Itanium™ Processor Floating-point Software Assistance and Floating-point Exception Handling

January 2000

Order Number: 245415-001

intel®

Contents

Figures

Tables

Introduction 1

This document describes the details regarding Floating-point Software Assistance exceptions (FP SWA requests) in particular, and floating-point exceptions in general on the Itanium™ processor, the first implementation of the IA-64 architecture. The document is useful to operating system writers and compiler writers, besides being useful to anyone who wants to obtain a better understanding of floating-point exceptions in the IA-64 architecture. Chapter 1 through Chapter 3 contain the general background information, while Chapter 4 through Chapter 7 are more focused, and go into a lower level of detail. Chapter 7 gives the information necessary in integrating the FP SWA Handler (the FP SWA EFI driver) with the operating system.

Chapter 1 contains the introduction and describes the software components of an operating system supporting floating-point exception handling.

Chapter 2 describes the Software Assistance (SWA) traps and faults on the Itanium processor, and

Chapter 3 lists the conditions causing floating-point exceptions, the floating-point exception priorities (distinguishing between exceptions raised [signaled] directly by the hardware and exceptions raised by the software), and specifies the response of the various Itanium processor instructions to floating-point exceptions - for both disabled (masked) and enabled (unmasked) exceptions.

Chapter 4 discusses the IA-64 architecturally mandated Software Assistance requests, which can be raised only by the divide and square root reciprocal approximation instructions (frcpa and frsqrta). The floating-point divide and square root operations (as well as other operations based on them, such as remainder, or integer divide and remainder) are implemented in software in the IA-64 architecture. The starting point which is provided by the reciprocal approximation instructions is followed by instructions that implement Newton-Raphson based or similar algorithms for divide and square root or their derivatives. Software assistance is required when the reciprocal approximation instructions implemented in hardware are not able to provide an initial value sufficient for the software algorithms to determine the IEEE correct results for divide or square root. Alternate algorithms are used when such requests are made by the hardware. The frequency of occurrence for software assistance requests is estimated in Section 4.3 and Section 4.5.

Chapter 5 examines the architecturally mandated Pseudo-Software Assistance requests, characteristic for the parallel divide and square root reciprocal approximation instructions. These requests are raised in situations similar to those for the scalar divide and square root reciprocal approximation instructions, but instead of leading to a SWA fault or trap, the output predicate of the parallel reciprocal approximation instruction is cleared. The frequency of occurrence for the pseudo-software assistance requests is estimated in Section 5.2 and Section 5.4.

Chapter 6 gives examples of software assistance requests. Examples of Itanium processor specific SWA requests are given first, followed by IA-64 architecturally mandated requests.

Chapter 7 describes the implementation of the software component that handles software assistance requests - the IA-64 Floating-point Emulation Library, and specifies the API that allows this library supported by Intel to be shared by various operating systems. The IA-64 Floating-point Emulation Library (which has the role of a Floating-point SWA Handler) is implemented as an EFI (Extensible Firmware Interface) driver.

Chapter 8 contains the references used in the text.

1.1 Related Documents

The IA-64 floating-point architecture and operations are discussed in several other documents. Information contained in these sources is useful, or helpful when reading the present document.The main source of information is the Intel IA-64 Architecture Software Developer's Guide [1]. Information specific to the Intel architecture can also be found in [2].

The recommended source for reference information on floating-point exceptions is the IEEE Standard 754-1985 for Binary Floating-point Computations [3].

1.2 Software Components Supporting Floating-point Exception Handling

Floating-point exception handling in the IA-64 architecture has new features compared to IA-32 [2]. First, the IA-64 floating-point architecture is more complex than that of previous Intel processors. There are new instructions, some with three input operands, static precision modes (e.g. in fma.s), static rounding modes (e.g. in fcvt.fx.trunc), and new floating-point formats and computation models. Second, the necessity for software assistance (SWA) is new in the Intel Architecture, and augments the floating-point exception handling mechanism. Third, the IA-64 architecture also has the ability to handle new parallel floating-point instructions. On the other hand, the IA-64 architecture has extended floating-point capabilities, both in performance and in accuracy.

When a floating-point exception occurs, the hardware saves a minimal amount of processor state information in interruption control registers (only the registers of interest for floating-point exceptions are listed): IPSR (CR16 - Interruption Processor Status Register), ISR (CR17 - Interruption Status Register), IIP (CR19 - Interruption Instruction Bundle Pointer), IIPA (CR22 - Interruption Instruction Previous Address), and IFS (CR23 - Interruption Function State). The information in IIP, IPSR, and IFS is saved only if PSR.ic is set to 1. The information in IIPA is saved only if PSR.ic was 1 prior to the interruption. Finally, ISR is saved regardless of the value of PSR.ic.

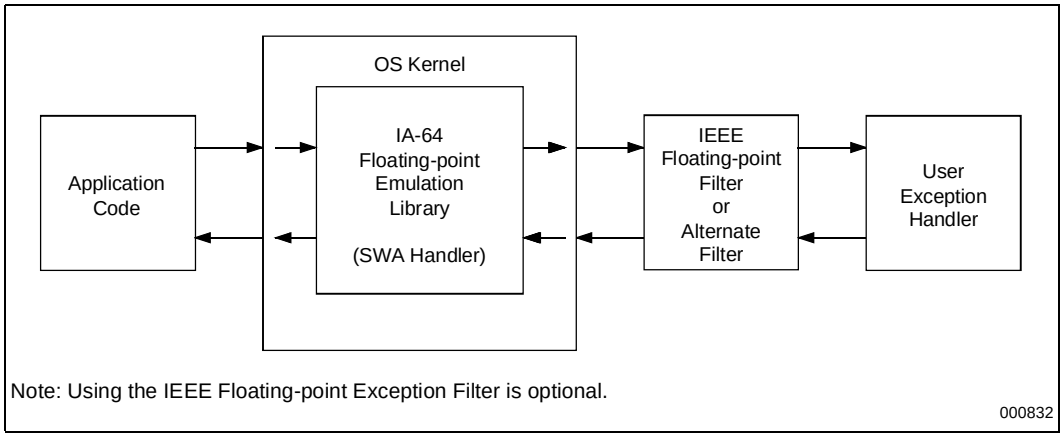
A branch to the interruption vector (Floating-point Fault Vector 0x5c00 or Floating-point Trap Vector 0x5d00) and then to a low-level OS handler allows saving more of the processor state, and propagates handling of the task higher in the operating system.

On any platform based on an IA-64 processor, two system-level components are used in floating-point exception handling: the operating system kernel floating-point trap handler, and the IA-64 Floating-point Emulation Library (FP SWA Handler). The kernel floating-point trap handler has the role to save state information not saved by the processor, and then to invoke the appropriate exception handler: the FP SWA Handler for FP SWA requests, or the FP SWA handler and then a user-level floating-point exception handler for other unmasked (enabled) floating-point exceptions.

Figure 1-1 depicts the control flow that occurs when an application running on an IA-64 processor causes a floating-point exception condition.

The IA-64 Floating-point Emulation Library (implemented as an EFI driver invoked by the OS kernel) is capable of emulating any floating-point instruction defined by the architecture. It handles the cases that require software assistance — situations that the hardware cannot handle, which fall into the three categories presented in the next subsection.

Figure 1-1. Flow of Control for IA-64 Floating-point Exceptions



Software Assistance Faults and Traps on the Itanium™ Processor 2

The three categories of Software Assistance exceptions: IA-64 architecturally mandated SWA faults, Itanium processor specific SWA faults, and Itanium™ processor specific SWA traps, are presented next.

2.1 IA-64 Architecturally Mandated SWA Faults

The architecturally mandated SWA faults occur for the scalar reciprocal approximation instructions, *frcpa* and *frsqrta*, when their input operands are such that they potentially prevent generation of the correct results by the iterative software algorithms that are employed for divide and square root. Alternate algorithms are implemented in the IA-64 Floating-point Emulation Library to provide the correct results in such situations.

In the case of the Itanium processor specific SWA exceptions (Section 2.2 and Section 2.3 below), the SWA fault or trap can be caused by both scalar or parallel instructions. The architecturally mandated SWA faults are caused only by scalar instructions (*frcpa* and *frsqrta*). The parallel counterparts of the reciprocal approximation instructions, *fprcpa* and *fprsqrta* just clear their output predicate (in situations in which the scalar instructions would raise SWA faults), expecting this to cause alternate algorithms to be executed in order to perform the parallel divide or square root operations.

2.2 Itanium™ Processor Specific SWA Faults

SWA faults are allowed in the IA-64 architecture for virtually any reason. The architecturally allowed SWA faults that occur on the Itanium processor are referred to as Itanium processor specific SWA faults, and they arise when floating-point instructions consume denormalized or unnormal operands. If the denormal exceptions are disabled (masked), the SWA fault is resolved by the IA-64 Floating-point Emulation Library (the SWA handler). If the denormal exceptions are enabled (unmasked), the SWA fault is converted to a denormal fault by the IA-64 Floating-point Emulation Library, and it is propagated through the OS kernel to the user level (a denormal fault exception handler must have been registered to handle it). As SWA faults may be raised for any reason in an IA-64 Architecture implementation in general, the IA-64 Floating-point Emulation Library was designed and implemented to be able to provide the correct result for any IA-64 floating-point arithmetic instruction, and for any values of the input operands.

2.3 Itanium™ Processor Specific SWA Traps

SWA traps are allowed in the IA-64 architecture when:

- Tiny results are generated and the underflow traps are disabled.
- Huge results are generated and the overflow traps are disabled.
- Inexact results are generated and the inexact traps are disabled.

Note that tiny numbers have non-zero values, but less in absolute value than the smallest positive normal floating-point number. Huge numbers have values larger in absolute value than the largest positive normal floating-point number. The result of a floating-point operation is evaluated for tininess or hugeness after rounding to the destination precision, but assuming an unbounded exponent (“first IEEE rounding”), and before the second rounding that takes into account the limited exponent range (“second IEEE rounding”). Note though that these two rounding steps are hypothetical, and that the hardware only performs the IEEE rounding in one step (combining the two steps outlined above). Breaking it into two steps just helps understanding the way numeric results are generated. For tiny results, rounding will require denormalization, i.e. shifting the significand to the right, while incrementing the exponent in order to bring it into the range allowed by the format, followed by rounding to the destination precision. This has to be carried out on the infinitely precise result, and the rounded result may be zero, a denormal, or the smallest normal number representable in the destination format, with the appropriate sign (which means that the first rounding step is not necessary in this case). If the result is huge, the second rounding will modify it either to the largest normal floating-point number representable in the destination format, or to infinity (with the appropriate sign).

The Itanium processor specific SWA traps occur only when tiny results are generated, the underflow traps are disabled, and the flush-to-zero mode is not enabled.

2.4 Handling Floating-point Exceptions

When a SWA request occurs, an instruction bundle is read, the excepting instruction is decoded, its input or output operands are read, a result is generated, and the processor state is modified by the software.

The IA-64 Floating-point Emulation Library provides a result only for SWA faults and traps. Exceptions are the cases when a SWA fault or trap generates a new floating-point exception, e.g. for an Itanium processor specific SWA faults, when the denormal exceptions are enabled (in which case the result is provided by a user handler for denormal exceptions). The library is invoked though for all the enabled (unmasked) floating-point exceptions (SWA or not).Three situations are possible.

The first possibility is for the emulation library to recognize a SWA fault or a SWA trap. It starts processing it, and if a result can be generated, it is passed back to the kernel floating-point trap handler, which in turn has to resume the thread that raised the exception.

The second possibility is for the emulation library to recognize an unmasked floating-point exception other than a SWA fault or trap, or to have to raise a new floating-point exception that occurs during the process of generating a result for the SWA fault or trap. This information is then returned to the kernel trap handler, which will have to propagate the exception to a user level floating-point exception handler.

The third possibility is for the emulation library to not recognize a floating-point fault or trap when called by the OS kernel (this may include the case when incorrect parameters were passed to it). In this situation, it returns to the OS kernel a value indicating failure, plus additional diagnostic information.

At the user level, the floating-point exception can be handled by a user handler directly, or by a filter function (usually an IEEE floating-point exception filter) that invokes a user handler.

The first function of an IEEE Floating-point Exception Filter is to transform the interruption information to a format that is easier to understand and handle by the user, and to invoke a user handler for the exception. The user provided result, and possibly other changes are propagated back into the processor state if execution is continued (in some programming environments, the

user has up to three options: to continue execution, to execute some cleanup code and exit, or to continue searching for another handler).

The second function of the filter is to hide the complexities of the parallel instructions from the user. If a floating-point fault occurs for example in the high half of a parallel floating-point instruction, and there is a user handler provided for that case, the parallel instruction is split into two scalar instructions. The result for the high half comes from the user handler, while the low half is re-executed. The two results are combined back into a parallel result, and execution can continue. More complicated cases are those when two faults and/or traps occur in the same instruction (the model used can be extended to more than 2-way parallel instructions). Note that usage of the IEEE Floating-point Exception Filter is not compulsory - the user may choose to handle enabled floating-point exceptions differently. A filter can be provided just as a convenient way to solve such situations. Still, at least a filter with reduced functionality is necessary in order to ensure full compliance with the IEEE-754 Standard requirements [6] regarding values to be passed to a user handler when a floating-point exception occurs (e.g. scaling of the hardware generated result when an overflow or underflow exception is raised has to be performed by the filter function).

The next chapter describes the IA-64 instructions that are capable of raising floating-point exceptions, and the conditions under which these exceptions may occur. The following chapters will focus almost exclusively on SWA exceptions, but the reason for presenting all the floating-point exceptions in the beginning is that SWA requests are floating-point exceptions themselves (even though not user visible), and because they can be combined with, or immediately followed by other floating-point exceptions.

Conditions Causing, and Responses to, Floating-point Exceptions 3

The Itanium processor arithmetic instructions, most of which can cause floating-point exceptions, are listed in Table 3-1 (the only arithmetic instruction that cannot raise floating-point exceptions is *fcvt.xf*, which converts a signed integer value to register file floating-point format).

Table 3-1. Itanium™ Processor Floating-point Arithmetic Instructions and Floating-point Exceptions which may be Raised

FP Instructions	Exceptions	
	Faults	**Traps**
fma	V, D, SWA	O, U, I, SWA
fnorm	V, D, SWA	O, U, I, SWA
fpma	V, D, SWA	O, U, I, SWA
fms	V, D, SWA	O, U, I, SWA
fpms	V, D, SWA	O, U, I, SWA
fnma	V, D, SWA	O, U, I, SWA
fpnma	V, D, SWA	O, U, I, SWA
fmax	V, D, SWA	
fpmax	V, D, SWA	
fmin	V, D, SWA	
fpmin	V, D, SWA	
famax	V, D, SWA	
fpamax	V, D, SWA	
famin	V, D, SWA	
fpamin	V, D, SWA	
fcmp	V, D, SWA	
fpcmp	V, D, SWA	
fcvt.fx	V, D, SWA	I
fpcvt.fx	V, D, SWA	I
fcvt.xf		
frcpa	V, Z, D, SWA	O,U,I [a]
fprcpa	V, Z, D, SWA	
frsqrta	V, D, SWA	I [a]
fprsqrta	V, D, SWA	

a. The traps indicated for frcpa and frsqrta cannot be generated directly by the hardware, but they can be raised by the IA-64 Floating-point Emulation Library, following a SWA fault for these instructions.

Among the instructions that are pseudo-ops, only *fnorm* is included, as it requires special attention. It is important to know that

fnorm.pc.sf f1 = f3

is equivalent to

fma.pc.sf f1 = f3, F1, F0

where F1 contains +1.0, F0 contains +0.0, and *f1* and *f3* are any other floating-point registers (*f3* can also be F1 or F0). It is not equivalent to an *fma* instruction using another combination of registers, e.g.

fma.pc.sf f1 = F1, *f3*, F0

For example, assuming that the denormal exceptions are disabled, the instruction above would raise a SWA fault for any unnormal operand with a non-zero exponent, while the former (equivalent to an *fnorm*) would not. The instruction

fma.pc.sf f1 = f3, F1, F0

will raise a SWA fault only if *f3* is unnormal and its biased exponent in floating-point register file format is 0, or if *f3* is unnormal and the denormal faults are enabled (the same holds for *fms.pc.sf f1 = f3*, F1, F0 and *fnma.pc.sf f1 = f3*, F1, F0). By contrast,

fma.pc.sf f1 = F1, *f*3, F0

will raise a SWA fault if *f3* is unnormal (regardless of its exponent, or of the denormal exceptions being disabled or not).

As noted, the similar statements hold for *fms* and *fnma*, i.e. the *fms* and *fnma* instructions follow the same conventions for software assistance as *fma*. For example,

fms.pc.sf f1 = f3, F1, F0

acts like an *fnorm.*

The other pseudo-ops that are not included in the list above are *fadd*, *fcvt.xuf*, *fmpy* (pseudo-ops of *fma*), *fnmpy* (pseudo-op of *fnma*), *fpmpy* (pseudo-op of *fpma*), *fpnmpy* (pseudo-op of *fpnma*), and *fsub* (pseudo-op of *fms*). Their floating-point exception behavior follows that of the instructions they are derived from, with one exception: if *fma*, *fms*, *fnma*, *fpma*, *fpms*, or *fpnma* raise a SWA fault and f2 is F0 (as in *fma.pc.sf f1 = f3*, *f4*, F0), then the add operation in the multiply-add has to be skipped (the result has to be the IEEE result for a multiply operations, and if the product *f3 * f4* is 0.0 in absolute value, then adding or subtracting 0.0 might change the sign of the result).

