preceding description of EA modifications, in little-endian mode structure *S* is placed in memory as shown in **Figure 3-4**.

00					11	12	13	14
	00	01	02	03	04	05	06	07
08	21	22	23	24	25	26	27	28
	08	09	0A	0B	0C	0D	0E	0F
10	‘D’	‘C’	‘B’	‘A’	31	32	33	34
	10	11	12	13	14	15	16	17
18			51	52		‘G’	‘F’	‘E’
	18	19	1A	1B	1C	1D	1E	1F
20					61	62	63	64
	20	21	22	23	24	25	26	27

Figure 3-4 PowerPC Little-Endian Structure *S* in Memory

Because of the modifications on the EA, the same structure *S* appears to the processor to be mapped into memory this way when LM = 1 (little-endian enabled). This is shown in **Figure 3-5**.

07	06	05	04	11 03	12 02	13 01	14 00
21 0F	22 0E	23 0D	24 0C	25 0B	26 0A	27 09	28 08
‘D’ 17	‘C’ 16	‘B’ 15	‘A’ 14	31 13	32 12	33 11	34 10
1F	1E	51 1D	52 1C	1B	‘G’ 1A	‘F’ 19	‘E’ 18
				61 23	62 22	63 21	64 20

Figure 3-5 PowerPC Little-Endian Structure *S* as Seen by Processor

Note that as seen by the program executing in the processor, the mapping for the structure *S* is identical to the little-endian mapping shown in **Figure 3-3**. From outside of the processor, the addresses of the bytes making up the structure *S* are as shown in **Figure 3-4**. These addresses match neither the big-endian mapping of **Figure 3-2** or the little-endian mapping of **Figure 3-3**. This must be taken into account when performing I/O operations in little-endian mode; this is discussed in **3.2.4 Input/Output in Little-Endian Mode**.

3.2.2.2 Misaligned Scalars

Performing an XOR operation on the low-order bits of the address of a scalar requires the scalar to be aligned on a boundary equal to a multiple of its length. When executing in little-endian mode (LM = 1), the RCPU takes an alignment exception whenever a load or store instruction is issued with a misaligned EA, regardless of whether such an access could be handled without causing an exception in big-endian mode (LM = 0).

The PowerPC architecture defines that half words, words, and double words be placed in memory such that the little-endian address of the lowest-order byte is the EA computed by the load or store instruction; the little-endian address of the next-lowest-order byte is one greater, and so on. **Figure 3-6** shows a four-byte word stored at little-endian address 5. The word is presumed to contain the binary representation of 0x1112 1314.

12 07	13 06	14 05	04	03	02	01	00	00
0F	0E	0D	0C	0B	0A	09	11 08	08

Figure 3-6 PowerPC Little-Endian Mode, Word Stored at Address 5

Figure 3-7 shows the same word stored by a little-endian program, as seen by the memory system (assuming big-endian mode).

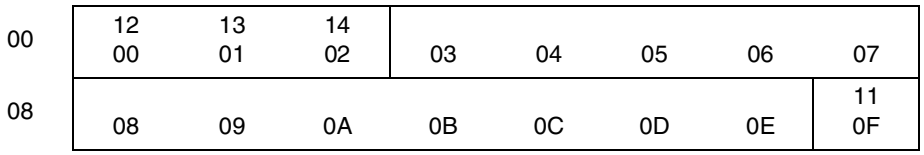

Figure 3-7 Word Stored at Little-Endian Address 5 as Seen by Big-Endian Addressing

NOTE

The misaligned word in this example spans two double words. The two parts of the misaligned word are not contiguous in the big-endian addressing space.

An implementation may choose to support only a subset of misaligned little-endian memory accesses. For example, misaligned little-endian accesses contained within a single double word may be supported, while those that span double words may cause alignment exceptions.

3.2.2.3 String Operations

The load and store string instructions, listed in **Table 3-3**, cause alignment exceptions when they are executed in little-endian mode.

Table 3-3 Load/Store String Instructions

Mnemonic	Description
lswi	Load String Word Immediate
lswx	Load String Word Indexed
stswi	Store String Word Immediate
stswx	Store String Word Indexed
lscbx	Load String and Compare Byte Indexed

String accesses are inherently misaligned; they transfer word-length quantities between memory and registers, but the quantities are not necessarily aligned on word boundaries.

NOTE

The system software must determine whether to emulate the excepting instruction or treat it as an illegal operation.

3.2.2.4 Load and Store Multiple Instructions

The load and store multiple instructions shown in **Table 3-4** cause alignment exceptions when executed in little-endian mode.

Table 3-4 Load/Store Multiple Instructions

Mnemonic	Instruction
lmw	Load Multiple Word
stmw	Store Multiple Word

Although the words addressed by these instructions are on word boundaries, each word is in the half of its containing double word opposite from where it would be in big-endian mode. Note that the system software must determine whether to emulate the excepting instruction or treat it as an illegal operation.

3.2.3 Instruction Memory Addressing in Little-Endian Mode

Each PowerPC instruction occupies 32 bits (one word) of memory. PowerPC processors fetch and execute instructions as if the current instruction address had been advanced one word for each sequential instruction. When operating in little-endian mode, the address is modified according to the little-endian rule for fetching word-length scalars; that is, it is XORed with 0b100. A program is thus an array of little-endian words with each word fetched and executed in order (not including branches).