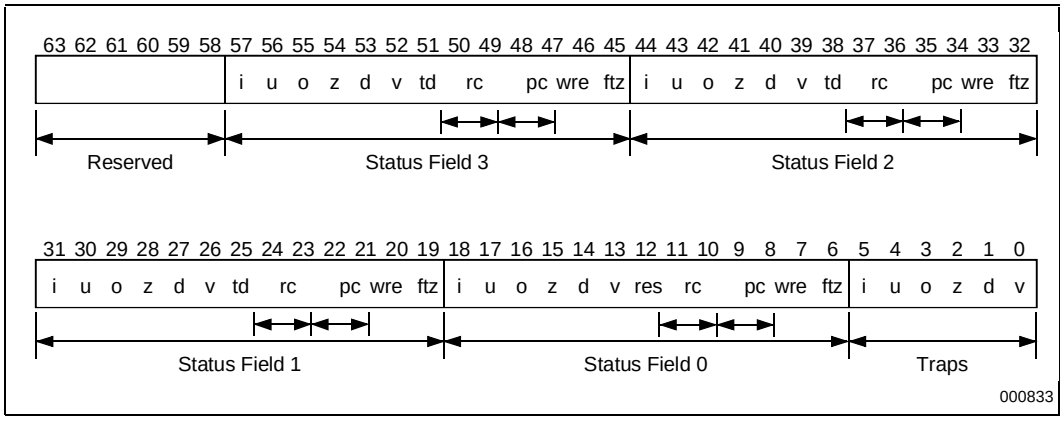
Note that only Invalid (V), Divide-by-Zero (Z), Overflow (O), Underflow (U), and Inexact (I) are IEEE exceptions. Denormal/Unnormal (D) exceptions are specific to the IA-32 architecture and to the IA-64 architecture, and SWA faults and traps are IA-64 specific. Note also that the invalid exceptions, identified by “V” here might be denoted by “I” in other documents, and the inexact exceptions identified by “I”, might be denoted by “P” (for “precision”).

In the following, references will often be made to the Floating-point Status Register, or FPSR (Application Register 40), and to the Interrupt Status Register code, or ISR code (the ISR is Control Register 17). Their definitions are included here too (but complete descriptions are given in [1]).

3.1 The Floating-point Status Register

The Floating-point Status Register (Application Register 40) contains the dynamic control and status information for floating-point operations. There is one main set of control and status information (FPSR.sf0) and three alternate sets (FPSR.sf1, FPSR.sf2, and FPSR.sf3). The FPSR layout is shown in Figure 3-1.

Figure 3-1. Floating-point Status Register (AR40)



Control bits 0 through 5 contain mask bits for floating-point exceptions (invalid, denormal, divide-by-zero, overflow, underflow, and inexact). If a mask bit is set, it disables the corresponding exceptions regardless of the status field being used.

Each of the four status fields contains seven control bits (*ftz* - flush-to-zero, *wre* - widest range exponent, *pc* - 2 bits for precision control, *rc* - 2 bits for rounding control, *td* - traps disabled [this bit is reserved in status field 0]), and six status flags (invalid, denormal, divide-by-zero, overflow, underflow, and inexact). Provided the underflow exceptions are disabled, the flush-to-zero mode (*ftz* = 1) causes tiny results to be truncated to the correctly signed zero, and the status flags for underflow and inexact exceptions to be set. The widest range exponent bit, *wre*, when set, specifies that the 17-bit exponent range will be used for floating-point calculations. The *pc* field specifies the dynamic precision for floating-point calculations (*pc* = 00 for 24-bit significands, *pc* = 10 for 53-bit significands, and *pc* = 11 for 64-bit significands). The *rc* field determines the rounding mode (*rc* = 00 for rounding to nearest, *rc* = 01 for rounding to negative infinity, *rc* = 10 for rounding to positive infinity, and *rc* = 11 for rounding to zero). When set, the *td* bit (applicable only to status fields 1, 2, and 3), disables the invalid, denormal, divide-by-zero, overflow, underflow, and inexact exceptions for floating-point operations using the corresponding status field. For status field 0, or when *td* = 0 for status fields 1, 2, or 3, control bits 0 through 5 in the FPSR determine which floating-point exceptions are masked.

3.2 The Interruption Status Register

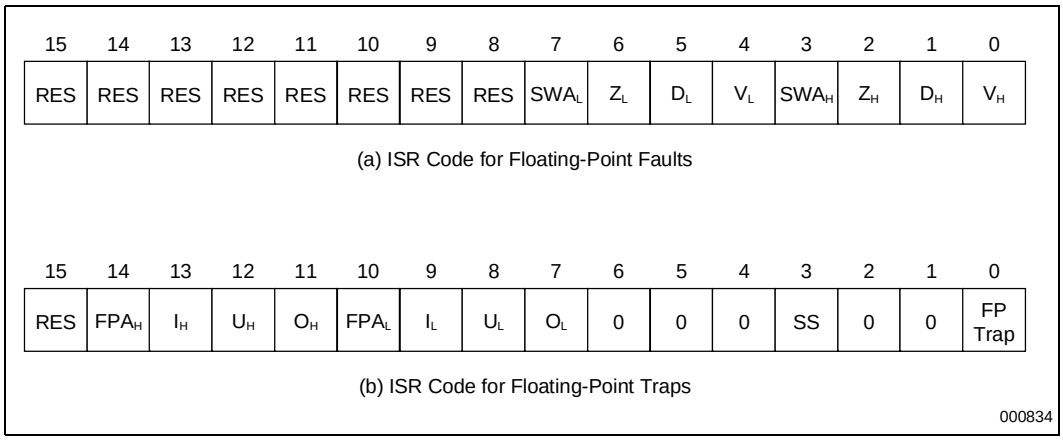
The Interruption Status Register (Control Register 17) receives information related to the nature of an interruption. Its lower 16 bits contain the ISR code, providing additional information specific to the current interruption. For unmasked floating-point exceptions, the ISR code contains the only indication for the cause of the interruption.

Figure 3-2 (a) shows the ISR code for floating-point faults. Only the lower eight bits are defined. Bits 0 through 3 (V - invalid operation, D - denormal operand, Z - divide-by-zero, SWA - software assistance) refer to floating-point faults raised by scalar instructions, or by the high order

components of parallel (SIMD - Single Instruction, Multiple Data stream) instructions. Bits 4 through 7 (V, D, Z, SWA) refer to floating-point faults raised by the low order components of parallel instructions.

Figure 3-2 (b) shows the ISR code for floating-point traps. Bit 0 (FP TRAP) is always 1, indicating a floating-point trap. Bit 3 (SS) indicates a single-step trap. Bits 7 through 10 (O - overflow, U - underflow, I - inexact, FPA) refer to floating-point traps raised by the low order components of parallel instructions. Bits 11 through 14 (O, U, I, FPA) refer to floating-point traps raised by scalar instructions, or by the high order components of parallel instructions. The FPA bit indicates that the significand is larger in absolute value than the significand of the infinitely precise result. Note that there is no bit in the ISR code to indicate the occurrence of a SWA trap. There is no ambiguity though - the SWA handler can detect this situation by examining the ISR code bits for traps and the FPSR exception bits that mask/unmask floating-point exceptions.

Figure 3-2. Interruption Status Register Code (ISR.code) from ISR (CR17)



3.3 Floating-point Exception Priority

The floating-point exception priority on the Itanium processor is the following:

1. NaTVal operand (not an exception, but handling this case has priority over floating-point exceptions).
2. Software Assistance (SWA) Floating-point Exception fault, Itanium processor specific, when one or more operands are unnormal (with the restrictions specified in Table 3-2 of Section 3.4).
3. Invalid Operation (V) Floating-point Exception fault due to one or more operands being in an unsupported format.
4. Invalid Operation (V) Floating-point Exception fault due to one or more operands that are signaling NaN (SNaN) (or QNaN for certain types of floating-point compare).
5. QNaN operand (not an exception, but handling this case has priority over lower-priority floating-point exceptions).
6. Invalid Operation (V) Floating-point Exception fault due to any reason other than those mentioned above (e.g. when executing frcpa for 0/0, or frsqrta for -Inf).
7. Zero Divide (Z) Floating-point Exception fault.
8. Denormal/Unnormal Operand (D) Floating-point Exception fault.

9. Software Assistance (SWA) Floating-point Exception fault, architecturally mandated (only for frcpa and frsqrta, when the exponents of the input operands satisfy certain conditions - see Chapter 4).
10. Software Assistance (SWA) Floating-point Exception trap: when an fma, fpma, fms, fpms, fnma, or fpnma operation has a tiny result, the underflow exceptions are disabled, and the flush-to-zero mode is not enabled.
11. Numeric Overflow (O) and Underflow (U) Floating-point Exception traps (inexactness can also be indicated in the ISR.code field).
12. Inexact (I) Floating-point Exception trap.

This list reflects the fact that the IA-64 architecture asks for SWA faults that are not architecturally mandated to be checked for prior to any other exceptions. For the Itanium processor though, these SWA faults will not occur if any invalid or divide-by-zero faults are raised (besides NaTVal or QNaN operands that do not cause invalid faults). Thus the "true" floating-point exception priority on the Itanium processor is illustrated in Figure 3-3 for faults, and Figure 3-4 and Figure 3-5 for traps. In both figures, the diamond shaped blocks are for decisions, the rectangular shaped blocks represent intermediate states, and the rectangular shaped blocks with rounded corners represent terminal states. The components represented with dotted lines correspond to software actions, while the rest are carried out directly by the processor hardware.

Figure 3-3, Figure 3-4 and Figure 3-5 apply to scalar floating-point instructions. For parallel instructions, two faults or two traps may occur simultaneously. In such cases, both faults or traps are reflected through bits set in the ISR register or in the appropriate status field of the FPSR register. Priorities are established on each half independently, but a fault in one half will have to be handled before a trap in the other half.

Note that the FPSR status flags are not updated when an umasked fault occur, and no result is provided to the exception handler. The status flags are updated on an unmasked trap (the exception handler will see the modified status bits in the appropriate status field of the FPSR) and a result is also provided to the exception handler. This is different from the IA-32 case, where status flags in the status word are updated for both unmasked floating-point faults and unmasked floating-point traps. In both cases of unmasked faults or traps, the cause of the exception is indicated by bits set in the ISR code.

If two floating-point faults or traps of the same kind (unmasked or not) occur in the two halves of a parallel instruction, the corresponding status flag in the FPSR can be viewed as a logical "OR" of the two hypothetical status flags for the individual halves (thus if a status flag is set when the exceptions are masked, one cannot tell whether the cause was in the low or in the high half of the instruction). For unmasked exceptions raised by parallel instructions, the FPSR has to be viewed in the same way, but the ISR code (as seen in Section 3.2 above) has separate sets of bits identifying exceptions in the low and the high halves.

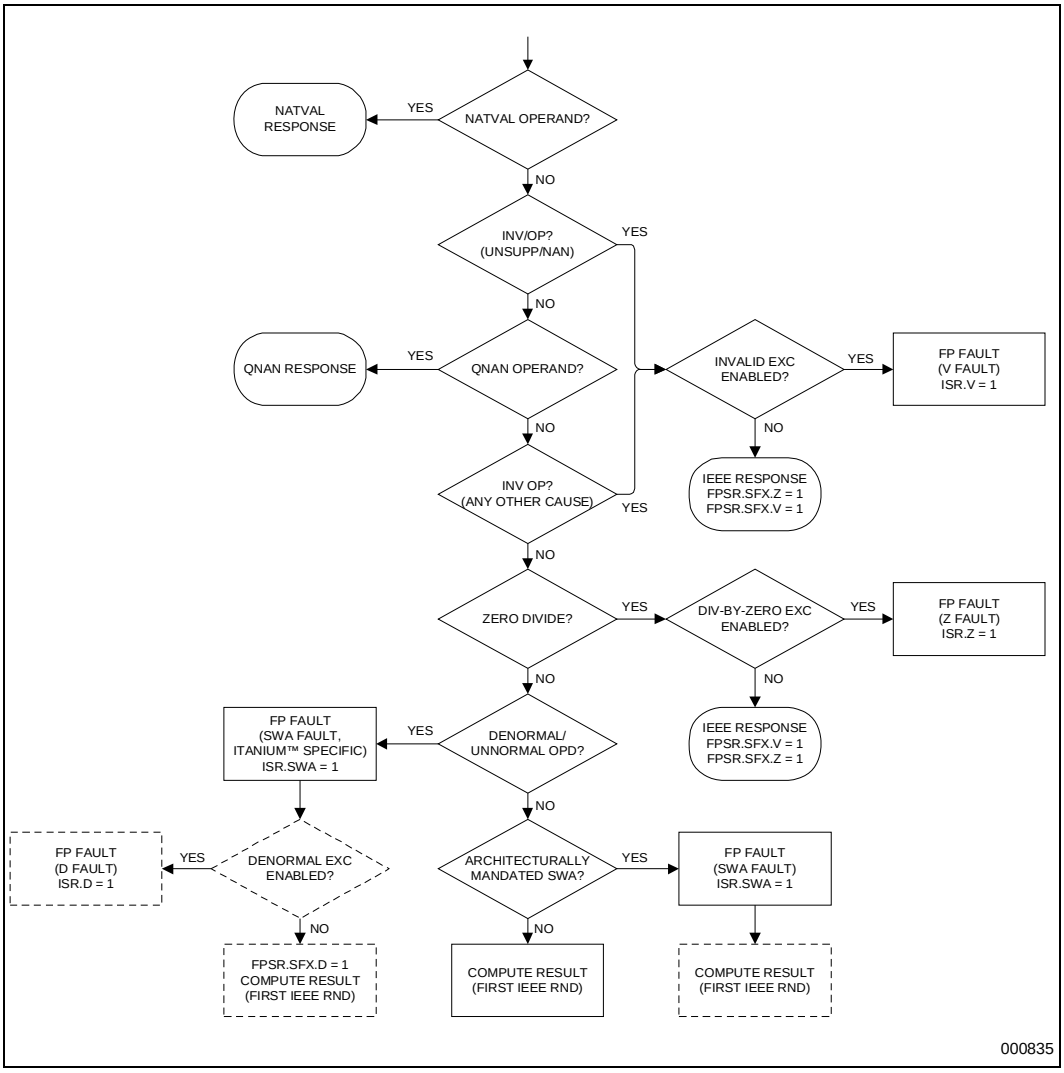
The flowchart in Figure 3-3 starts after the input operands have been read. It includes also the cases of NaTVal and NaN operands (not exceptions, but fitting as priority among floating-point exceptions). In this figure, "IEEE Response" does not apply literally to all the instructions, as not all have their behavior specified in the IEEE standard. When this is the case, behavior following the spirit of the standard is defined for each Itanium processor instruction. The flowchart in Figure 3-3 ends with the computation of the result corresponding to the "first IEEE rounding" as specified in the IEEE Standard 754-1985 [6] (again, this does not apply literally to all the instructions), i.e. the result rounded to the destination precision, but with unbounded exponent. Special cases worth mentioning are those when the result of the *frcpa* and *frsqrta* instructions is computed in software following an Itanium processor specific SWA fault (in the leftmost of the three final states marked "COMPUTE RESULT" in Figure 3-3): if any input argument is unnormal and the mathematical conditions for the architecturally mandated SWA faults (see Table 3-2 of Section 3.3) are not met, then only an 11-bit reciprocal approximation value is returned, that will

allow computation by a compiler-inlined instruction sequence of the result for the divide or square root operation.

There is one exception to the flowchart in Figure 3-3: the frsqrta and fprsqrta instructions applied to a negative non-zero unnormal argument (this includes negative non-zero denormal values) does not signal an invalid fault directly. Instead, an Itanium processor specific SWA fault is raised. If the invalid exceptions are disabled (masked), the QNaN Indefinite value is returned. If the invalid exceptions are enabled (unmasked), an invalid fault is raised.

Note that in the process of computing the result of an instruction (when no unmasked floating-point fault exception occurs, or following a SWA fault), floating-point traps may be raised. This computation may be initiated in hardware, or in software when it follows a SWA fault.

Figure 3-3. Exception Priority for Itanium™ Processor Floating-point Faults



The flowchart in Figure 3-4 applies to the case when the result of the first IEEE rounding (as shown in Figure 3-3) was computed in hardware. The result is tested for zero or infinity, and then for tininess or hugeness. The result is tiny if its exponent satisfies $e < e_{min}$, and it is huge if $e > e_{max}$. In Figure 3-4 (and Figure 3-5 too), assigning to an ISR bit (e.g. in "ISR.I ASSIGNED") means setting it to 0 or 1, as appropriate for the operation it refers to. Updating an FPSR bit (e.g. in

"FPSR.SFX.I UPDATED") means performing a logical OR between the old value of the status flag bit in the user status field of the FPSR, and the value of the status flag (e.g the inexact status flag bit) for the current operation.

If the computation of the result of the first IEEE rounding operation shown in Figure 3-3 was performed in software (following a floating-point SWA fault), then the exception priority for floating-point traps is that depicted in Figure 3-5.

In both Figure 3-4 and Figure 3-5, the "second IEEE rounding" has as a result a value rounded to the destination precision and with bounded exponent range. Special cases of the second IEEE rounding can be those when the result is the correctly signed zero, denormal, smallest normal floating-point value, largest normal floating-point value, or infinity. The second IEEE rounding (for the instructions where this is applicable) starts with the result of the first IEEE rounding (plus a few additional bits of information - rounding mode, and round and sticky bits from the first IEEE rounding), and is not based on the values of the input operands.

Figure 3-4. Exception Priority for Itanium™ Processor Floating-point Traps Generated by a Hardware Initiated Computation of the Result

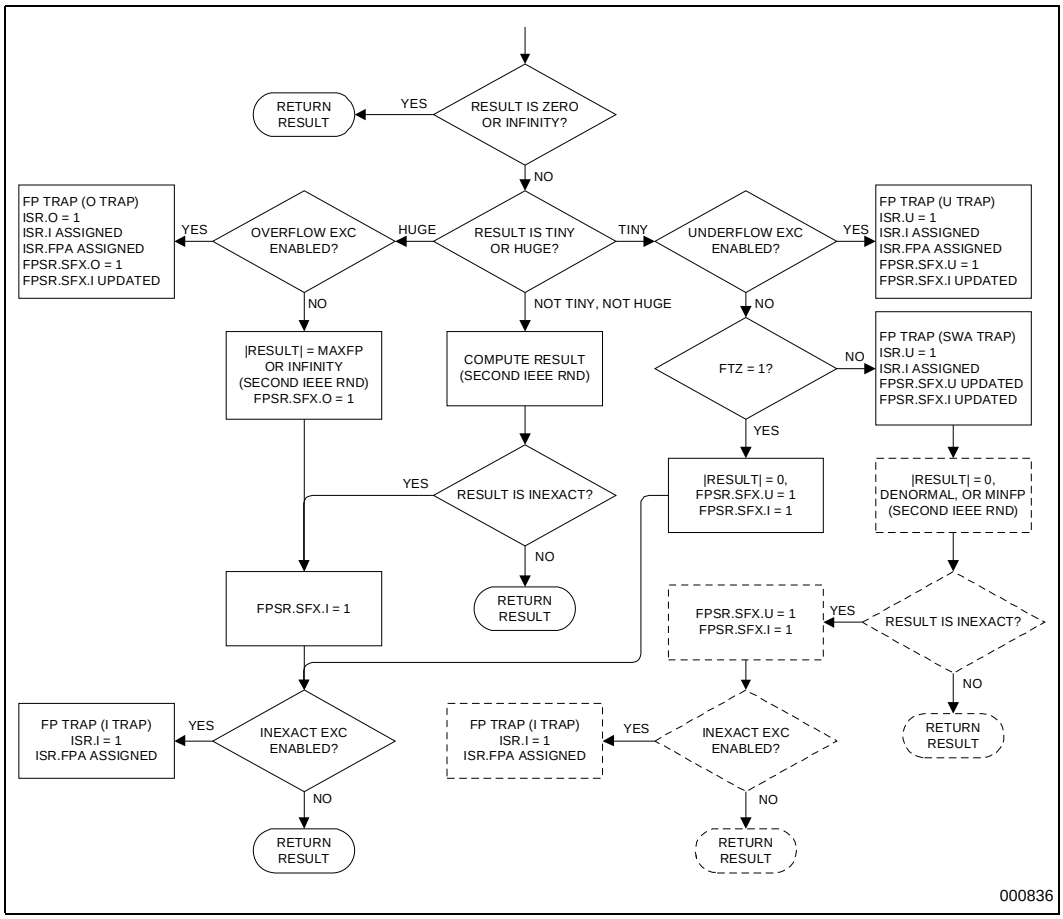
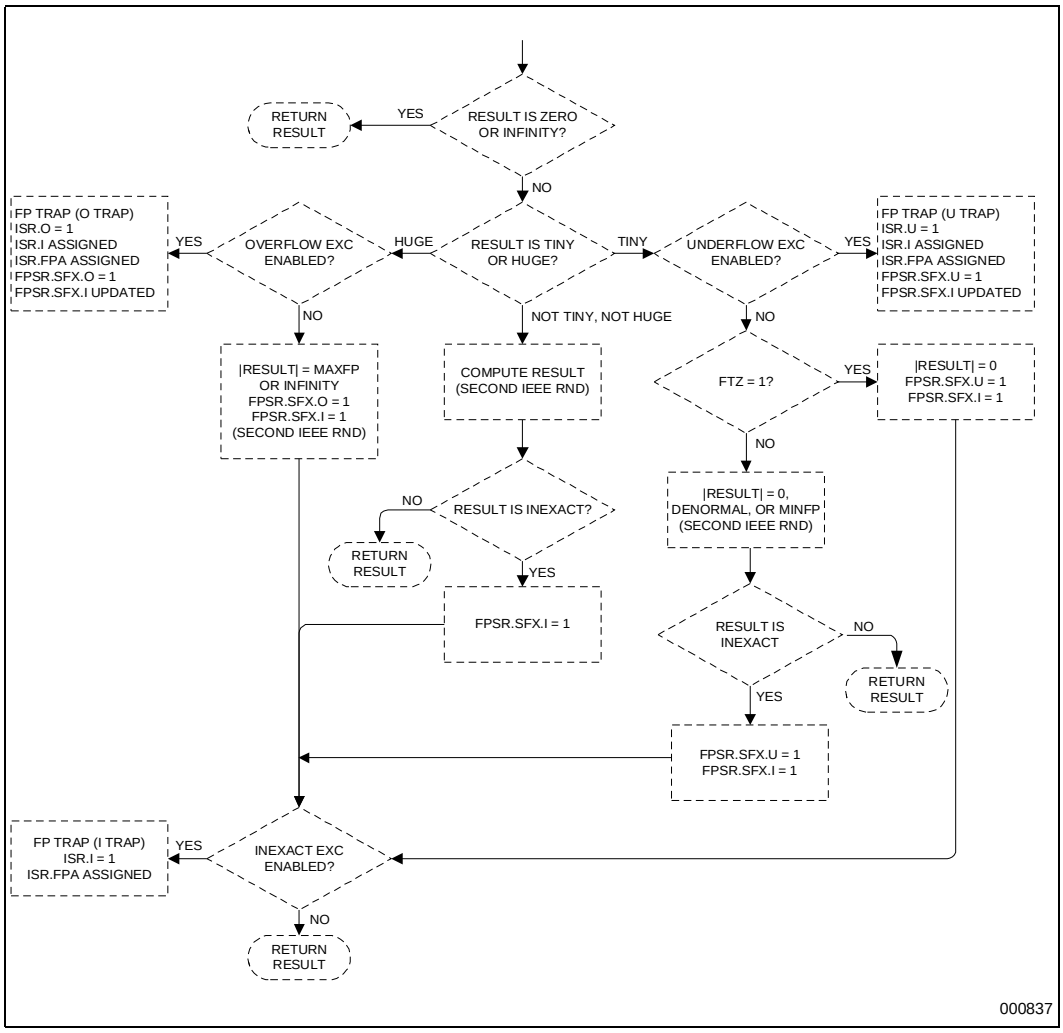


Figure 3-5. Exception Priority for Itanium™ Processor Floating-point Traps Generated by a Software Initiated Computation of the Result



For SWA traps, unmasked underflow traps, and unmasked overflow traps, the exception handler (the IA-64 Floating-point Emulation Library for SWA traps, or a user handler otherwise) receives the result after the first IEEE rounding, with exponent truncated to 17 bits. For unmasked inexact traps, the exception handler receives the result after the second IEEE rounding (which can include the special cases of a correctly signed zero, denormal, smallest normal floating-point number, largest floating-point number, or infinity).

Note that if an enabled floating-point fault is taken, no status flag is updated in the FPSR, and no result is provided to the exception handler. If an enabled floating-point trap is taken, the appropriate status flags are updated in the FPSR, and a result is provided to the exception handler. (The way the status flags are being set is different from the IA-32 behavior, where status flags are updated in the status word on any enabled floating-point exception.) In both cases (enabled faults or traps), the cause of the exception is indicated by bits set in the ISR code.

3.4 Conditions Causing Floating-point Exceptions on the Itanium™ Processor

The conditions under which floating-point exceptions occur for each Itanium processor arithmetic instruction are listed in Table 3-2 below. This table applies to both disabled and enabled exceptions. Note that SWA requests (faults or traps) are always “enabled” - there is no way to disable them. It is assumed that NaTVal values are filtered in advance. It is also assumed throughout this document that the qualifying predicate of any instruction being discussed is set to 1. For both disabled and enabled exceptions, the response is as specified by, or in the spirit of the IEEE Standard 754-1985 for Binary Floating-point Computations [3] (but note that the standard does not specify the behavior for certain operations such as fused multiply-add or min and max).

Note also that for parallel instructions, the single precision format does not allow for unsupported operands.

Note: *In Table 3-2, exceptions, and the conditions causing them, are listed in the decreasing order of priority. For parallel instructions, it will be considered from now on that the low part is evaluated first, followed by the high part (this is just a convention, and is not imposed by the architecture).*

Table 3-2. Conditions that Determine Occurrence of Floating-point Exceptions

Floating-point Instruction	Exception	Condition
fma.pc.sf f1 = f3, f4, f2 fms.pc.sf f1 = f3, f4, f2 fnma.pc.sf f1 = f3, f4, f2	SWA fault, Itanium™ processor specific	Any unnormal operand, and no operand is unsupported, and no operand is a NaN, and the operation does not lead to (Inf -Inf), (-Inf + Inf), Inf * 0, Inf * (-0), -Inf * 0, -Inf * (-0), 0 * Inf, -0 * Inf, 0 * (-Inf), or -0 * (-Inf)
	V	Any unsupported operand, or any SNaN operand, or no NaN operand and the operation leads to (Inf -Inf), (-Inf + Inf), Inf * 0, Inf * (-0), -Inf * 0, -Inf * (-0), 0 * Inf, -0 * Inf, 0 * (-Inf), or -0 * (-Inf)
	D	Any unnormal operand (not raised directly by the hardware, but following an Itanium processor specific SWA fault)
	SWA trap, Itanium processor specific	Result tiny, the underflow traps are disabled, and the flush-to-zero mode is not enabled
	O	Result huge
	U	Result tiny if underflow exceptions are enabled; result tiny and inexact if the underflow exceptions are disabled; the latter case will occur directly from the hardware only if the flush-to-zero mode is enabled, otherwise it would have already generated a SWA trap
	I	Result inexact
fnorm.pc.sf f1 = f3	SWA fault, Itanium processor specific	Unnormal operand, and (biased exponent in floating-point register file format is 0 or denormal exceptions are enabled), and the operand is not unsupported, and the operand is not a NaN. Note that in floating-point register file format, an unnormal with a biased exponent of 0 is equivalent to the same unnormal with the biased exponent of 0xc001 (0xc001 = 0xffff - 0x3ffe), which corresponds to an unbiased decimal exponent of -16382, the minimum value for 15-bit exponents
	V	Unsupported operand, or SNaN operand
	D	Unnormal operand (not raised directly by the hardware, but following an Itanium processor specific SWA fault)
	SWA Trap, Itanium processor specific	Result tiny, the underflow traps are disabled, and the flush-to-zero mode is not enabled

Table 3-2. Conditions that Determine Occurrence of Floating-point Exceptions (Cont'd)

Floating-point Instruction	Exception	Condition
	O	Result huge
	U	Result tiny if underflow exceptions are enabled; result tiny and inexact if they are disabled; the latter case will occur directly from the hardware only if the flush-to-zero mode is enabled, otherwise it would have already generated a SWA trap
	I	Result inexact
fpma.pc.sf f1 = f3, f4, f2 fpms.pc.sf f1 = f3, f4, f2 fpnma.pc.sf f1 = f3, f4, f2	SWA fault, Itanium processor specific	Any denormal operand, and no operand is a NaN, and the operation does not lead to (+Inf - Inf), (-Inf + Inf), Inf * 0, Inf * (-0), -Inf * 0, -Inf * (-0), 0 * Inf, -0 * Inf, 0 * (-Inf), or -0 * (-Inf)
	V	Any SNaN operand, or no NaN operand and the operation leads to (-Inf + Inf), (-Inf + Inf), Inf * 0, Inf * (-0), -Inf * 0, -Inf * (-0), 0 * Inf, -0 * Inf, 0 * (-Inf), or -0 * (-Inf)
	D	Any denormal operand (not raised directly by the hardware, but following an Itanium processor specific SWA fault)
	SWA trap, Itanium processor specific	Result tiny, the underflow traps are disabled, and the flush-to-zero mode is not enabled
	O	Result huge
	U	Result tiny if underflow exceptions are enabled; result tiny and inexact if they are disabled; the latter case will occur directly from the hardware only if the flush-to-zero mode is enabled, otherwise it would have already generated a SWA trap
	I	Result inexact
fmax.sf f1 = f2, f3 fmin.sf f1 = f2, f3 famax.sf f1 = f2, f3 famin.sf f1 = f2, f3	SWA Fault, Itanium processor specific	Any unnormal operand, and no operand is unsupported, and no operand is a NaN
	V	Any unsupported operand, or any NaN operand
	D	Any unnormal operand (not raised directly by the hardware, but following an Itanium processor specific SWA fault)
fpmax.sf f1 = f2, f3 fpmin.sf f1 = f2, f3 fpamax.sf f1 = f2, f3 fpamin.sf f1 = f2, f3	SWA Fault, Itanium processor specific	Any denormal operand, and no operand is a NaN
	V	Any NaN operand
	D	Any denormal operand (not raised directly by the hardware, but following an Itanium processor specific SWA fault)
fcmp.frel.fctype.sf p1, p2 = f2, f3	SWA Fault, Itanium processor specific	Any unnormal operand, and no operand is unsupported, and no operand is a NaN
	V	Any unsupported operand, or any SNaN operand, or (any QNaN operand if 'frel' is one of lt, le, nlt, nle)
	D	Any unnormal operand (not raised directly by the hardware, but following an Itanium processor specific SWA fault)
fpcmp.frel.sf f1 = f2, f3	SWA Fault, Itanium processor specific	Any denormal operand, and no operand is a NaN
	V	Any SNaN operand, or (any QNaN operand if 'frel' is one of lt, le, nlt, nle)

Table 3-2. Conditions that Determine Occurrence of Floating-point Exceptions (Cont'd)

Floating-point Instruction	Exception	Condition
	D	Any denormal operand (not raised directly by the hardware, but following an Itanium processor specific SWA fault)
fcvt.fx.sf f1 = f2 fcvt.fxu.sf f1 = f2 fcvt.fx.trunc.sf f1 = f2 fcvt.fxu.trunc.sf f1 = f2	SWA Fault, Itanium processor specific	Unnormal operand, and the operand is not unsupported, and the operand is not a NaN
	V	Unsupported operand, or NaN operand, or input is too large in absolute value
	D	Unnormal operand (not raised directly by the hardware, but following an Itanium processor specific SWA fault)
	I	Result inexact
fpcvt.fx.sf f1 = f2 fpcvt.fxu.sf f1 = f2 fpcvt.fx.trunc.sf f1 = f2 fpcvt.fxu.trunc.sf f1 = f2	SWA Fault, Itanium processor specific	Denormal operand, and the operand is not a NaN
	V	Any NaN operand, or any input is too large in absolute value
	D	Any denormal operand (not raised directly by the hardware, but following an Itanium processor specific SWA fault)
	I	Result inexact
frcpa.sf f1, p2 = f2, f3	SWA fault, Itanium processor specific	any unnormal operand, and no operand is unsupported, and no operand is a NaN, and the operation is not Inf/Inf, and it is not [pseudo]0/[pseudo]0, and it is not (non-zero normal)/[pseudo]0, and it is not (non-pseudo 0 unnormal)/[pseudo]0 (with any combination of signs; the square brackets indicate an optional component)
	V	Any unsupported operand, or any SNaN operand, or the operation is Inf/Inf, or it is [pseudo]0/[pseudo]0 (with any combination of signs)
	Z	Operation is (non-zero normal)/[pseudo]0, or (non-pseudo 0 unnormal)/[pseudo]0 (with any combination of signs)
	D	Any unnormal operand (not raised directly by the hardware, but following an Itanium processor specific SWA fault)
	SWA fault, arch. mandated	For floating-point register file format, if $e_b \leq e_{min} - 1$, or $e_b \geq e_{max} - 2$, or $e_a - e_b \geq e_{max}$, or $e_a - e_b \leq e_{min} + 1$, or $e_a \leq e_{min} + N - 1$ (N = 64 is the significand precision)
	O	Result huge; this can only occur after processing an architecturally mandated SWA fault (not directly raised by the hardware), when the result of the divide operation is provided by the SWA handler for frcpa, and the output predicate is cleared
	U	Result tiny if underflow exceptions are enabled; result tiny and inexact if underflow exceptions are disabled; this can only occur after processing an architecturally mandated SWA fault (not directly raised by the hardware), when the result of the divide operation is provided by the SWA handler for frcpa, and the output predicate is cleared
	I	Result inexact; this can only occur after processing an architecturally mandated SWA fault (not directly raised by the hardware), when the result of the divide operation is provided by the SWA handler for frcpa, and the output predicate is cleared
fprcpa.sf f1, p2 = f2, f3	SWA fault, Itanium processor specific	Any denormal operand, and no operand is a NaN, and the operation is not Inf/Inf, and it is not 0/0, and it is not (non-zero normal)/0 (with any combination of signs)