Consider the following example:

```
loop:
        cmplwi   r5,0
        beq      done
        lwzux    r4, r5, r6
        add      r7, r7, r4
        subi     r5, 1
        b        loop
done:
        stw      r7, total
```

Assuming the program starts at address 0, these instructions are mapped into memory for big-endian execution as shown in **Figure 3-8**.

00	loop: cmplwi r5, 8				beq done			
	00	01	02	03	04	05	06	07
08	lwzux r4, r5, r6				add r7, r7, r4			
	08	09	0A	0B	0C	0D	0E	0F
10	subi r5, 1				b loop			
	10	11	12	13	14	15	16	17
18	done: stw r7, total							
	18	19	1A	1B	1C	1D	1E	1F

Figure 3-8 PowerPC Big-Endian Instruction Sequence as Seen by Processor

If this same program is assembled for and executed in little-endian mode, the mapping seen by the processor appears as shown in **Figure 3-9**.

Each machine instruction appears in memory as a 32-bit integer containing the value described in the instruction description, regardless of whether the processor is operating in big- or little-endian mode. This is because scalars are always mapped in memory in big-endian byte order.

beq done				loop: cmplwi				00
07	06	05	04	03	02	01	00	
add r7, r7, r4				lwzux r4, r5, r6				08
0F	0E	0D	0C	0B	0A	09	08	
b loop				subi r5, 1				10
17	16	15	14	13	12	11	10	
				done: stw r7, total				18
1F	1E	1D	1C	1B	1A	19	18	

Figure 3-9 PowerPC Little-Endian Instruction Sequence as Seen by Processor

When little-endian mapping is used, all references to the instruction stream must follow little-endian addressing conventions, including addresses saved in system registers when the exception is taken, return addresses saved in the link register, and branch displacements and addresses.

- An instruction address placed in the link register by branch and link, or an instruction address saved in an SPR when an exception is taken is the address that a program executing in little-endian mode would use to access the instruction as a word of data using a load instruction.
- An offset in a relative branch instruction reflects the difference between the addresses of the instructions, where the addresses used are those that a pro-

gram executing in little-endian mode would use to access the instructions as data words using a load instruction.

- A target address in an absolute branch instruction is the address that a program executing in little-endian mode would use to access the target instruction as a word of data using a load instruction.

3.2.4 Input/Output in Little-Endian Mode

Input/output operations transfer a byte stream on both big- and little-endian systems. For a PowerPC system running in big-endian mode, both the processor and the memory subsystem recognize the same byte as byte 0. However, this is not true for a PowerPC system running in little-endian mode because of the modification of the three low-order bits when the processor accesses memory.

In order for I/O transfers in little-endian mode to appear to transfer bytes properly, they must be performed as if the bytes transferred were accessed one at a time, using the little-endian address modification appropriate for the single-byte transfers (XOR the bits with 0b111). This does not mean that I/O on little-endian PowerPC machines must be done using only one-byte-wide transfers. Data transfers can be as wide as desired, but the order of the bytes within double words must be as if they were fetched or stored one at a time.

3.3 Floating-Point Data

This subsection describes how floating-point data is represented in floating-point registers and in memory.

3.3.1 Floating-Point Data Format

The PowerPC architecture defines the representation of a floating-point value in two different binary, fixed-length formats: a 32-bit format for a single-precision floating-point value or a 64-bit format for a double-precision floating-point value. Data in memory may use either the single-precision or double-precision format. Floating-point registers use the double-precision format.

The length of the exponent and the fraction fields differ between these two precision formats. The structure of the single-precision format is shown in **Figure 3-10**; the structure of the double-precision format is shown in **Figure 3-11**.

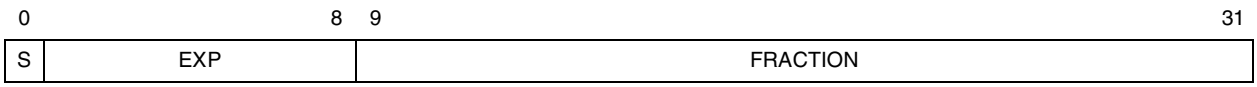

Figure 3-10 Floating-Point Single-Precision Format

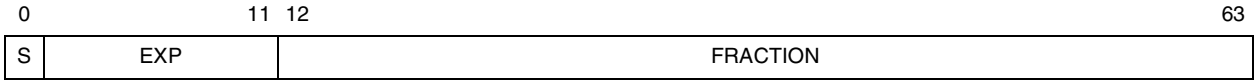

Figure 3-11 Floating-Point Double-Precision Format

Values in floating-point format consist of three fields:

- S (sign bit).
- EXP (exponent + bias)
- FRACTION (fraction)

If only a portion of a floating-point data item in memory is accessed, as with a load or store instruction for a byte or half word (or word in the case of floating-point double-precision format), the value affected depends on whether the PowerPC system is using big- or little-endian byte ordering, which is described in **3.2 Byte Ordering**. Big-endian mode is the default.

The significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is a one for normalized numbers and a zero for denormalized numbers in the unit bit position (that is, the first bit to the left of the binary point). Parameters for the two floating-point formats are listed in **Table 3-5**.

Table 3-5 IEEE Floating-Point Fields

Parameter	Single-Precision	Double-Precision
Exponent bias	+127	+1023
Maximum exponent (unbiased)	+127	+1023
Minimum exponent	–126	–1022
Format width	32 bits	64 bits
Sign width	1 bit	1 bit
Exponent width	8 bits	11 bits
Fraction width	23 bits	52 bits
Significand width	24 bits	53 bits

The exponent is expressed as an 8-bit value for single-precision numbers or an 11-bit value for double-precision numbers. These bits hold the biased exponent; the true value of the exponent can be determined by subtracting 127 for single-precision numbers and 1023 for double-precision values. This is shown in **Figure 3-12**. Note that using a bias eliminates the need for a sign bit. The highest-order bit is

used both to generate the number, and is an implicit sign bit. Note also that two values are reserved — all bits set indicates that the number is an infinity or NaN and all bits cleared indicates that the number is either zero or denormalized.

3.3.2 Value Representation

The PowerPC architecture defines numerical and non-numerical values representable within single- and double-precision formats. The numerical values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The non-numerical values representable are the positive and negative infinities and the NaNs. The positive and negative infinities are adjoined to the real numbers but are not numbers themselves, and the standard rules of arithmetic do not hold when they appear in an operation. They are related to the real numbers by “order” alone. It is possible, however, to define restricted operations among numbers and infinities as defined in the following paragraphs. The relative location on the real number line for each of the defined entities is shown in **Figure 3-13**.

	Biased Exponent (binary)	Single-Precision (unbiased)	Double-Precision (unbiased)
	11. 11	Reserved for Infinities and NaNs	
Positive	11. 10	+127	+1023
	11. 01	+126	+1022
	.	.	.
	.	.	.
	.	.	.
	10. 00	1	1
Zero →	01. 11	0	0
	01. 10	–1	–1
	.	.	.
Negative	.	.	.
	.	.	.
	00. 01	–126	–1022
	00. 00	Reserved for Zeros and Denormalized Numbers	

Figure 3-12 Biased Exponent Format

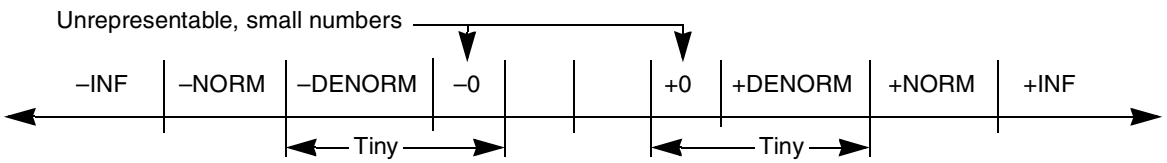

Figure 3-13 Approximation to Real Numbers

The positive and negative NaNs are not related to the numbers or ±× by order or value, but they are encodings that convey diagnostic information such as the representation of uninitialized variables.

Table 3-6 describes each of the floating-point formats.

Table 3-6 Recognized Floating-Point Numbers

Sign Bit	Exponent (Biased)	Leading Bit	Mantissa	Value
0	Maximum	x	Non-zero	+NaN
0	Maximum	x	Zero	+Infinity
0	0 < Exponent < Maximum	1	Non-zero	+Normalized
0	0	0	Non-zero	+Denormalized
0	0	0	Zero	+0
1	0	0	Zero	–0
1	0	0	Non-zero	–Denormalized
1	0 < Exponent < Maximum	1	Non-zero	–Normalized
1	Maximum	x	Zero	–Infinity
1	Maximum	x	Non-zero	–NaN

3.3.3 Normalized Numbers (±NORM)

The values for normalized numbers have a biased exponent value in the range:

- 1–254 in single-precision format
- 1–2046 in double-precision format

The implied unit bit is one. Normalized numbers are interpreted as follows:

$$NORM = (-1)^{s} \times 2^{E} \times (1.fraction)$$

where (s) is the sign, (E) is the unbiased exponent and (1.fraction) is the significand composed of a leading unit bit (implied bit) and a fractional part. The format for normalized numbers is shown in Figure 3-14.

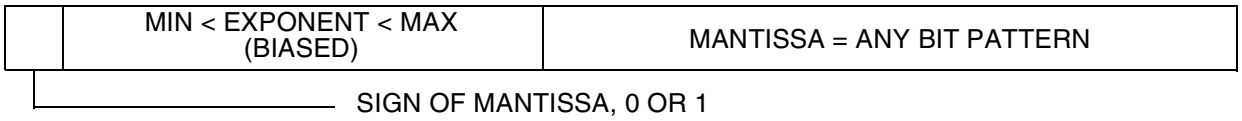

Figure 3-14 Format for Normalized Numbers

The ranges covered by the magnitude (M) of a normalized floating-point number are approximately equal to the following:

Single-precision format:

$1.2x10^{-38}$ ð M ð $3.4x10^{38}$

Double-precision format:

$2.2x10^{-308}$ ð M ð $1.8x10^{308}$

3.3.4 Zero Values (±0)

Zero values have a biased exponent value of zero and a fraction value of zero. This is shown in **Figure 3-15**. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (that is, comparison regards +0 as equal to –0).