Table 3-2. Conditions that Determine Occurrence of Floating-point Exceptions (Cont'd)

Floating-point Instruction	Exception	Condition
	V	Any SNaN operand, or the operation is Inf/Inf, or 0/0 (with any combination of signs)
	Z	Operation is non-zero normal/0, or denormal/0 (with any combination of signs)
	D	Any denormal operand (not raised directly by the hardware, but following an Itanium processor specific SWA fault) Note that if $e_b \leq e_{min} - 1$, or $e_b \geq e_{max} - 2$, or $e_a - e_b \geq e_{max}$, or $e_a - e_b \leq e_{min} + 1$, or $e_a \leq e_{min} + N - 1$ and no SWA, invalid, divide-by-zero, or denormal exception occurs, then fprcpa clears the output predicate and returns the best possible approximations in the floating-point output register (see Section 5.1) (N = 24 is the significand precision)
frsqrta.sf f1, p2 = f3	SWA fault, Itanium processor specific	Unnormal operand (positive or negative), and the operand is not unsupported, and the operand is not a NaN, and the operand is not -Inf, and (the operand is not strictly negative normal or it is a pseudo-zero) Note: the last condition allows negative pseudo-zeros to raise a SWA request
	V	Operand is unsupported, or the operand is SNaN , or the operand is -Inf, or (the operand is strictly negative finite and it is not a pseudo-zero) ['strictly negative' excludes -0.0]; note that if the operand is a non-zero negative unnormal (or denormal), the invalid fault is not raised directly by the hardware, but it follows an Itanium processor specific SWA fault
	D	Unnormal operand, strictly positive (not raised directly by the hardware, but following an Itanium processor specific SWA fault)
	SWA fault, arch. mandated	for floating-point register file format, if $e_a \leq e_{min} + N - 1$ (N = 64 is the significand precision)
	I	Result inexact; this can only occur after processing an architecturally mandated SWA fault (not directly raised by the hardware), when the result of the square root operation is provided by the SWA handler, and the output predicate is cleared
fprsqrta.sf f1, p2 = f3	SWA fault, Itanium processor specific	Denormal operand (negative or positive), and the operand is not a NaN, the operand is not -Inf, and the operand is not a strictly negative normal floating-point number
	V	Operand is SNaN , or the operand is -Inf, or the operand is strictly negative finite; note that if the operand is a non-zero negative denormal, the invalid fault is not raised directly by the hardware, but it follows an Itanium processor specific SWA fault
	D	Denormal operand, strictly positive (not raised directly by the hardware, but following an Itanium processor specific SWA fault Note that if $e_a \leq e_{min} + N - 1$ and no SWA, invalid, or denormal exception occurs, then fprsqrta clears the output predicate and returns the best possible approximation in the floating-point output register (see also 5.3) (N = 24 is the significand precision)

Note that Table 3-2 has a separate entry for *fnorm*, that emphasizes the conditions leading to floating-point exceptions, and the differences between *fma* and *fnorm* in raising Itanium processor specific SWA faults. For other purposes, *fnorm* behaves like a pseudo-op of *fma*.

If an enabled exception (V, Z, D, O, U, I) is raised, the result is provided by the corresponding user handler (provided execution of the application program containing the excepting instruction is continued). For Software Assistance cases, the result is the IEEE mandated one, unless an enabled IEEE exception is raised as a result of the computation. The user then provides the result, via a user

exception handler (again, if the application program containing the excepting instruction is continued).

If the SWA exception handler is invoked or a user floating-point exception handler is reached after an enabled exception has been raised, the indication regarding the cause of the exception is contained in the ISR code (the 16 least-significant bits of the ISR register). There are different interruption vectors for floating-point exception faults and for floating-point exception traps. The FPSR register is changed by the excepting instruction before the handler is invoked to update status flags only for floating-point traps (SWA trap, overflow, underflow, or inexact). The handler may make other changes to the FPSR, which will become effective when execution of the application program that caused the exception is resumed.

If any of the conditions shown in the last column of Table 3-2 is met for IEEE exceptions (V, Z, O, U, I) when the corresponding exception is disabled, then the appropriate FPSR flags are set (as specified by the IEEE-754 Standard for Binary Floating-point Computations [3]), and the result is the IEEE mandated one. A special case occurs for underflow: the U flag in the FPSR is set only if the result is both tiny and inexact, while an enabled underflow exception is raised if the result is merely tiny. Tininess is established after rounding to the destination precision, but with unbounded exponent, i.e. after the first IEEE rounding (the result of which, if non-zero, will be normalized if possible). Inexactness is established either after the first IEEE rounding (once the result is inexact at this stage it will not become exact through a second rounding), or after the second IEEE rounding, to the destination precision and a bounded exponent. This second and final result will be different from the first one only if the result is tiny and significant bits are lost through denormalization, or if the result is huge (then the delivered result is infinity or the largest floating-point value in the destination format, depending on the rounding mode).

Note that if a denormal/unnormal operand is encountered, and the denormal exceptions are disabled, a SWA fault is raised and the result will be provided by the SWA fault handler (the IA-64 Floating-point Emulation Library). The same holds for architecturally mandated SWA faults or for SWA traps.

An explanation is required for the denormal and underflow exceptions. In Table 3-2, “any unnormal operand” or “any denormal operand” (with the specified restrictions on the other operands), leads to a SWA (Software Assistance) fault. The same condition, “any unnormal operand” or “any denormal operand”, is specified for denormal exceptions. In the Itanium processor implementation, the hardware never raises a denormal exception for these conditions, raising instead a SWA fault. If the denormal exceptions are disabled, the SWA handler (the IA-64 Floating-point Emulation Library) returns the result. If the denormal exceptions are enabled, the SWA handler just returns an exception code modified from SWA fault to denormal exception, and the OS kernel trap handler will have to invoke the corresponding (user registered) exception handler.

A similar situation exists for underflow exceptions, when one of the *fma*, *fnorm*, *fms, fnma*, *fpma*, *fpms*, or *fpnma* instructions has a tiny result. An underflow trap will be taken if the “result is tiny” (see Table 3-2) and the underflow traps are enabled. If the underflow traps are disabled and the result is tiny, a SWA trap will be raised instead. In this case, if the result is tiny but exact, the U flag will not be set in the FPSR (it will preserve its previous value). If the result is tiny and inexact, both U and I flags will be set in the FPSR. If the I traps are enabled, the inexact trap will be taken (raised from within the SWA trap handler).

The next subsections specify the masked (disabled) and the unmasked (enabled) responses of the Itanium processor (and IA-64 in general) arithmetic instructions to floating-point exceptions, in the order: SWA faults, V, Z, D, SWA traps, O, U, I. In each case, it is assumed that an enabled floating-point exception of higher priority does not occur. The masked and the unmasked response are placed in the same table as in the simpler cases. Note that the meaning of “updated” and “assigned” is the same as in Section 3.3 (“sticky” status bits are “updated” by a logical OR; other state bits are “assigned” a value of 0 or 1).

3.5 Response to SWA Faults

Table 3-3 lists the actions performed by the IA-64 Floating-point Emulation Library in response to a SWA fault. The emulation library is invoked by the operating system kernel, reached in this case via the floating-point fault vector.

Table 3-3. Response of the IA-64 Arithmetic Instructions to SWA Faults

Floating-point Instruction	Exception
fma.pc.sf f1 = f3, f4, f2 fms.pc.sf f1 = f3, f4, f2 fnma.pc.sf f1 = f3, f4, f2 fpma.pc.sf f1 = f3, f4, f2 fpms.pc.sf f1 = f3, f4, f2 fpnma.pc.sf f1 = f3, f4, f2	SWA fault, Itanium processor specific

```
if the denormal exceptions are enabled
        clear the SWA bit and set the D bit in the ISR code
        raise a denormal exception
else
        compute the infinitely precise result rounded to the destination precision,
                with unbounded exponent (first IEEE rounding)
        if the result is huge
                if the overflow exceptions are enabled
                        clear the SWA bit, set the O bit and assign the I and fpa
                                bits in the ISR code
                        set the D and O bits, and update the I bit in FPSR.sfx
                        truncate the exponent of the result to 17 bits
                        raise an overflow exception
                else
                        calculate the IEEE mandated result, according to the
                                rounding mode (a correctly signed infinity or largest
                                normal floating-point number) (second IEEE rounding)
                        if the inexact exceptions are enabled
                                clear the SWA bit, set the I bit, and assign the fpa
                                        bit in the ISR code
                                set the D, O, and I bits in FPSR.sfx
                                raise an inexact exception
                        else
                                set the D, O and I bits in FPSR.sfx
                                return the IEEE mandated result
                        endif
                endif
        else if the result is tiny
                if the underflow exceptions are enabled
                        clear the SWA bit, set the U bit, and assign the I and fpa
                                bits in the ISR code
                        set the D bit, and update the U and I bits in FPSR.sfx
                        truncate the exponent of the result to 17 bits
                        raise an underflow exception
                else
                        if the ftz mode is disabled
                                denormalize the result (a correctly signed zero,
                                denormal, or smallest normal is obtained) (second
                                IEEE rounding)
                                if the result is inexact
```

```
                                        if the inexact exceptions are enabled
                                                clear the SWA bit, set the I bit, and
                                                        assign the fpa bit in the ISR
                                                        code
                                                set the D, U, and I bits in FPSR.sfx
                                                raise an inexact exception
                                        else
                                                set the D, U and I bits in FPSR.sfx
                                                return the result
                                        endif
                                else
                                        set the D bit in FPSR.sfx
                                        return the result
                                endif
                        else
                                set the result to the correctly signed zero
                                if the inexact exceptions are enabled
                                        clear the SWA bit, set the I bit, and clear
                                                the fpa bit in the ISR code
                                        set the D, U, and I bits in FPSR.sfx
                                        raise an inexact exception
                                else
                                        set the D, U and I bits in FPSR.sfx
                                        return the result
                                endif
                        endif
                endif
        else
                compute the result rounded to the destination precision,
                        with bounded exponent range (second IEEE rounding)
                if the result is inexact
                        if the inexact exceptions are enabled
                                clear the SWA bit, set the I bit, and assign the
                                        fpa bit in the ISR code
                                set the D and I bits in FPSR.sfx
                                raise an inexact exception
                        else
                                set the D and I bits in FPSR.sfx
                                return the result
                        endif
                else
                        set the D bit in FPSR.sfx
                        return the result
                endif
        endif
endif
```

fmax.sf f1 = f2, f3 fmin.sf f1 = f2, f3 famax.sf f1 = f2, f3 famin.sf f1 = f2, f3 fpmax.sf f1 = f2, f3 fpmin.sf f1 = f2, f3 fpamax.sf f1 = f2, f3 fpamin.sf f1 = f2, f3	SWA Fault, Itanium processor specific

```
if the denormal exceptions are enabled
```

```
        clear the SWA bit and set the D bit in the ISR code
        raise a denormal exception
else
        compute the result (equal to one of the two inputs)
        set the D bit in FPSR.sfx
        return result
endif
```

fcmp.frel.fctype.sf p1, p2 = f2, f3 fpcmp.frel.sf f1 = f2, f3	SWA fault, Itanium processor specific

```
if the denormal exceptions are enabled
        clear the SWA bit and set the D bit in the ISR code
        raise a denormal exception
else
        compute the result predicates
        set the D bit in FPSR.sfx
        return result
endif
```

fcvt.fx.sf f1 = f2 fcvt.fxu.sf f1 = f2 fcvt.fx.trunc.sf f1 = f2 fcvt.fxu.trunc.sf f1 = f2 fpcvt.fx.sf f1 = f2 fpcvt.fxu.sf f1 = f2 fpcvt.fx.trunc.sf f1 = f2 fpcvt.fxu.trunc.sf f1 = f2	SWA fault, Itanium processor specific

```
if the denormal exceptions are enabled
        clear the SWA bit and set the D bit in the ISR code
        raise a denormal exception
else
        compute the result
        if the result is inexact
                if the inexact exceptions are enabled
                        clear the SWA bit, set the I bit, and clear the
                                fpa bit in the ISR code
                        set the D and I bits in FPSR.sfx
                        raise an inexact exception
                else
                        set the D and I bits in FPSR.sfx
                        return the result
                endif
        else
                set the D bit in FPSR.sfx
                return the result
        endif
endif
```

frcpa.sf f1, p2 = f2, f3	SWA fault, Itanium processor specific or arch. mandated

```
if any input operand is unnormal and the denormal exceptions are enabled
        clear the SWA bit and set the D bit in the ISR code
```

```
        raise a denormal exception
else
        if any input operand is unnormal, set the D bit in FPSR.sfx
        if this is (an architecturally mandated SWA fault) or (an Itanium processor
                specific SWA fault and the mathematical conditions for an
                architecturally mandated SWA fault are met)
                compute the infinitely precise result for the divide operation
                        rounded to the destination precision, with unbounded
                        exponent (first IEEE rounding)
                if the result is huge
                        if the overflow exceptions are enabled
                                clear the SWA bit, set the O and assign the I and
                                        fpa bits in the ISR code
                                set the O bit, and update the I and fpa bits in
                                        FPSR.sfx
                                truncate the exponent of the result to 17 bits
                                raise an overflow exception
                        else
                                calculate the IEEE mandated result according to the
                                        rounding mode (a correctly signed infinity or
                                                largest normal floating-point number)
                                                (second IEEE rounding)
                                if the inexact exceptions are enabled
                                        clear the SWA bit, set the I bit, and assign
                                                the fpa bit in the ISR code
                                        set the O and I bits in FPSR.sfx
                                        raise an inexact exception
                                else
                                        set the O and I bits in FPSR.sfx
                                        return the IEEE mandated result and a clear
                                                output predicate
                                endif
                        endif
                else if the result is tiny
                        if the underflow exceptions are enabled
                                clear the SWA bit, set the U bit, and assign the
                                        I and fpa bits in the ISR code
                                update the U, I and fpa bits in FPSR.sfx
                                truncate the exponent of the result to 17 bits
                                raise an underflow exception
                        else
                                if the ftz mode is disabled
                                        denormalize the result (a correctly signed
                                                zero, denormal, or smallest normal is
                                                obtained) (second IEEE rounding)
                                        if the result is inexact
                                                if the inexact exceptions are enabled
                                                        clear the SWA bit, set I bit,
                                                                and assign the fpa bit
                                                                in the ISR code
                                                        set the U and I bits in
                                                                FPSR.sfx
                                                        raise an inexact exception
                                                else
                                                        set the U and I bits in
                                                                FPSR.sfx
```

```
                                                        return the result and a clear
                                                                output predicate
                                                endif
                                        else
                                                return the result and a clear output
                                                        predicate
                                        endif
                                else
                                        set the result to the correctly signed zero
                                        if the inexact exceptions are enabled
                                                clear the SWA bit, set the I bit, and
                                                        assign the fpa bit in the ISR
                                                                code
                                                set the U and I bits in FPSR.sfx
                                                raise an inexact exception
                                        else
                                                set U and I bits in FPSR.sfx
                                                return the result and a clear output
                                                        predicate
                                        endif
                                endif
                        endif
                else
                        compute the result rounded to the destination precision,
                                with bounded exponent range (second IEEE rounding)
                        if the result is inexact
                                if the inexact exceptions are enabled
                                        clear the SWA bit, set the I bit, and assign
                                                the fpa bit in the ISR code
                                        set the I bit in FPSR.sfx
                                        raise an inexact exception
                                else
                                        set the I bit in FPSR.sfx
                                        return the result and a clear output
                                                predicate
                                endif
                        else
                                return the result and a clear output predicate
                        endif
                endif
        else (if this is an Itanium processor specific SWA fault and the
                mathematical conditions for an architecturally mandated SWA fault
                are not met)
                return the 11-bit table approximation for the inverse of the
                        denominator (second argument), and an output predicate set
                                to 1
        endif
endif
```

---

fprcpa.sf f1, p2 = f2, f3 — SWA fault, Itanium processor specific

```
if the denormal exceptions are enabled
        clear the SWA bit and set the D bit in the ISR code
        raise a denormal exception
```

```
else
        set the D bit in FPSR.sfx
        return the 11-bit table approximation for the inverse of the denominator
                (which may range from zero to infinity) and a clear output predicate
endif
```

In situations similar to those that lead to an architecturally mandated SWA fault for frcpa (this does not exclude cases with denormal input operands covered above), the fprcpa instruction will instead clear the output predicate, and will set the value of the output floating-point register as shown below (assume a/b is to be computed):

```
if e_b ≤ e_min - 1
        result = Inf, with the sign of the denominator
else if e_b ≥ e_max - 2
        result = 0, with the sign of the denominator
else if e_a - e_b ≥ e_max or e_a - e_b ≤ e_min + 1 or e_a ≤ e_min + N - 1
        result = 11-bit table approximation for the inverse of b
```

frsqrta.sf f1, p2 = f3	SWA fault, Itanium processor specific or arch. mandated

```
if the invalid exceptions are enabled and the input operand is a strictly negative
                unnormal
        clear the SWA bit and set the V bit in the ISR code
        raise an invalid exception
else if the input operand is a strictly positive unnormal and the
                denormal exceptions are enabled
        clear the SWA bit and set the D bit in the ISR code
        raise a denormal exception
else if the input operand is unnormal set the D bit in FPSR.sfx
        if an architecturally mandated SWA fault or an Itanium processor specific
                SWA fault and the mathematical condition for an architecturally
                mandated SWA fault is met
                compute the infinitely precise result for the square root operation
                        rounded to the destination precision, with unbounded
                        exponent (first IEEE rounding)
                compute the result rounded to the destination precision,
                        with bounded exponent range (second IEEE rounding)
                if the inexact traps are disabled or the result is exact
                        if the result is inexact
                                set the I bit in FPSR.sfx
                        endif
                        return the result and a clear output predicate
                else (if the result is inexact and the inexact traps are enabled)
                        clear the SWA bit, set the I bit, and assign the fpa bit in
                                the ISR code
                        set the I bit in FPSR.sfx
                        raise an inexact exception
                endif
        else if an Itanium processor specific SWA fault and the mathematical
                condition for an architecturally mandated SWA fault is not met
                return the 11-bit table approximation for the inverse of the square
                        root of the input argument and an output predicate set to 1
        endif
endif
```

fprsqrta.sf f1, p2 = f3	SWA fault, Itanium processor specific

```
if the invalid exceptions are enabled and the input operand is a strictly negative
                denormal
        clear the SWA bit and set the V bit in the ISR code
        raise an invalid exception
else if the denormal exceptions are enabled
        clear the SWA bit and set the D bit in the ISR code
        raise a denormal exception
else
        set the D bit in FPSR.sfx
        return the 11-bit table approximation for the inverse of the square root
                of the input argument and a clear output predicate
endif
```

Note: In situations similar to those that lead to an architecturally mandated SWA fault for frsqrta (this does not exclude cases with a denormal input operand covered above), the fprsqrta instruction will instead clear the output predicate, and will set the value of the output floating-point register as shown below (assume sqrt (a) is to be computed):

```
if e_a ≤ e_min + N - 1
        result = 11-bit table approximation for the inverse of sqrt(a)
```

3.6 Response to Invalid Faults

The IEEE-754 Standard for Binary Floating-point Computations [3] does not specify uniquely the value of the result when an invalid exception is raised in case one of the operands is a signaling NaN, and the invalid exceptions are disabled. In addition, the IA-64 assembly instructions do not match exactly the operations described in the standard. Table 3-4 covers this and other cases, specifying the masked response of the IA-64 arithmetic instructions to invalid exceptions. The notation Q(fn) in Table 3-4 signifies a ‘‘quieted’’ NaN, assuming that the floating-point register fn contains a signaling NaN (this means changing the second most significant bit in the significand of the NaN in fn from 0 to 1). QNaN Indefinite is the NaN having the 82-bit pattern of 0x3ffffc000000000000000, which applies for scalar instructions, and the 32-bit pattern of 0xffc00000 which applies for parallel instructions.