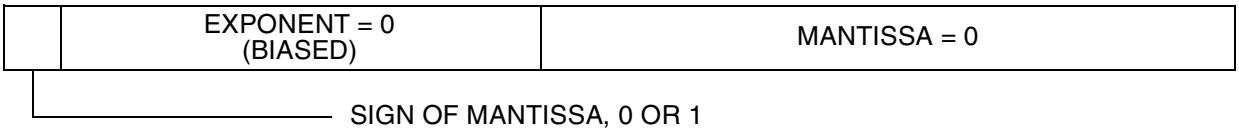

Figure 3-15 Format for Zero Numbers

3.3.5 Denormalized Numbers (±DENORM)

Denormalized numbers have a biased exponent value of zero and a non-zero fraction value. The format for denormalized numbers is shown in **Figure 3-16**.

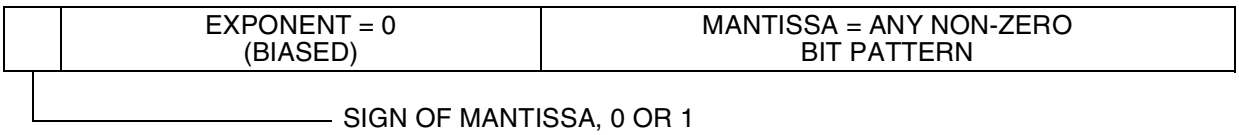

Figure 3-16 Format for Denormalized Numbers

Denormalized numbers are non-zero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is zero. Denormalized numbers are interpreted as follows:

$$DENORM = (-1)^{s} \times 2^{Emin} \times (0.fraction)$$

Emin is the minimum representable exponent value (that is, –126 for single-precision, –1022 for double-precision).

3.3.6 Infinities (±×)

Positive and negative infinities have the maximum biased exponent value:

- 255 in the single-precision format
- 2047 in the double-precision format

The format for infinities is shown in **Figure 3-17**.

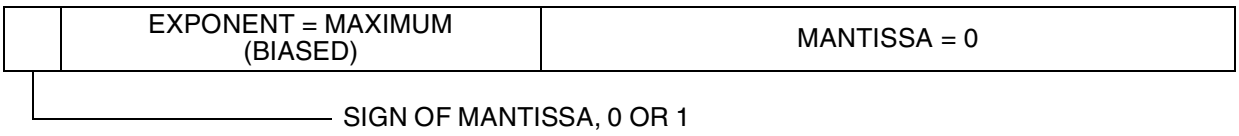

Figure 3-17 Format for Positive and Negative Infinities

The fraction value is zero. Infinities are used to approximate values greater in magnitude than the maximum normalized value. Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined between numbers and infinities. Infinities and the reals can be related as follows:

-× < every finite number < +×

Arithmetic using infinite numbers is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in **6.11.10.6 Invalid Operation Exception Conditions**.

3.3.7 Not a Numbers (NaNs)

NaNs have the maximum biased exponent value and a non-zero fraction value. The format for NaNs is shown in **Figure 3-18**. The sign bit of NaNs is ignored (that is, NaNs are neither positive nor negative). If the high-order bit of the fraction field is a zero, the NaN is a signaling NaN (SNaN); otherwise it is a quiet NaN (QNaN).

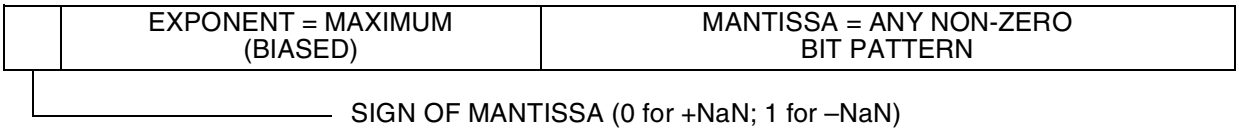

Figure 3-18 Format for NANs

Signaling NaNs signal exceptions when they are specified as arithmetic operands.

Quiet NaNs represent the results of certain invalid operations, such as invalid arith-

metic operations on infinities or on NaNs, when the invalid operation exception is disabled (FPSCR[VE] = 0). Quiet NaNs propagate through all operations, except ordered comparison, floating round to single precision, and conversion to integer operations. Quiet NaNs do not signal exceptions, except during ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of operations and used to convey diagnostic information to help identify results from invalid operations.

When a QNaN results from an operation because an operand is a NaN or because a QNaN is generated due to a disabled invalid operation exception, the following rule is applied to determine the QNaN with the high-order fraction bit set to one that is to be stored as the result:

```
If (frA) is a NaN
Then frD ← (frA)
    Else if (frB) is a NaN
        Then frD ← (frB)
        Else if (frC) is a NaN
            Then frD ← (frC)
            Else if generated QNaN
                Then frD ← generated QNaN
```

If the operand specified by **fr**A is a NaN, that NaN is stored as the result. Otherwise, if the operand specified by **fr**B is a NaN (if the instruction specifies an **fr**B operand), that NaN is stored as the result. Otherwise, if the operand specified by **fr**C is a NaN (if the instruction specifies an **fr**C operand), that NaN is stored as the result. Otherwise, if a QNaN is generated by a disabled invalid operation exception, that QNaN is stored as the result. If a QNaN is to be generated as a result, the QNaN generated has a sign bit of zero, an exponent field of all ones, and a high-order fraction bit of one with all other fraction bits zero. An instruction that generates a QNaN as the result of a disabled invalid operation generates this QNaN. This is shown in **Figure 3-19**.

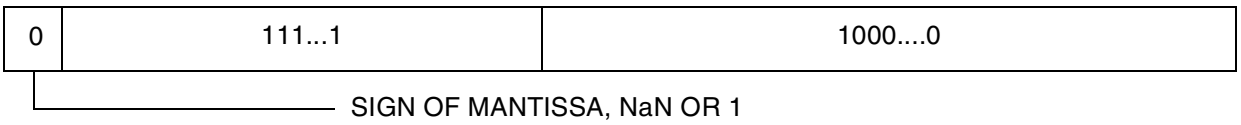

Figure 3-19 Representation of QNaN

3.3.8 Sign of Result

The following rules govern the sign of the result of an arithmetic operation, when the operation does not yield an exception. These rules apply even when the operands or results are ±0 or ±∞.

The sign of the result of an addition operation is the sign of the source operand having the larger absolute value. The sign of the result of the subtraction operation, x – y, is the same as the sign of the result of the addition operation, x+(–y).

When the sum of two operands with opposite sign, or the difference of two operands with the same sign, is exactly zero, the sign of the result is positive in all rounding modes except round toward negative infinity (–×), in which case the sign is negative.

- The sign of the result of a multiplication or division operation is the exclusive OR of the signs of the source operands.
- The sign of the result of a round to single-precision or convert to/from integer operation is the sign of the source operand.

For multiply-add instructions, these rules are applied first to the multiplication operation and then to the addition or subtraction operation (one of the source operands to the addition or subtraction operation is the result of the multiplication operation).

3.3.9 Normalization and Denormalization

When an arithmetic operation produces an intermediate result, consisting of a sign bit, an exponent, and a non-zero significand with a zero leading bit, the result is not a normalized number and must be normalized before it is stored.

A number is normalized by shifting its significand left while decrementing its exponent by one for each bit shifted, until the leading significand bit becomes one. The guard bit and the round bit participate in the shift with zeros shifted into the round bit; see **3.4.1 Execution Model for IEEE Operations**.

During normalization, the exponent is regarded as if its range were unlimited. If the resulting exponent value is less than the minimum value that can be represented in the format specified for the result, the intermediate result is said to be “tiny” and the stored result is determined by the rules described in **6.11.10.9 Underflow Exception Condition**. The sign of the number does not change.

When an arithmetic operation produces a non-zero intermediate result whose exponent is less than the minimum value that can be represented in the format specified, the stored result may need to be denormalized. The result is determined by the rules described in **6.11.10.9 Underflow Exception Condition**.

A number is denormalized by shifting its significand to the right while incrementing its exponent by one for each bit shifted until the exponent equals the format's minimum value. If any significant bits are lost in this shifting process, a loss of accuracy has occurred, and an underflow exception is signaled. The sign of the number does not change.

When denormalized numbers are operands of multiply and divide operations, operands are prenormalized internally before the operations are performed.

3.3.10 Data Handling and Precision

There are specific instructions for moving floating-point data between the FPRs and memory. Data in double-precision format is not altered during the move. Sin-

gle-precision data is converted to double-precision format when loaded from memory into an FPR. A format conversion from double- to single-precision is performed when data from an FPR is stored. Floating-point exceptions cannot occur during these operations.

All arithmetic operations use floating-point double-precision format.

Floating-point single-precision formats are used by the following four types of instructions:

- Load Floating-Point Single-Precision (**lfs**) — This instruction accesses a single-precision operand in single-precision format in memory, converts it to double-precision, and loads it into an FPR. Exceptions are not detected during the load operation.
- Round to floating-point single-precision — If the operand is not already in single-precision range, the floating round to single-precision instruction rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any exceptions according to respective enable bits in the FPSCR. The instruction places that operand into an FPR as a double-precision operand. For results produced by single-precision arithmetic instructions and by single-precision loads, this operation does not alter the value.
- Single-precision arithmetic instructions — These instructions take operands from the FPRs in double-precision format, perform the operation as if it produced an intermediate result correct to infinite precision and with unbounded range, and then force this intermediate result to fit in single-precision format. Status bits in the FPSCR and in the condition register are set to reflect the single-precision result. The result is then converted to double-precision format and placed into an FPR. The result falls within the range supported by the single format.

  For single-precision operations, source operands must be representable in single-precision format. If they are not, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the condition register, are undefined.
- Store Floating-Point Single-Precision (**stfs**) — This form of instruction converts a double-precision operand to single-precision format and stores that operand into memory. If the operand requires denormalization in order to fit in single-precision format, it is automatically denormalized prior to being stored. No exceptions are detected on the store operation (the value being stored is effectively assumed to be the result of an instruction of one of the preceding three types).