Table 3-4 lists the actions performed by the hardware as part of the masked response to an invalid fault. In addition, the invalid exception status flag, V, is set in the appropriate status field of the FPSR.

Table 3-4. Masked Response of the IA-64 Arithmetic Instructions to Invalid Exceptions (listed in decreasing order of their priority)

Floating-point Instruction	Exception	Condition and Result
fma.pc.sf f1 = f3, f4, f2 fms.pc.sf f1 = f3, f4, f2 fnma.pc.sf f1 = f3, f4, f2	V	Any unsupported operand f1 = QNaN Indefinite Any SNaN operand if f4 is SNaN, then f1 = Q (f4) else if f2 is SNaN, then f1 = Q (f2) else if f3 is SNaN,then f1 = Q (f3) Operation leads to (-Inf + Inf), (+Inf -Inf), Inf * 0, Inf * (-0), -Inf * 0, -Inf * (-0), 0 * Inf, -0 * Inf, 0 * (-Inf), or -0 * (-Inf) f1 = QNaN Indefinite
fpma.pc.sf f1 = f3, f4, f2 fpms.pc.sf f1 = f3, f4, f2 fpnma.pc.sf f1 = f3, f4, f2	V	Any SNaN operand if f4 is SNaN, then f1 = Q (f4) else if f2 is SNaN, then f1 = Q (f2) else if f3 is SNaN, then f1 = Q (f3) Operation leads to (-Inf + Inf), (+Inf -Inf), Inf * 0, Inf * (-0), -Inf * 0, -Inf * (-0), 0 * Inf, -0 * Inf, 0 * (-Inf), or -0 * (-Inf) f1 = QNaN Indefinite
fmax.sf f1 = f2, f3 fmin.sf f1 = f2, f3 famax.sf f1 = f2, f3 famin.sf f1 = f2, f3	V	Any unsupported operand or any NaN operand f1 = f3
fpmax.sf f1 = f2, f3 fpmin.sf f1 = f2, f3 fpamax.sf f1 = f2, f3 fpamin.sf f1 = f2, f3	V	Any NaN operand f1 = f3
fcmp.frel.fctype.sf p1, p2 = f2, f3	V	Any unsupported operand if ‘frel’ is unord, then p1, p2 = 1, 0 else p1, p2 = 0, 1 Any SNaN operand, or (any QNaN operand and ‘frel’ is one of lt, le, nlt, nle) if ‘frel’ is unord, then p1, p2 = 1, 0 else p1, p2 = 0, 1
fpcmp.frel.sf f1 = f2, f3	V	Any SNaN operand, or (any QNaN operand if ‘frel’ is one of lt, le, nlt, nle) if ‘frel’ is unord, then f1 = 0xffffffff else f1 = 0x00000000
fcvt.fx.sf f1 = f2 fcvt.fxu.sf f1 = f2 fcvt.fx.trunc.sf f1 = f2 fcvt.fxu.trunc.sf f1 =f2	V	Unsupported, or NaN operand, or too large in absolute value f1 = Integer Indefinite (0x1003e8000000000000000)
fpcvt.fx.sf f1 = f2 fpcvt.fxu.sf f1 = f2 fpcvt.fx.trunc.sf f1 = f2 fpcvt.fxu.trunc.sf f1 = f2	V	Any NaN operand, or too large in absolute value f1 = Integer Indefinite (32-bit code 0x80000000 in the appropriate half; exponent: 0x1003e)
frcpa.sf f1, p2 = f2, f3	V	Any unsupported operand f1 = QNaN Indefinite, p2 = 0 Any SNaN operand if f2 is SNaN, then f1 = Q (f2), p2 = 0 else f1 = Q (f3), p2 = 0 Operation is Inf/Inf, or it is [pseudo]0/[pseudo]0 f1 = QNaN Indefinite, p2 = 0

Table 3-4. Masked Response of the IA-64 Arithmetic Instructions to Invalid Exceptions (listed in decreasing order of their priority) (Cont'd)

Floating-point Instruction	Exception	Condition and Result
fprcpa.sf f1, p2 = f2, f3	V	Any SNaN operand if f2 is SNaN, then f1 = Q (f2), p2 = 0 else f1 = Q (f3), p2 = 0 Operation is Inf/Inf, or 0/0 (with any combination of signs) f1 = QNaN Indefinite, p2 = 0
frsqrta.sf f1, p2 = f3	V	Unsupported operand f1 = QNaN Indefinite, p2 = 0 Operand is SNaN f1 = Q (f3), p2 = 0 Operand is -Inf, or (the operand is strictly negative and it is not a pseudo-zero) f1 = QNaN Indefinite, p2 = 0
fprsqrta.sf f1, p2 = f3	V	Operand is SNaN f1 = Q (f3), p2 = 0 Operand is -Inf, or the operand is strictly negative (this excludes -0) f1 = QNaN Indefinite, p2 = 0

In Table 3-4, for parallel instructions, it is implied that the result specified applies only to the half (halves) that corresponds to an invalid exception being raised.

The unmasked response of the IA-64 arithmetic instructions listed in Table 3-4 to invalid exceptions is to leave the operands unchanged, and to set the V bit in the ISR code. The operating system kernel, reached via the floating-point fault vector, will then invoke the user floating-point exception handler, if one has been registered. Otherwise, the default action should be to terminate the application.

3.7 Valid Operations with NaNs

Rules similar to those in Table 3-4 establish the value of the result in case one or more input operands are quiet NaNs (QNaNs), and an invalid exception is not raised (for fcmp and fpcmp, this means that 'frel' is none of lt, le, nlt, or nle; otherwise, fcmp or fpcmp will raise an invalid exception). Table 3-5 lists the value of the hardware-provided result in such cases, i.e. when at least one of the operands is a quiet NaN, and no exception of higher priority applies (see the exception priorities at the beginning of Section 3.3), regardless of whether the invalid exceptions are enabled or not. Table 3-5 is not related to any floating-point exception, but it is included here for completeness because responding to QNaNs fits as priority between responding to invalid exceptions due to unsupported or SNaN operands and to invalid exceptions due to causes other than unsupported or SNaN operands. The results listed in Table 3-5 are generated by the hardware.

Table 3-5. Result of Floating-point Arithmetic Instructions for QNaN Input(s), in the Absence of Floating-point Exceptions

Floating-point Instruction	Condition and Result
fma.pc.sf f1 = f3, f4, f2 fms.pc.sf f1 = f3, f4, f2 fnma.pc.sf f1 = f3, f4, f2 fms.pc.sf f1 = f3, f4, f2	Any QNaN operand if f4 is QNaN, then f1 = f4 else if f2 is QNaN, then f1 = f2 else if f3 is QNaN, then f1 = f3

Table 3-5. Result of Floating-point Arithmetic Instructions for QNaN Input(s), in the Absence of Floating-point Exceptions

Floating-point Instruction	Condition and Result
fpma.pc.sf f1 = f3, f4, f2 fpms.pc.sf f1 = f3, f4, f2 fpnma.pc.sf f1 = f3, f4, f2	Any QNaN operand if f4 is QNaN, then f1 = f4 else if f2 is QNaN, then f1 = f2 else if f3 is QNaN, then f1 = f3
fcmp.frel.fctype.sf p1, p2 = f2, f3	Any QNaN operand, and 'frel' is none of lt, le, nlt, nle if 'frel' is unord, then p1, p2 = 1, 0 else p1, p2 = 0, 1
fpcmp.frel.sf f1 = f2, f3	Any QNaN operand, and 'frel' is none of lt, le, nlt, nle if 'frel' is unord, then f1 = 0xffffffff else f1 = 0x00000000
frcpa.sf f1, p2 = f2, f3	Any QNaN operand if f2 is QNaN, then f1 = f2, p2 = 0 else f1 = f3, p2 = 0
fprcpa.sf f1, p2 = f2, f3	Any QNaN operand if f2 is QNaN, then f1 = f2, p2 = 0 else f1 = f3, p2 = 0
frsqrta.sf f1, p2 = f3	Operand is QNaN f1 = f3, p2 = 0
fprsqrta.sf f1, p2 = f3	Any operand is QNaN f1 = f3, p2 = 0

3.8 Response to Divide-by-Zero Faults

Table 3-6 lists the actions performed by the hardware as part of the masked response to a divide-by-zero fault.

Table 3-6. Masked Response of the IA-64 Arithmetic Instructions to Divide-by-Zero Exceptions

Floating-point Instruction	Exception	Result
frcpa.sf f1, p2 = f2, f3 fprcpa.sf f1, p2 = f2, f3	Z	if a is non-[pseudo-]zero finite and b is [pseudo-] zero, then f1=Inf, with the sign of a/b, and p2=0 (the IEEE 754 Standard mandated result); set the Z bit in FPSR.sfx

The unmasked response of the IA-64 arithmetic instructions listed in Table 3-6 to divide-by-zero exceptions is to leave the operands unchanged, and to set the Z bit in the ISR code. The operating system kernel, reached via the floating-point fault vector, will then invoke the user floating-point exception handler, if one has been registered. Otherwise, the default action should be to terminate the application.

3.9 Response to Denormal Faults

Both the masked and the unmasked response to denormal faults originate in the IA-64 Floating-Point Emulation Library (see Figure 3-3). The emulation library is invoked by the operating system kernel, reached via the floating-point fault vector, for an Itanium processor specific SWA fault.

The masked response of the IA-64 arithmetic instructions to denormal exceptions is identical to the response to Itanium processor specific SWA faults, when the denormal exceptions are masked (see Table 3-3 in Section 3.5), which include providing a result and setting the D status flag in the appropriate status field of the FPSR.

The unmasked response of the IA-64 arithmetic instructions to denormal exceptions is to leave the operands unchanged, and to set the D bit in the ISR code. The IA-64 Floating-point Emulation Library performs these operations, and returns a denormal exception code to the operating system kernel, which will then invoke the user floating-point exception handler, if one has been registered. Otherwise, the default action should be to terminate the application. The unmasked response to denormal exceptions is included in Table 3-3 of Section 3.5 above.

3.10 Response to SWA Traps

Table 3-7 summarizes the actions performed by the IA-64 Floating-point Emulation Library (SWA handler) in response to an Itanium processor specific SWA trap. The library is invoked by the operating system kernel, reached via the floating-point trap vector, for a SWA trap. The result of the first IEEE rounding is already available.

Table 3-7. Response of the IA-64 Arithmetic Instructions to Itanium™ Processor Specific SWA Traps

Floating-point Instruction	Exception
fma.pc.sf f1 = f3, f4, f2 fms.pc.sf f1 = f3, f4, f2 fnma.pc.sf f1 = f3, f4, f2 fpma.pc.sf f1 = f3, f4, f2 fpms.pc.sf f1 = f3, f4, f2 fpnma.pc.sf f1 = f3, f4, f2	SWA Trap, Itanium processor specific

```
"undo" the first IEEE rounding, denormalize, and round to the destination
precision, with bounded exponent range (the result is a correctly signed zero,
        denormal, or smallest normal floating-point number)
if the result is inexact
        if the inexact exceptions are enabled
                clear the SWA bit, set the I bit, and assign the fpa bit in the ISR
code
                set the U and I bits in FPSR.sfx
                raise an inexact exception
        else
                set the U and I bits in FPSR.sfx
                return the result
        endif
else
        return the result
endif
```

3.11 Response to Overflow Traps

For overflow traps, both the masked and the unmasked response are presented in Table 3-8. The actions listed are performed by the hardware or by the IA-64 Floating-point Emulation Library, depending on whether the exception was raised by the hardware or by the emulation library (in

Figure 3-4 of Section 3.3 the overflow trap can be raised only by the hardware, while in Figure 3-5, the overflow trap can be raised only from software, by the emulation library).

Table 3-8. Response of the IA-64 Arithmetic Instructions to Overflow Exceptions

Floating-point Instruction	Exception
fma.pc.sf f1 = f3, f4, f2 fms.pc.sf f1 = f3, f4, f2 fnma.pc.sf f1 = f3, f4, f2 fpma.pc.sf f1 = f3, f4, f2 fpms.pc.sf f1 = f3, f4, f2 fpnma.pc.sf f1 = f3, f4, f2 frcpa.sf f1, p2 = f2, f3	O

Masked response:

```
set the result to the correctly signed infinity or to the largest normal
       floating-point number, according to the rounding mode, as mandated by
       the IEEE 754 Standard (second IEEE rounding)
if the inexact exceptions are disabled
       set the O and I bits in FPSR.sfx
       return the IEEE mandated result
else
       set the I bit and assign the fpa bit in the ISR code
       set the O and I bits in FPSR.sfx
       raise an inexact exception
endif
```

Unmasked response:

```
truncate to 17 bits the exponent of the result from the first IEEE rounding
set the O bit and assign the I and fpa bits in the ISR code
set the O bit and update the I bit in FPSR.sfx
raise an overflow exception
```

The IEEE Standard 754 for Binary Floating-point Computations [3] requests that when raising an overflow trap, the user handler be provided with the result rounded to the destination precision, as if with unbounded exponent, but then scaled down by a factor equal to 2 raised to $3 \cdot 2^{n-2}$, where n is the number of bits in the exponent of the floating-point format used to represent the result (this will bring it close to the middle of the range covered by the particular format). The scaling factors for the various formats are determined by the following:

Number of Bits in Exponent	Exponent of the Base-2 Scaling Factor
8	$3 * 2^6 = 192 = 0xc0$
11	$3 * 2^9 = 1536 = 0x600$
15	$3 * 2^{13} = 24566 = 0x6000$
17	$3 * 2^{15} = 98304 = 0x18000$

The actual scaling is not performed by the hardware, nor by the IA-64 Floating-point Emulation Library. It is typically performed by an IEEE filter that is invoked before calling the user floating-point exception handler, if one has been registered (otherwise the default action is usually to terminate the application).

3.12 Response to Underflow Traps

For underflow traps, both the masked and the unmasked response are presented in Table 3-9. The actions listed are performed by the hardware or by the IA-64 Floating-point Emulation Library, depending on whether the exception was raised by the hardware or by the emulation library (in Figure 3-4 of Section 3.3 the underflow trap can be raised only by the hardware, while in Figure 3-5, the underflow trap can be raised only from software, by the emulation library).