When the result of a load floating-point single-precision (**lfs**), floating-point round to single-precision (**frsp***x*), or single-precision arithmetic instruction is stored in an FPR, the low-order 29 fraction bits are zero. This is shown in **Figure 3-20**.

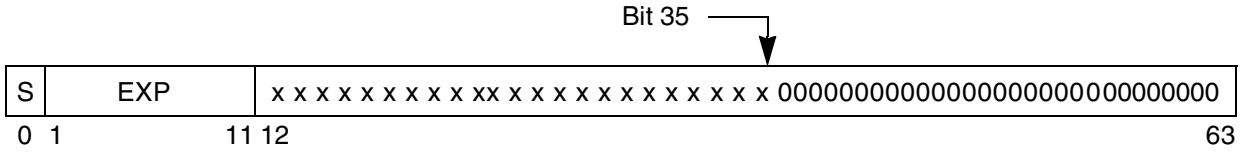

Figure 3-20 Single-Precision Representation in an FPR

The floating-point round to single-precision (**frps***x*) instruction allows conversion from double to single precision with appropriate exception checking and rounding. This instruction should be used to convert double-precision floating-point values (produced by double-precision load and arithmetic instructions) to single-precision values before storing them into single-format memory elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions can be stored directly, or used directly as operands for single-precision arithmetic instructions, without preceding the store, or the arithmetic instruction, by **frsp***x*.

A single-precision value can be used in double-precision arithmetic operations. The reverse is true only if the double-precision value can be represented in single-precision format. Some implementations may execute single-precision arithmetic instructions faster than double-precision arithmetic instructions. Therefore, if double-precision accuracy is not required, using single-precision data and instructions can speed operations.

3.3.11 Rounding

All arithmetic instructions defined by the PowerPC architecture produce an intermediate result considered infinitely precise. This result must then be written with a precision of finite length into an FPR. After normalization or denormalization, if the infinitely precise intermediate result cannot be represented in the precision required by the instruction, it is rounded before being placed into the target FPR.

The instructions that potentially round their result are the arithmetic, multiply-add, and rounding and conversion instructions. As shown in **Figure 3-21**, whether rounding occurs depends on the source values.

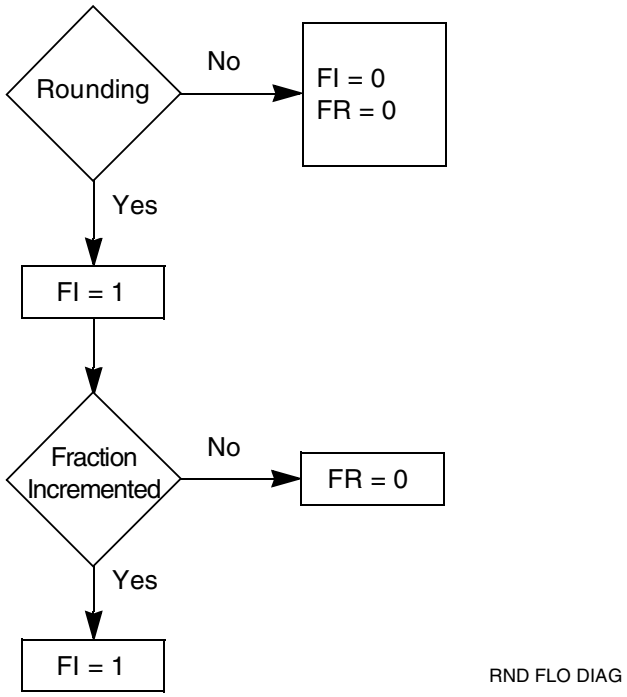

Figure 3-21 Rounding Flow Diagram

Each of these instructions sets FPSCR bits FR and FI, according to whether rounding occurs (FI) and whether the fraction was incremented (FR). If rounding occurs, FI is set to one and FR may be either zero or one. If rounding does not occur, both FR and FI are cleared. Other floating-point instructions do not alter FR and FI. Four modes of rounding are provided that are user-selectable through the floating-point rounding control field in the FPSCR. These are encoded as follows in **Table 3-7**.

Table 3-7 FPSCR Bit Settings — RN Field

RN	Rounding Mode
00	Round to nearest
01	Round toward zero
10	Round toward +infinity
11	Round toward –infinity

Let Z be the infinitely precise intermediate arithmetic result or the operand of a conversion operation. If Z can be represented exactly in the target format, no rounding occurs and the result in all rounding modes is equivalent to truncation of Z. If Z cannot be represented exactly in the target format, let Z1 and Z2 be the next larger and next smaller numbers representable in the target format that bound Z; then Z1 or Z2 can be used to approximate the result in the target format.

Figure 3-22 shows a graphical representation of Z, Z1, and Z2.

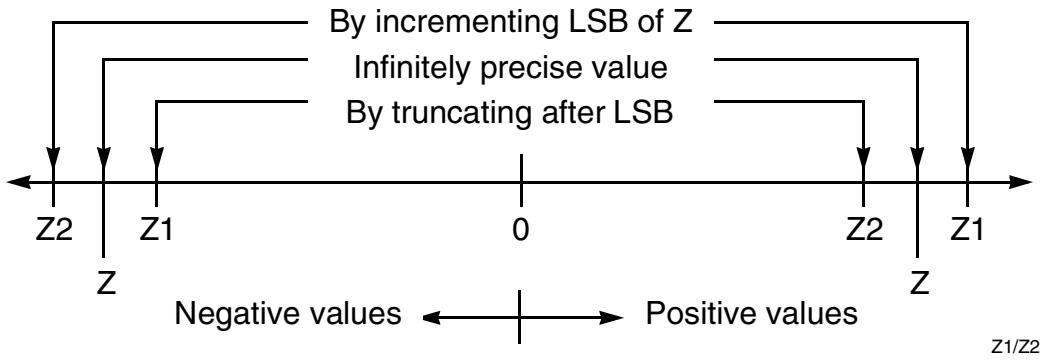

Figure 3-22 Relation of Z1 and Z2

Rounding follows the four following rules:

- Round to nearest — Choose the best approximation (Z1 or Z2). In case of a tie, choose the one which is even (i.e., with least significant bit equal to zero). Refer to **3.4.1 Execution Model for IEEE Operations** for details on how the processor selects the best approximation.
- Round toward zero — Choose the smaller in magnitude (Z1 or Z2).
- Round toward +infinity — Choose Z1.
- Round toward –infinity — Choose Z2.

If Z is to be rounded up and Z1 does not exist (that is, if there is no number larger than Z that is representable in the target format), then an overflow exception occurs if Z is positive and an underflow exception occurs if Z is negative. Similarly, if Z is to be rounded down and Z2 does not exist, then an overflow exception occurs if Z is negative and an underflow exception occurs if Z is positive. The results in these cases are defined in **6.11.10 Floating-Point Assist Exception (0x00E00)**.

3.4 Floating-Point Execution Models

The following paragraphs describe the floating-point execution models for IEEE operations, as well as that for a special multiply-add type of instruction. In addition, the execution model for non-IEEE compliant operation, used to accelerate time-critical operations, is described.

The IEEE-754 standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision arithmetic instructions to have either (or both) single-precision or double-precision operands, but states that single-precision arithmetic instructions should not accept double-precision operands.

The PowerPC architecture follows these guidelines:

- Double-precision arithmetic instructions can have operands of either or both precisions.

- Single-precision arithmetic instructions require all operands to be single-precision.
- Double-precision arithmetic instructions produce double-precision values.
- Single-precision arithmetic instructions produce single-precision values.

For arithmetic instructions, conversions from double- to single-precision must be done explicitly by software, while conversions from single- to double-precision are done implicitly.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic uses two additional bit positions to avoid potential transient overflow conditions. An extra bit is required when denormalized double-precision numbers are prenormalized. A second bit is required to permit computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is one:

- Underflow during multiplication using a denormalized factor.
- Overflow during division using a denormalized divisor.

3.4.1 Execution Model for IEEE Operations

The following description uses 64-bit arithmetic as an example. Thirty-two-bit arithmetic is similar except that the fraction field is a 23-bit field and the single-precision guard, round, and sticky bits (described in this section) are logically adjacent to the 23-bit FRACTION (or mantissa) field.

The bits and fields for the IEEE 64-bit execution model are defined as follows:

- The S bit is the sign bit.
- The C bit is the carry bit that captures the carry out of the significand.
- The L bit is the leading unit bit of the significand which receives the implicit bit from the operands.
- The FRACTION is a 52-bit field that accepts the fraction (mantissa) of the operands.
- The guard (G), round (R), and sticky (X) bits are extensions to the low-order bits of the accumulator. The G and R bits are required for post normalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits that may appear to the low-order side of the R bit, either due to shifting the accumulator right or other generation of low-order result bits. The G and R bits participate in the left shifts with zeros being shifted into the R bit. **Table 3-8** shows the relationship among the G, R, and X bits, the intermediate result (IR), the next lower in magnitude representable number (NL), and the next higher in magnitude representable number (NH).

Table 3-8 Interpretation of G, R, and X Bits

G	R	X	Interpretation
0	0	0	IR is exact
0	0	1	IR closer to NL
0	1	0	
0	1	1	
1	0	0	IR midway between NL and NH
1	0	1	IR closer to NH
1	1	0	
1	1	1	

The significand of the intermediate result is made up of the L bit, the FRACTION, and the G, R, and X bits.

The infinitely precise intermediate result of an operation is the result normalized in bits L, FRACTION, G, R, and X of the floating-point accumulator.

Before results are stored into an FPR, the significand is rounded if necessary, using the rounding mode specified by FPSCR[RN]. If rounding causes a carry into C, the significand is shifted right one position and the exponent is incremented by one. This may yield an inexact result and possibly exponent overflow. Fraction bits to the left of the bit position used for rounding are stored into the FPR, and low-order bit positions, if any, are set to zero.