Table 3-9. Response of the IA-64 Arithmetic Instructions to Underflow Exceptions

Floating-point Instruction	Exception	Result (Below)
fma.pc.sf f1 = f3, f4, f2 fms.pc.sf f1 = f3, f4, f2 fnma.pc.sf f1 = f3, f4, f2 fpma.pc.sf f1 = f3, f4, f2 fpms.pc.sf f1 = f3, f4, f2 fpnma.pc.sf f1 = f3, f4, f2 frcpa.sf f1, p2 = f2, f3	U	

Masked response:

```
"undo" the first IEEE rounding, denormalize, and round to the destination
precision, with bounded exponent range (the result is a correctly signed zero,
        denormal, or smallest normal floating-point number)
if the result is exact or the inexact exceptions are disabled
        if the result is inexact
                set the U and I bits in FPSR.sfx
        endif
        return the result
else
        set the I bit and update the fpa bit in the ISR code
        set the U and I bits in FPSR.sfx
        raise an inexact exception
endif
```

Unmasked response:

```
truncate to 17 bits the exponent of the result from the first IEEE rounding
set the U bit and assign the I and fpa bits in the ISR code
set the U bit and update the I bit in FPSR.sfx
raise an underflow exception
```

The IEEE Standard 754 for Binary Floating-point Computations [3] requests that when raising an underflow trap, the user handler be provided with the result rounded to the destination precision, as if with unbounded exponent, but then scaled up by a factor equal to 2 raised to $3 \cdot 2^{n-2}$, where n is the number of bits in the exponent of the floating-point format used to represent the result (this will bring it close to the middle of the range covered by the particular format). The scaling factors for the various formats are the same as those for unmasked overflow exceptions, listed above.

Just as for overflow, the actual scaling is not performed by the hardware, nor by the IA-64 Floating-point Emulation Library. It is typically performed by an IEEE filter that is invoked before calling the user floating-point exception handler, if one has been registered (otherwise the default action is usually to terminate the application).

3.13 Response to Inexact Traps

For inexact traps, both the masked and the unmasked response are presented in Table 3-10. The actions listed are performed by the hardware or by the IA-64 Floating-point Emulation Library, depending on whether the exception was raised by the hardware or by the emulation library (in Figure 3-4 of Section 3.3 both situations are possible, while in Figure 3-5, the inexact trap can be raised only from software, by the emulation library).

Table 3-10. Response of the IA-64 Arithmetic Instructions to Inexact Exceptions

Floating-point Instruction	Exception	Result (Below)
fma.pc.sf f1 = f3, f4, f2 fms.pc.sf f1 = f3, f4, f2 fnma.pc.sf f1 = f3, f4, f2 fpma.pc.sf f1 = f3, f4, f2 fpms.pc.sf f1 = f3, f4, f2 fpnma.pc.sf f1 = f3, f4, f2 fcvt.fx.sf f1 = f2 fcvt.fxu.sf f1 = f2 fcvt.fx.trunc.sf f1 = f2 fcvt.fxu.trunc.sf f1 = f2 fpcvt.fx.sf f1 = f2 fpcvt.fxu.sf f1 = f2 fpcvt.fx.trunc.sf f1 = f2 fpcvt.fxu.trunc.sf f1 = f2 frcpa.sf f1, p2 = f2, f3 frsqrta.sf f1, p2 = f3	I	

Masked response:

```
set the I bit in FPSR.sfx
return the result
```

Unmasked response:

```
set the I bit and update the fpa bit in the ISR code
set the I bit in FPSR.sfx
raise an inexact exception
```

The explanations in Table 3-2 through Table 3-10 above are given for scalar instructions, or for ‘one half’ for parallel ones. For parallel instructions, one or two simultaneous exception conditions per instruction are possible. If any is a SWA fault condition, it may also lead to one or two trap conditions per instruction. If only SWA faults and traps are involved (and no other enabled floating-point exception conditions), then the result for the excepting half (halves) of the instruction is provided as explained in Table 3-3 and Table 3-7 (response to SWA faults and to SWA traps). The non-excepting half, if any, is executed if it is associated with a SWA fault in the other half, or its result is left unchanged if it is associated with a SWA trap. If any enabled exceptions occur while executing a parallel instruction, then a user handler has to be invoked in order to generate a result. Support for handling such situations is provided through an IEEE Floating-point Exception Filter (details on its operation are given in a separate man page). The role of the filter is to simplify handling of unmasked floating-point exceptions in user code (it shields the user from many machine specific aspects, and it simplifies and increases the portability of the user code).

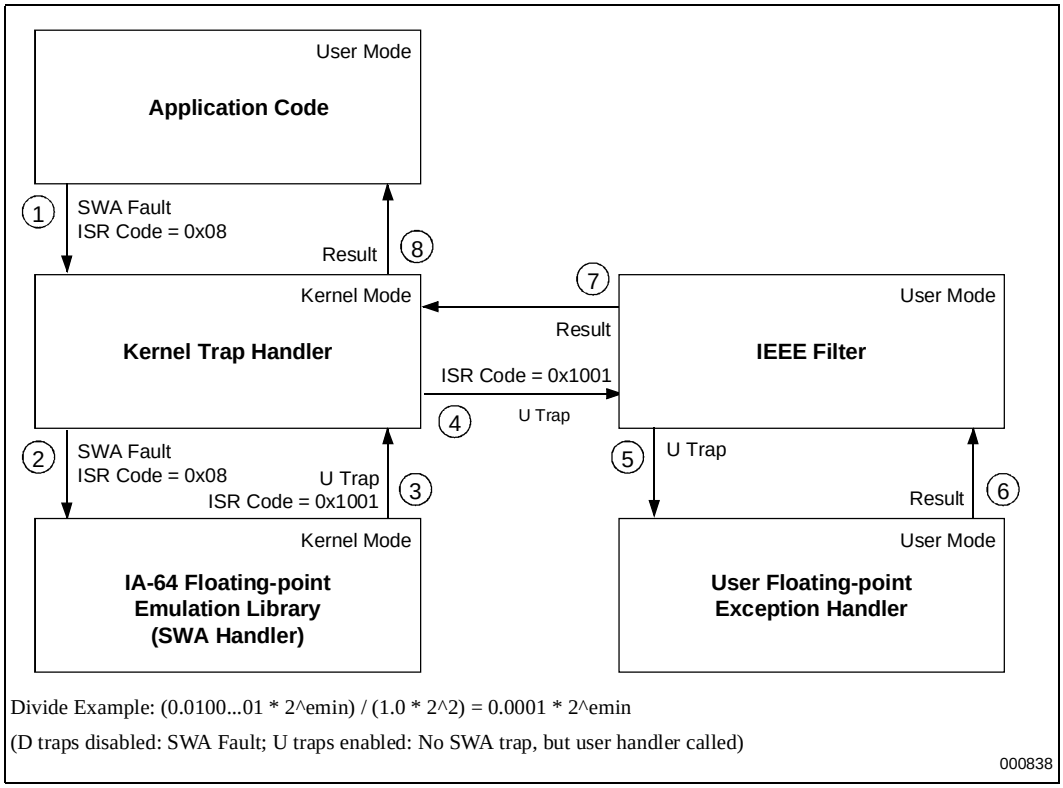
3.14 Examples

The control flow and the sequence of steps followed in processing floating-point exceptions are illustrated next by four examples. The order of the sequence of steps followed is marked in each figure. Software components that occur multiple times in the same figure indicate different invocations (only in Example 1 and Example 4).

Example 1

The first example is that of a divide operation that raises a SWA fault, and then an underflow trap (underflow traps are assumed to be enabled).

Figure 3-6. Flow of Control for Handling a SWA Fault Raised by a Divide Operation, Followed by an Underflow Trap

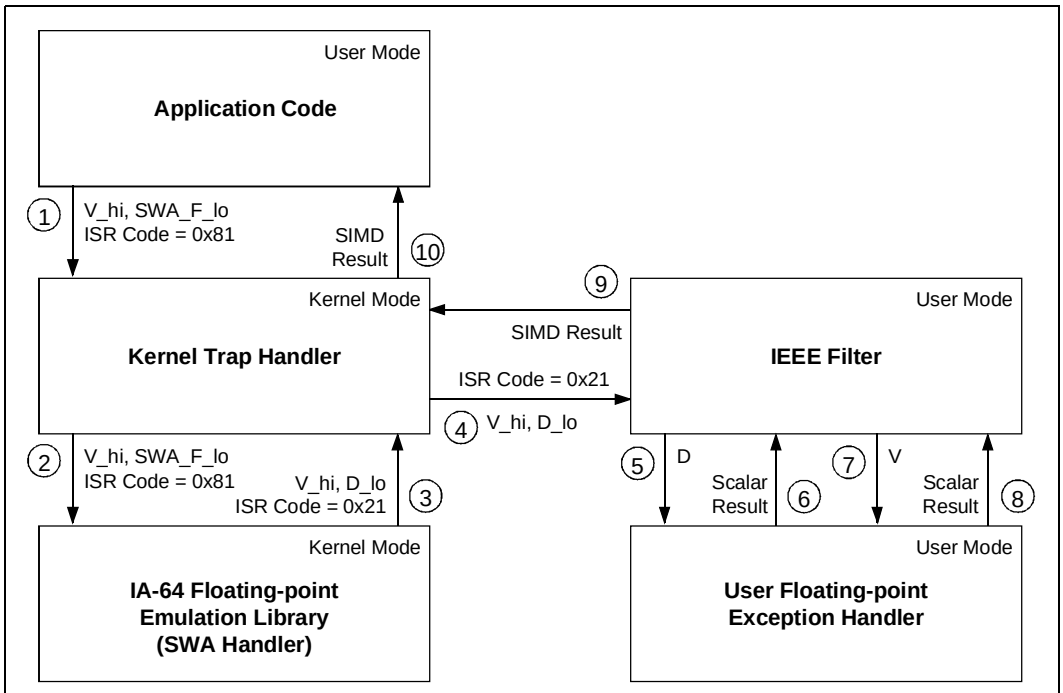


Assuming a scalar divide operation, the SWA fault is raised by an *frcpa* instruction that jump-starts the iterative computation for computing the result of the divide operation. As the result is provided by the user exception handler for unmasked (enabled) underflow exceptions, the output predicate of frpca has to be clear when execution of the application program containing it is resumed (clearing the output predicate is the task of the user handler or of the IEEE Floating-point Exception Filter). The clear output predicate disables the iterative computation following *frcpa*, as the result is already in the correct floating-point register (the iterative computation will be in general inlined automatically by the compiler).

Example 2

The second example illustrates the case of a parallel instruction that raises an invalid fault in the high half, and a SWA fault in the low half (reported first as a SWA fault, but then converted to a denormal fault by the IA-64 Floating-point Emulation Library). Both invalid and denormal exceptions are assumed to be enabled. This example could be that of an *fpma* instruction having a denormal operand in the low half (e.g. $0.1 * 2^{-126}$) and a signaling NaN in the high half (the *fpma* operation has three single precision input operands for each half). It is assumed that processing of the two halves at the user level takes place in the little endian order: low half first, followed then by the high half (this is just a convention, and is not imposed by the architecture).

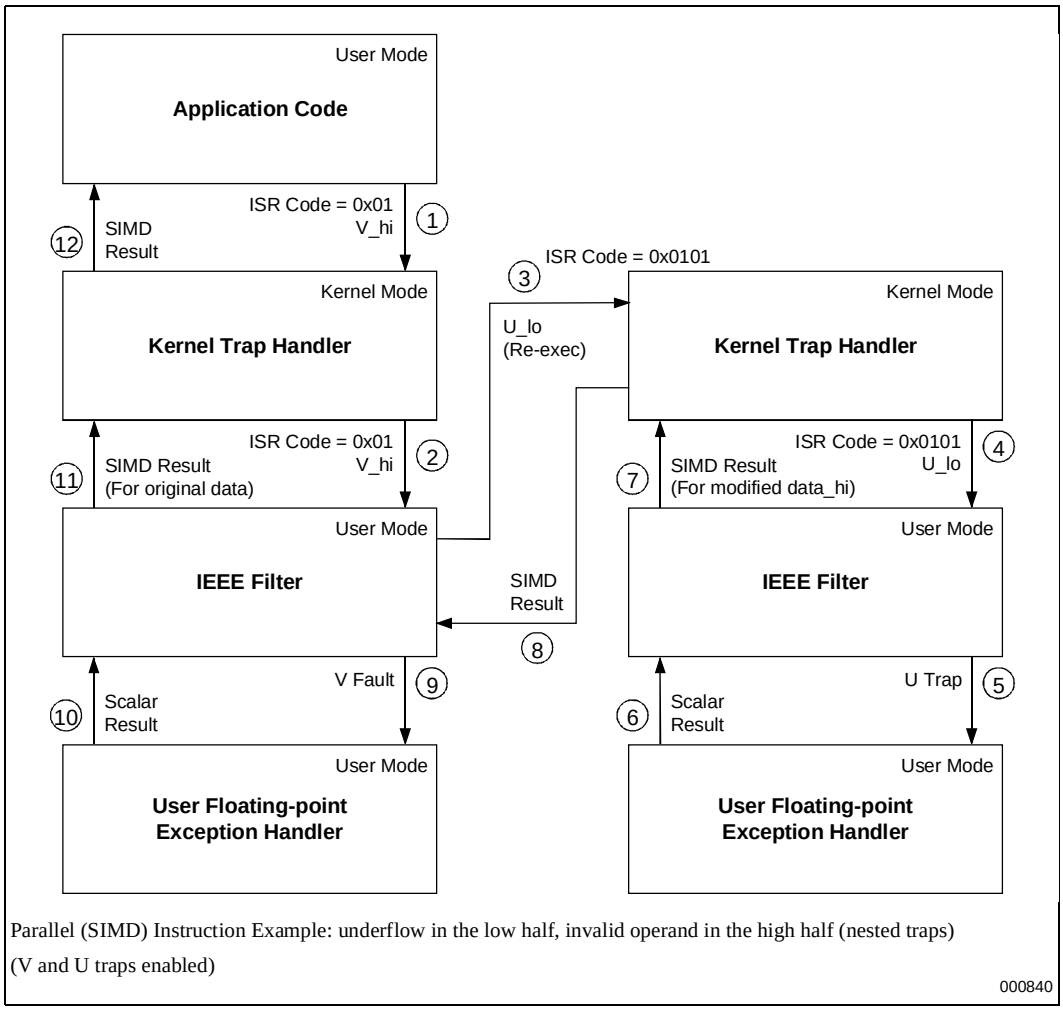
Figure 3-7. Flow of Control for Handling a Double Fault (V high, SWA Fault low), Raised by an IA-64 Parallel Instruction



Example 3

The third example illustrates the case of a parallel instruction that raises an invalid fault in the high half, and an underflow trap in the low half, with no SWA requests involved. Both invalid and underflow exceptions are assumed to be enabled. As only the fault is detected first, the IEEE filter tries to re-execute the low half of the instruction, generating a new exception (underflow trap). Note that steps 2 and 4 (the calls to the IEEE Filter from the kernel trap handler) should also include a brief call to the Floating-point Emulation Library. This is not represented in Figure 3-8, as the only action performed by the floating-point emulation library in these cases is to return to the caller.

Figure 3-8. Flow of Control for Handling a Fault in the High Half (V high), and a Trap in the Low Half (U low) of an IA-64 Parallel Instruction



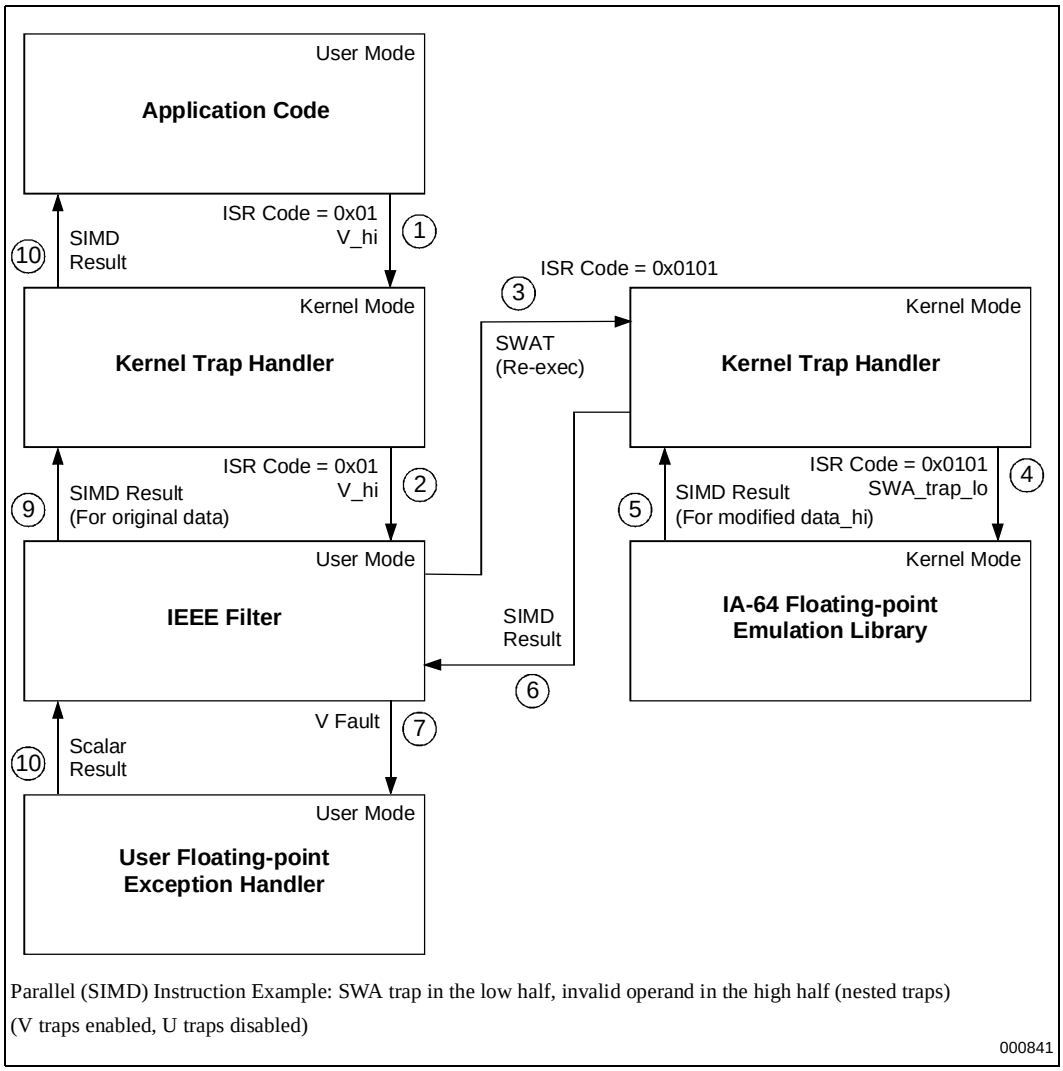
Parallel (SIMD) Instruction Example: underflow in the low half, invalid operand in the high half (nested traps)
(V and U traps enabled)

000840

Example 4

Finally, the fourth example repeats the third one, but with underflow exceptions disabled. This transforms the underflow trap into a SWA trap.