Four rounding modes are provided which are user-selectable through FPSCR[RN] as described in **3.3.11 Rounding**. For rounding, the conceptual guard, round, and sticky bits are defined in terms of accumulator bits.

Table 3-9 shows the positions of the guard, round, and sticky bits for double-precision and single-precision floating-point numbers.

Table 3-9 Location of the Guard, Round and Sticky Bits

Format	Guard	Round	Sticky
Double	G bit	R bit	X bit
Single	24	25	26–52 G,R,X

Rounding can be treated as though the significand were shifted right, if required, until the least significant bit to be retained is in the low-order bit position of the FRACTION. If any of the guard, round, or sticky bits are non-zero, the result is inexact.

Z1 and Z2, defined in **3.3.11 Rounding**, can be used to approximate the result in the target format when one of the following rules is used:

- Round to nearest
  - — Guard bit = 0: The result is truncated. (Result exact (GRX = 000) or closest to next lower value in magnitude (GRX = 001, 010, or 011)
  - — Guard bit = 1: Depends on round and sticky bits:
    - Case a: If the round or sticky bit is one (inclusive), the result is incremented. (result closest to next higher value in magnitude (GRX = 101, 110, or 111))
    - Case b: If the round and sticky bits are zero (i.e., the result is midway between the closest representable values), the result is rounded to an even value. That is, if the low-order bit of the result is one, the result is incremented. If the low-order bit of the result is zero, the result is truncated.
- If during the round to nearest process, truncation of the unrounded number produces the maximum magnitude for the specified precision, the following action is taken:
  - — Guard bit = 1: Store infinity with the sign of the unrounded result.
  - — Guard bit = 0: Store the truncated (maximum magnitude) value.
- Round toward zero — Choose the smaller in magnitude of Z1 or Z2. If the guard, round, or sticky bit is non-zero, the result is inexact.
- Round toward +infinity
  Choose Z1.
- Round toward –infinity
  Choose Z2.

Where the result is to have fewer than 53 bits of precision because the instruction is a floating round to single-precision or single-precision arithmetic instruction, the intermediate result either is normalized or is placed in correct denormalized form before the result is potentially rounded.

3.4.2 Execution Model for Multiply-Add Type Instructions

The PowerPC architecture makes use of a special form of instruction that performs up to three operations in one instruction (a multiply, an add, and a negate). With this added capability is the special feature of being able to produce a more exact intermediate result as an input to the rounder. The 32-bit arithmetic is similar except that the fraction field is smaller.

NOTE

The rounding occurs only after add; therefore, the computation of the sum and product together are infinitely precise before the final result is rounded to a representable format.

The first part of the operation is a multiply. The multiply has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), the significand is shifted right one position, placing the L bit into the most significant bit of the FRACTION and placing the C bit into the L bit. All 106 bits (L bit plus the frac-

tion) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount added to that exponent to make it equal to the other input's exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X' bit. The add operation also produces a result conforming to the above model with the X' bit taking part in the add operation.

The result of the add is then normalized, with all bits of the add result, except the X' bit, participating in the shift. The normalized result provides an intermediate result as input to the rounder that conforms to the model described in **3.4.1 Execution Model for IEEE Operations**, where:

- The guard bit is bit 53 of the intermediate result.
- The round bit is bit 54 of the intermediate result.
- The sticky bit is the OR of all remaining bits to the right of bit 55, inclusive.

If the instruction is floating negative multiply-add or floating negative multiply-subtract, the final result is negated.

Status bits are set to reflect the result of the entire operation: for example, no status is recorded for the result of the multiplication part of the operation.

3.4.3 Non-IEEE Operation

The RCPU depends on a software envelope to fully implement the IEEE-754 floating-point specification. Even when all exceptions are disabled (i.e., when exception enable bits in the FPSCR are cleared), tiny results and denormalized operands cause FPU exceptions that invoke a software routine to deliver (with hardware assistance) the correct IEEE result.

To accelerate time-critical operations and make them more deterministic, the RCPU provides a non-IEEE mode of operation. In this mode, whenever a tiny result is detected and floating-point underflow exception is disabled (FPSCR[UE] = 0), the hardware delivers a correctly signed zero instead of invoking the floating-point assist exception handler.

Non-IEEE mode is entered by setting the NI (non-IEEE enable) bit in the FPSCR.

Denormalized numbers are never generated in non-IEEE mode. Therefore, when denormalized operands are detected, they are treated exactly as they are in IEEE mode. Refer to **6.11.10 Floating-Point Assist Exception (0x00E00)** for more information.

The hardware never asserts the FPSCRXX (inexact) bit on an underflow condition; it is done as a part of the floating-point assist interrupt handler. Therefore, in non-IEEE mode, FPSCRXX cannot be depended upon to be a complete accumulation of all inexact conditions.

3.4.4 Working Without the Software Envelope

Even when the processor is operating in non-IEEE mode, the software envelope may be invoked when denormalized numbers are used as the input to the calculation or when an enabled IEEE exception is detected. To ensure that the software envelope is never invoked, the user needs to do the following:

- Set the NI bit in the FPSCR to enable non-IEEE mode.
- Disable all floating-point exceptions.
- Avoid using denormalized numbers as inputs to floating-point calculations.