Figure 3-9. Flow of Control for Handling a Fault in the High Half (V high), and a SWA Trap in the Low Half of an IA-64 Parallel Instruction



Parallel (SIMD) Instruction Example: SWA trap in the low half, invalid operand in the high half (nested traps)

(V traps enabled, U traps disabled)

000841

Architecturally Mandated Floating-point Software Assistance 4

The architecturally mandated software assistance is necessary for the scalar reciprocal approximation instructions, *frcpa* and *frsqrta*, that help implement in software the floating-point and integer divide and remainder, and respectively the floating-point square root operations.

4.1 Conditions that Require Architecturally Mandated SWA

The architecturally mandated SWA conditions for the scalar divide and square root are presented next (the conditions for divide apply to the remainder operation as well, as it is based on the divide algorithm). Similar conditions exist for the parallel divide and square root operations, but they do not lead to SWA requests. The parallel instructions behavior will be detailed in Chapter 5.

4.1.1 Architecturally Mandated SWA Conditions for Divide

For divide, if *a/b* has to be calculated, the *frcpa* instruction provides an initial approximation of 1/*b* that allows starting a Newton-Raphson (or similar) iterative process to compute the correctly rounded value of *a/b* as specified by the IEEE-754 Standard for Binary Floating-point Computations [3]. Several floating-point divide algorithms are available, depending on the instruction type (parallel or not), on the precision of the arguments and of the result, but also on whether best latency or best throughput is preferred. For the scalar single, double, and double-extended precision algorithms (see [1] for the single and double precision algorithms) the result can be calculated correctly for any valid input values of the arguments. Special cases exist only for the algorithm that operates on register file format floating-point numbers. If the inputs are in floating-point register file format, with 17-bit exponents, then some of the intermediate computation steps might underflow, overflow, or lose precision. A sample algorithm for register file format computations is shown below. The same algorithm can be used for double-extended precision calculations.

Consider the following sample algorithm for calculating a/b in floating-point register file format, where a, b, y_0, e_0, y_1, e_1, y_2, e_2, y_3, e_3, y_4, q_0, r_0, q_1, r_1, and q_2 are floating-point numbers with N = 64 bits in the significand, y_0 is an 11-bit approximation of 1/b, *rn* is the IEEE rounding to nearest mode, and *rnd* is any IEEE rounding mode. The precision of the calculation is indicated for each step.

1. $y_0 = 1/b \cdot (1+\varepsilon_0), \lvert\varepsilon_0\rvert \le 2^{-m}, \quad m = 8.886$	table lookup
2. $e_0 = (1 - b \cdot y_0)_{rn}$	register file double-extended precision
3. $y_1 = (y_0 + e_0 \cdot y_0)_{rn}$	register file double-extended precision
4. $e_1 = (e_0^2)_{rn}$	register file double-extended precision
5. $y_2 = (y_1 + e_1 \cdot y_1)_{rn}$	register file double-extended precision
6. $e_2 = (1 - b \cdot y_2)_{rn}$	register file double-extended precision
7. $y_3 = (y_2 + e_2 \cdot y_2)_{rn}$	register file double-extended precision

8. $e_3 = (1 - b \cdot y_3)_{rn}$ register file double-extended precision
9. $y_4 = (y_3 + e_3 \cdot y_3)_{rn}$ register file double-extended precision
10. $q_0 = (a \cdot y_0)_{rn}$ register file double-extended precision
11. $y_o = (a - b \cdot q_0)_{rn}$ register file double-extended precision
12. $q_1 = (q_0 + r_0 \cdot y_3)_{rn}$ register file double-extended precision
13. $r_1 = (a - b \cdot q_1)_{rn}$ register file double-extended precision
14. $q_2 = (q_1 + r_1 \cdot y_4)_{rnd}$ register file double-extended precision

The algorithm generates $q_2 = (a/b)_{rnd}$, the correctly rounded floating-point register file format value of *a*/*b*. In the actual implementation, each of the 14 computation steps above translates into one IA-64 assembly language instruction. The first and the last step use status field 0 from the FPSR (the user status field), while all the intermediate steps use status field 1 (reserved for special computations by software conventions; status field 1 uses rounding to nearest, and the register file double-extended floating-point format [1], with 17-bit exponents and 64-bit significands). Steps (2) through the last are predicated by the output predicate of the *frcpa* instruction (corresponding to step (1) above). Thus, when the result of the divide operation is provided directly by the hardware or by the SWA handler (the IA-64 Floating-point Emulation Library), the output predicate will be cleared and steps (2) through the last will be skipped. For this to work, the *frcpa* instruction and the last instruction in the sequence need to have the same output register.

The conditions that might cause certain intermediate steps to overflow, underflow, or lose precision are the following, and they identify situations when the Itanium processor will have to ask for software assistance (SWA):

$$\begin{cases} (a) & e_b \le e_{min} - 1 \quad (y_i \text{ might be huge}) \\ (b) & e_b \ge e_{max} - 2 \quad (y_i \text{ might be tiny}) \\ (c) & e_a - e_b \ge e_{max} \quad (q_i \text{ might be huge}) \\ (d) & e_a - e_b \le e_{min} + 1 \quad (q_i \text{ might be tiny}) \\ (e) & e_a \le e_{min} + N - 1 \quad (r_i \text{ might lose precision}) \end{cases}$$

An observation is necessary for condition (a). The condition for avoiding generation of a huge y_i was determined mathematically as

$$\{(a) \quad e_b \le e_{min} - 2 \quad (y_i \text{ might be huge})$$

As written above, it would not require Software Assistance for some denormal values of *b* (for those that have $e_b = e_{min} - 1$, when combined with a value of *a* that has a negative exponent, but larger than $e_{min} + N - 1$; see Figure 4-1 in Section 4.2). If *frcpa* returns a valid approximation for the reciprocal 1/*b*, the divide algorithm will generate a correct result. The disadvantage is that steps (2), (6), (8), (11), and (13) of the algorithm above will all cause Itanium processor specific SWA faults, because *b* is denormal. It is therefore better to include the case $e_b = e_{min} - 1$ with the architecturally mandated SWA faults, thus allowing for the result to be generated with one single SWA fault. Condition (a) was thus modified to:

$$\{(a) \quad e_b \le e_{min} - 1 \quad (y_i \text{ might be huge})$$

An observation is necessary also for condition (b). When $e_b = e_{max} - 2$, some y_i might become tiny, but the precision loss will not be catastrophic, i.e. the final result of the divide operation will still be correct. Yet, producing a denormal y_i and then consuming it in the next instruction would cause an Itanium processor specific SWA trap, and then an Itanium processor specific SWA fault. This could

happen again for a subsequent y_i, which would slow down the computation significantly. The decision was taken to ask for software assistance for $e_b = e_{max} - 2$ too, not only for $e_b \geq e_{max} - 1$ (condition (b) is already modified above).

When an IA-64 architecturally mandated SWA fault is raised, the SWA handler will scale the input values appropriately, will calculate the result of the divide operation for the scaled values, and will scale back the result. The output predicate of *frcpa* will be set to 0.

4.1.2 Architecturally Mandated SWA Conditions for Square Root

For the square root, if $\sqrt{a}$ has to be calculated, the *frsqrta* instruction provides an initial approximation of $1/(\sqrt{a})$, that allows starting a Newton-Raphson or similar iterative process to compute the correctly rounded value of $\sqrt{a}$ as specified by the IEEE-754 Standard for Binary Floating-point Computations [3]. Several floating-point square root algorithms are available, depending on the instruction type (parallel or not), on the precision of the argument and of the result, but also on whether best latency or best throughput is preferred. For the scalar single, double, and double-extended precision algorithms devised (see [1] for the single and double precision algorithms) the result can be calculated correctly for any valid input values of the argument. Special cases exist only for the algorithm that accepts register file format floating-point numbers. If the inputs are in register file floating-point format (with 17-bit exponents), then some of the intermediate computation steps might lose precision. A sample algorithm for register file format computations is shown below. The same algorithm can be used for double-extended precision calculations.

Consider the following sample algorithm for calculating $\sqrt{a}$ in floating-point register file format, where a, h, t_1, t_2, t_3, t_4, t_5, t_6, y_1, y_2, S, H, d, S_1, H_1, d_1, and R are floating-point numbers with N = 64 bits in the significand, y_0 is an 11-bit approximation of $1/(\sqrt{a})$, *rn* is the IEEE rounding to nearest mode, and *rnd* is any IEEE rounding mode. The precision of the calculation is indicated for each step.

1. $y_0 = 1/\sqrt{a} \cdot (1 + \varepsilon_0),\ \lvert\varepsilon_0\rvert \leq 2^{-m},\ m = 8.831$	table lookup
2. $h = (1/2 \cdot a)_{rn}$	register file double-extended precision
3. $t_1 = (y_0 \cdot y_0)_{rn}$	register file double-extended precision
4. $t_2 = (1/2 - t_1 \cdot h)_{rn}$	register file double-extended precision
5. $y_1 = (y_0 + t_2 \cdot y_0)_{rn}$	register file double-extended precision
6. $t_3 = (y_1 \cdot h)_{rn}$	register file double-extended precision
7. $t_4 = (1/2 - t_3 \cdot y_1)_{rn}$	register file double-extended precision
8. $y_2 = (y_1 + t_4 \cdot y_1)_{rn}$	register file double-extended precision
9. $S = (a \cdot y_2)_{rn}$	register file double-extended precision
10. $t_5 = (y_2 \cdot h)_{rn}$	register file double-extended precision
11. $H = (1/2 \cdot y_2)_{rn}$	register file double-extended precision
12. $d = (a - S \cdot S)_{rn}$	register file double-extended precision
13. $t_6 = (1/2 - t_5 \cdot y_2)_{rn}$	register file double-extended precision
14. $S_1 = (S + d \cdot H)_{rn}$	register file double-extended precision
15. $H_1 = (H + t_6 \cdot H)_{rn}$	register file double-extended precision
16. $d_1 = (a - S_1 \cdot S_1)_{rn}$	register file double-extended precision
17. $R = (S_1 + d_1 \cdot H_1)_{rnd}$	register file double-extended precision

The algorithm generates $R = (\sqrt{a})_{rnd}$, the correctly rounded floating-point register file format value of $\sqrt{a}$. In the actual implementation, each of the 17 computation steps above translates into one IA-64 assembly language instruction. The first and the last step use status field 0 from the FPSR (the user status field), while all the intermediate steps use status field 1 (reserved for special computations by software conventions). Steps (2) through the last are predicated by the output predicate of the *frsqrta* instruction (corresponding to step (1) above). Thus, when the result of the square root operation is provided directly by the hardware or by the SWA handler (the IA-64 Floating-point Emulation Library), the output predicate will be cleared and steps (2) through the last will be skipped. For this to work, the *frsqrta* instruction and the last instruction in the sequence have to have the same output register.

The condition that might cause certain intermediate steps to lose precision is shown below. It identifies situations when the Itanium processor will have to ask for software assistance (SWA):

$$e_a \leq e_{min} + N - 1 \ (d_i \text{ might lose precision})$$

When an IA-64 architecturally mandated SWA fault is raised, the SWA handler will scale the input value appropriately, will calculate the result of the square root for the scaled value, and will scale back the result. The output predicate of *frsqrta* will be set to 0.

4.1.3 Floating-point Traps Raised by the SWA Handler for Architecturally Mandated SWA Faults

The IA-64 architecturally mandated SWA conditions were presented in Table 3-2 of Section 3.4.

If *frcpa* raises an IA-64 architecturally mandated SWA fault, the SWA handler (the IA-64 Floating-point Emulation Library) will provide the result for the divide operation (not just an approximation for the inverse of the denominator), and will clear the output predicate. The software assistance handler can also raise an underflow, overflow, or inexact exception pertaining to the result of the divide operation. If any input is unnormal and the denormal exceptions are enabled, the SWA handler will just convert the ISR code to that of a denormal exception and the operating system will search for a corresponding user exception handler (see Section 4.2, Algorithms for SWA Faults for Floating-point Divide, for more details)

Similarly, if *frsqrta* raises an IA-64 architecturally mandated SWA fault, the SWA handler (the IA-64 Floating-point Emulation Library) will provide the result for the square root operation (not just an approximation for the inverse of the square root of the denominator), and will clear the output predicate. The software assistance handler can also raise an inexact exception pertaining to the result of the square root operation. If the input is unnormal and the denormal exceptions are enabled, the SWA handler will just convert the ISR code to that of a denormal exception and the operating system will search for a corresponding user exception handler (see Section 4.4, Algorithms for SWA Faults for Floating-point Square Root for more details).

The existence of floating-point traps raised by the SWA handler (the IA-64 Floating-point Emulation Library) was acknowledged also in Section 3.3, ‘‘Floating-point Exception Priority’’, through Figure 3-4 and Figure 3-5. Figure 3-3 shows also that a denormal fault can be raised by the SWA handler following an Itanium processor specific SWA fault.

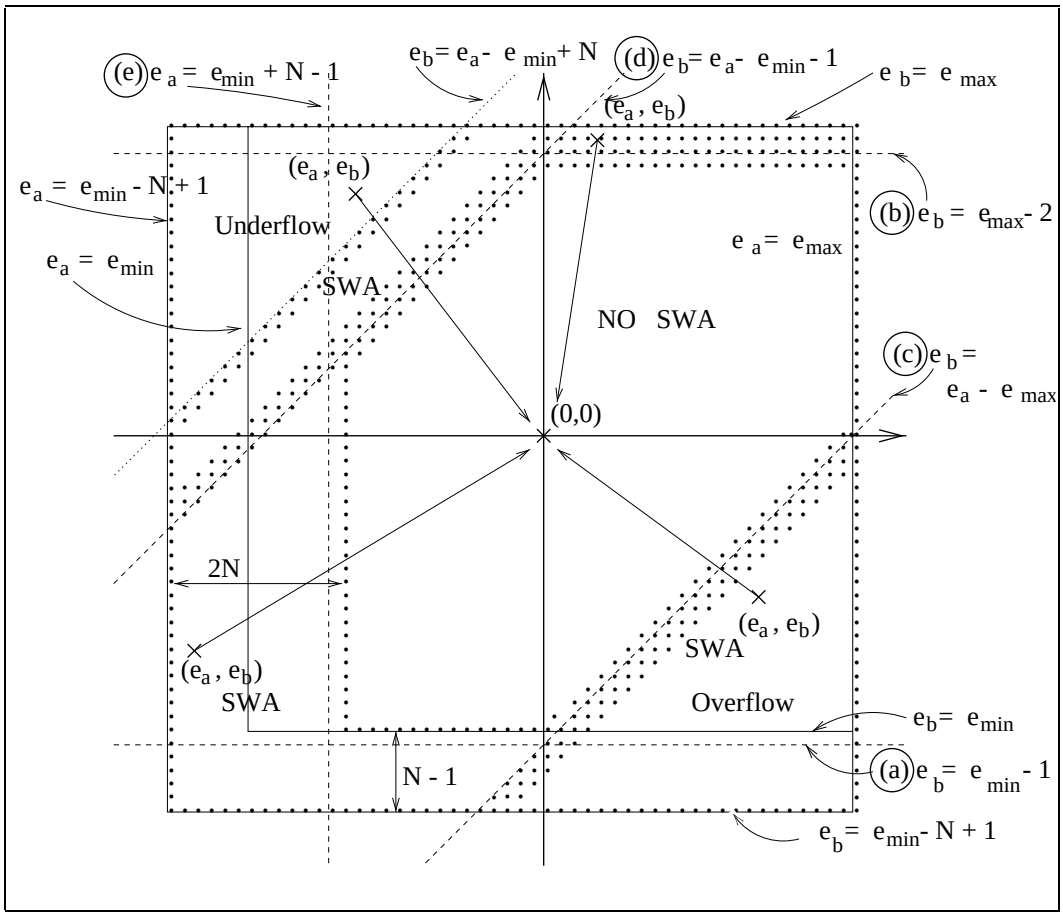
Of the floating-point exceptions that can be raised from software, some traps can never be raised directly by the hardware. These are overflow, underflow, and inexact traps raised following an architecturally mandated SWA fault for *frcpa*, and inexact traps raised following an architecturally mandated SWA fault for *frsqrta*. A user floating-point exception handler reached for such an enabled exception will associate it with *frcpa* or *frsqrta* respectively (so from the user’s point of view, these traps appear to be raised by the two scalar reciprocal approximation instructions).

4.2 Algorithms for SWA Faults for Floating-point Divide

When an architecturally mandated SWA fault is raised for *frcpa*, the SWA handler, which is invoked by the operating system kernel, uses alternate algorithms to calculate the result for the divide operation.

Figure 4-1 delimits the regions of the (e_a, e_b) plane where IA-64 architecturally mandated SWA faults are required. Note that e_a and e_b are integer numbers, which means that only points of integer coordinates in plane have to be considered. We have $e_a,\ e_b \in [e_{min} - N + 1,\ e_{max}]$, as denormal values are also allowed. The five conditions, (a) through (e), that determine the architecturally mandated SWA faults are represented by half-planes, delimited by straight lines as shown in Figure 4-1. The innermost irregular hexagon delimited by dotted lines (including its boundaries) identifies the points in plane for which IA-64 architecturally mandated software assistance is not necessary. The regions marked *Underflow* and *Overflow* contain points where the result of the divide operation is either tiny or huge. For some points on these regions' boundaries, the underflow or overflow might or might not occur, depending also on the significands of the dividend and of the divisor.

Figure 4-1. Architecturally Mandated SWA Conditions for frcpa



The various conditions that require architecturally mandated SWA faults for the divide operation will be examined next in detail. A floating-point number *x* will be represented as a product of its sign, significand, and a power of 2:

$$x = \sigma_x \cdot s_x \cdot 2^{e_x}$$

The four IEEE rounding modes are *rn* (rounding to nearest), *rm* (rounding to negative infinity), *rp* (rounding to positive infinity), and *rz* (rounding to zero). An unspecified rounding mode is denoted by *rnd*.

The thirteen cases that follow are listed in the order in which they are checked for in the source code of the FP SWA handler. Their sequence translates into an ‘‘if - else if - else if - ... - else’’ construct. This means that when examining any condition, it may be looked at as if logically AND-ed with the negations of all the previous conditions. Note though that in any of the following cases, if any input argument to *frcpa* is unnormal and the denormal exceptions are enabled, a denormal fault will be taken (which implies setting the D flag in the ISR code, and leaving the FPSR unchanged).

Case (I) $e_b \le e_a - e_{max} - 2$ (part of condition (c) for SWA)

It can be shown that:

$$\left|\frac{a}{b}\right| \ge MAXFP + 1ulp$$

holds, where *MAXFP* is the largest normal number that can be represented in the floating-point register file format.

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} \ge \frac{s_a}{s_b} \cdot 2^{e_{max}+2} \ge \frac{1}{2 - 2^{-N+1}} \cdot 2^{e_{max}+2} > 2^{e_{max}+1}$$
$$= MAXFP + 1ulp$$

The calculation of $|a/b|$ will always raise an overflow exception (enabled or not), no matter which rounding mode is used. The first computation steps are shown below (some details are omitted).

- Scale *a* and *b*:

$$a_1 = a \cdot 2^{-e_a},\ e_{a_1} = e_a - e_a = 0$$
$$b_1 = b \cdot 2^{-e_b},\ e_{b_1} = e_b - e_b = 0$$

- Calculate c_1 using the algorithm for double-extended and floating-point register file format values, and a local value of the Floating-point Status Register (FPSR), but with the user settings for rounding mode, precision and computation mode:

$$c_1 = \frac{a_1}{b_1} = c \cdot 2^{e_b - e_a}$$

This is a normal floating-point number:

$$|c_1| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b - e_a + e_b} = \frac{s_a}{s_b} \in \left(\frac{1}{2}, 2\right)$$

The result depends now on whether an overflow or inexact trap occurs or not. The three cases that matter are listed below.

(I a). If the overflow traps are disabled and the inexact traps are disabled, then return the IEEE mandated result for the divide operation:

$$\frac{a}{b} = \begin{cases} +\infty, \text{ if } \sigma_a \wedge \sigma_b = 0 \text{ and } rnd \text{ is } rn \text{ or } rp \\ +MAXFP, \text{ if } \sigma_a \wedge \sigma_b = 0 \text{ and } rnd \text{ is } rm \text{ or } rz \\ -\infty, \text{ if } \sigma_a \wedge \sigma_b = 1 \text{ and } rnd \text{ is } rn \text{ or } rm \\ -MAXFP, \text{ if } \sigma_a \wedge \sigma_b = 1 \text{ and } rnd \text{ is } rp \text{ or } rz \end{cases}$$

where the symbol ^ denotes the exclusive OR operation.

- Set to 1 the overflow and inexact status flags in the FPSR.
- Clear the result predicate of the *frcpa* instruction.
- Return TRUE to the OS kernel trap handler, indicating that the result of the divide operation is provided.

(I b). If the overflow traps are enabled, the result of the divide has to be calculated with exponent modulo 2^{17}, and has to be delivered to the OS kernel, which will in turn pass it to the user trap handler. The computation steps are listed below.

- Calculate the biased exponent for $c = c_1 \cdot 2^{e_a - e_b}$:

  $$e_c + bias = e_{c_1} + e_a - e_b + bias$$

  As $e_c + bias \geq 2^{17} - 1$, set $O = 1$ in the ISR code
- Set the result to $c^* = \sigma_c \cdot s_c \cdot 2^{(e_c + bias) mod 2^{17}}$, which will be delivered to the exception handler.
- If the result is inexact, determine the *fpa* bit. Using a local value of the FPSR, with rounding to nearest and in floating-point register file format, calculate

  $$d_1 = |b_1| \cdot |c_1| - |a_1|$$

  As c_1 is within 1 ulp of a_1/b_1 (no matter which rounding mode is used to calculate it), the calculation for d_1 will be exact. Also, d_1 will be normal due to the range of a_1. Two cases are possible, as the result is inexact (this excludes the case $d_1 = 0$):

  If $d_1 < 0$, set $I = 1$ and $fpa = 0$ in the ISR code.

  If $d_1 > 0$, set $I = 1$ and $fpa = 1$ in the ISR code.
- Set $O = 1$ and update I in the FPSR.
- Clear the output predicate of *frcpa*.

- Return FALSE to the OS kernel trap handler, indicating that a new (and different) exception has to be raised, and that it provides a result for the divide operation that has to be propagated to the user exception handler. Also indicate that a fault was converted to a floating-point trap (this will help the OS kernel to increment the instruction pointer correctly).

(I c). If the inexact traps are enabled and the result is inexact, then return the IEEE mandated result for the divdide operation:

$$\frac{a}{b} = \begin{cases} +\infty, \text{ if } \sigma_a \wedge \sigma_b = 0 \text{ and } rnd \text{ is } rn \text{ or } rp \\ +MAXFP, \text{ if } \sigma_a \wedge \sigma_b = 0 \text{ and } rnd \text{ is } rm \text{ or } rz \\ -\infty, \text{ if } \sigma_a \wedge \sigma_b = 1 \text{ and } rnd \text{ is } rn \text{ or } rm \\ -MAXFP, \text{ if } \sigma_a \wedge \sigma_b = 1 \text{ and } rnd \text{ is } rp \text{ or } rz \end{cases}$$

- Set $I = 1$ in the ISR code and $fpa = 1$ if the result is infinite (in absolute value).
- Set $O = 1$ and $I = 1$ in the FPSR.
- Clear the output predicate.
- Return FALSE to the OS kernel trap handler, indicating that a new (and different) exception has to be raised, and that it provides a result for the divide operation that has to be propagated to the user exception handler. Also indicate that a fault was converted to a floating-point trap (this will help the OS kernel to increment the instruction pointer correctly).

Case (II) $e_b = e_a - e_{max} - 1$ and $s_a \geq s_b$ (part of condition (c) for SWA)

It can be shown that:

$$\left|\frac{a}{b}\right| \geq MAXFP + 1ulp$$

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} = \frac{s_a}{s_b} \cdot 2^{e_{max}+1} \geq 2^{e_{max}+1} = MAXFP + 1ulp$$

This case is similar to Case (I) above, and it can be logically OR-ed with Case (I).

Case (III) $e_b = e_a - e_{max} - 1$ and $s_a < s_b$ (part of condition (c) for SWA)

It can be shown that

$$MAXFP \leq \left|\frac{a}{b}\right| \leq MINFP$$

holds, where *MINFP* is the smallest normal number that is representable in floating-point register file format. This means that *a*/*b* will be a normal floating-point number, no matter which rounding mode is used to calculate it.

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} = \frac{s_a}{s_b} \cdot 2^{e_{max}+1} \leq \frac{2 - 2^{-N+2}}{2 - 2^{-N+1}} \cdot 2^{e_{max}+1}$$

First, it will be shown that:

$$\left|\frac{a}{b}\right| < MAXFP \Leftrightarrow$$

$$\left|\frac{a}{b}\right| < (2 - 2^{-N+1}) \cdot 2^{e_{max}}$$

From above, it is sufficient to show that:

$$\frac{2 - 2^{-N+2}}{2 - 2^{-N+1}} \cdot 2^{e_{max}+1} < (2 - 2^{-N+1}) \cdot 2^{e_{max}} \Leftrightarrow$$

$$2 \cdot (2 - 2^{-N+2}) < (2 - 2^{-N+1})^2 \Leftrightarrow$$

$$4 - 2^{-N+3} < 4 - 2^{-N+3} + 2^{-2N+2}$$

which is true.

Next, it will be shown that:

$$\left|\frac{a}{b}\right| \geq MINFP$$

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} = \frac{s_a}{s_b} \cdot 2^{e_{max}+1} \geq \frac{1}{2 - 2^{-N+1}} \cdot 2^{e_{max}+1} > 2^{e_{max}} > MINFP$$

From above:

$$MINFP < \left|\frac{a}{b}\right| < MAXFP$$

which means that *a*/*b* is a normal floating-point number, but it still might raise an inexact exception, if the inexact exceptions are enabled. In any case, the result of the divide operation has to be calculated. The computation steps are listed below.

- Scale *a* and *b*:

$$a_1 = a \cdot 2^{-e_a},\ e_{a_1} = e_a - e_a = 0$$

$$b_1 = b \cdot 2^{-e_b},\ e_{b_1} = e_b - e_b = 0$$

Calculate c_1 using the algorithm for double-extended and floating-point register file format values, and a local value of the FPSR, but with the user settings for rounding mode, precision and computation mode:

$$c_1 = \frac{a_1}{b_1} = c \cdot 2^{-e_a + e_b}$$

This is a normal floating-point number:

$$|c_1| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b - e_a + e_b} = \frac{s_a}{s_b} \in \left(\frac{1}{2}, 2\right)$$

- Calculate $c = c_1 \cdot 2^{e_a - e_b}$.
  The result depends now on whether the inexact (I) traps are enabled or not.

(III a). If the inexact traps are disabled or the result is exact, then:

- Set the I flag in the FPSR if the result of the divide operation is inexact.
- Clear the result predicate of the *frcpa* instruction.
- Return TRUE to the OS kernel trap handler, indicating that it provides the result of the divide operation.

(III b). If the inexact traps are enabled and the result of the divide operation is inexact, the quotient has to be delivered to the OS kernel trap handler, which will raise an inexact exception and will then pass the quotient to the user trap handler.

The following computation steps have to be performed:

- Determine the *fpa* bit. Using a local value of the FPSR, with rounding to nearest and in floating-point register file format, calculate

  $$d_1 = |b_1| \cdot |c_1| - |a_1|$$

  As c_1 is within 1 ulp of a_1/b_1 (no matter which rounding mode is used to calculate it), the computation of d_1 will be exact. Also, d_1 will be normal, due to the range of a_1.
- Two cases are possible (as the result is inexact):
  If $d_1 < 0$, set $I = 1$ and $fpa = 0$ in the ISR code.
  If $d_1 > 0$, set $I = 1$ and $fpa = 1$ in the ISR code.
- Set $I = 1$ in the FPSR
- Clear the result predicate of the *frcpa* instruction.
- Return FALSE to the OS kernel trap handler, indicating that a new (and different) exception has to be raised, and that it provides a result for the divide operation that has to be propagated to the user exception handler. Also indicate that a fault was converted to a floating-point trap (this will help the OS kernel to increment the instruction pointer correctly).

Case (IV) $e_b = e_a - e_{max}$ (part of condition (c) for SWA)

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} = \frac{s_a}{s_b} \cdot 2^{e_{max}} \leq \frac{2 - 2^{-N+1}}{1} \cdot 2^{e_{max}} = MAXFP$$

Also:

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} = \frac{s_a}{s_b} \cdot 2^{e_{max}} \geq \frac{1}{2 - 2^{-N+1}} \cdot 2^{e_{max}} > 2^{e_{max}-1} > MINFP$$

From above:

$$MINFP < \left|\frac{a}{b}\right| \leq MAXFP$$

This case is thus similar to Case (III) above, and it can be logically OR-ed with Case (III).

Case (V) $e_a - e_{max} + 1 \leq e_b$ and $e_b \leq e_a - e_{min} - 2$ and ($e_a \leq e_{min} + N - 1$ or $e_b \leq e_{min} - 1$ or $e_b \geq e_{max} - 2$) (part of conditions (a), (b), and (e) for SWA)

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} \geq \frac{s_a}{s_b} \cdot 2^{e_{min} + 2} \geq \frac{1}{2 - 2^{-N+1}} \cdot 2^{e_{min} + 2} > 2^{e_{min} + 1}$$

$$> 2^{e_{min}} = MINFP$$

Also:

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} \leq \frac{s_a}{s_b} \cdot 2^{e_{max} - 1} \leq \frac{2 - 2^{-N+1}}{1} \cdot 2^{e_{max} - 1}$$

$$= \frac{1}{2} \cdot MAXFP < MAXFP$$

From above:

$$MINFP < \left|\frac{a}{b}\right| \leq MAXFP$$

This case too is similar to Case (III) above, and it can be logically OR-ed with Case (III).

Case (VI) $e_b = e_a - e_{min} - 1$ (part of conditions (b), (d) and (e) for SWA)

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} = \frac{s_a}{s_b} \cdot 2^{e_{min} + 1} \geq \frac{1}{2 - 2^{-N+1}} \cdot 2^{e_{min} + 1}$$

$$> \frac{1}{2} \cdot 2^{e_{min} + 1} = 2^{e_{min}} = MINFP$$

Also:

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} = \frac{s_a}{s_b} \cdot 2^{e_{min} + 1} < \frac{2 - 2^{-N+1}}{1} \cdot 2^{e_{min} + 1}$$

$$< (2 - 2^{-N+1}) \cdot 2^{e_{max}} = MAXFP$$

From above:

$$MINFP < \left|\frac{a}{b}\right| < MAXFP$$

This case too is similar to Case (III) above, and it can be logically OR-ed with Case (III).

Case (VII) $e_b = e_a - e_{min}$ and $s_a \geq s_b$ (part of conditions (b), (d) and (e) for SWA)

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} = \frac{s_a}{s_b} \cdot 2^{e_{min}} \geq 2^{e_{min}} = MINFP$$

Also:

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} = \frac{s_a}{s_b} \cdot 2^{e_{min}} \leq \frac{2 - 2^{-N+1}}{1} \cdot 2^{e_{min}}$$

$$< (2 - 2^{-N+1}) \cdot 2^{e_{max}} = MAXFP$$

From above:

$$MINFP \leq \left|\frac{a}{b}\right| < MAXFP$$

This case also is similar to Case (III) above, and it can be logically OR-ed with Case (III).

Case (VIII) $e_b = e_a - e_{min}$ and $s_a < s_b$ (part of conditions (b), (d) and (e) for SWA)

It can be shown that:

$$\left|\frac{a}{b}\right| \leq MINFP - 1ulp$$

where $MINFP - 1ulp$ is the smallest normal minus 1 ulp (1 ulp considered for the binade $[2^{emin-1}, 2^{emin})$.

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} = \frac{s_a}{s_b} \cdot 2^{e_{min}} \leq \frac{2 - 2^{-N+2}}{2 - 2^{-N+1}} \cdot 2^{e_{min}}$$

We will show that a/b is smaller than the smallest normal minus 1 ulp. For this, it is sufficient to show that:

$$\frac{2 - 2^{-N+2}}{2 - 2^{-N+1}} \cdot 2^{e_{min}} < (2 - 2^{-N+1}) \cdot 2^{e_{min}-1} \Leftrightarrow$$

$$2 \cdot (2 - 2^{-N+2}) < (2 - 2^{-N+1})^2 \Leftrightarrow$$

$$4 - 2^{-N+3} < 4 - 2^{-N+3} + 2^{-2N+2}$$

which is true.

It can also be shown that *a*/*b* is larger than the smallest denormal (which is not important anymore for the computation of the result of *a*/*b*.)

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} = \frac{s_a}{s_b} \cdot 2^{e_{min}} \geq \frac{1}{2 - 2^{-N+1}} \cdot 2^{e_{min}}$$

$$> 2^{e_{min}-1} > 2^{e_{min}-N+1}$$

where $2^{e_{min}-N+1}$ is the value of the smallest denormal that can be represented in the given format.

The calculation of the tiny value |*a*/*b*| might raise an underflow and/or an inexact exception, no matter which rounding mode is used. The main computation steps are listed below.

- Scale *a* and *b*:

$$a_1 = a \cdot 2^{-e_a},\ e_{a_1} = e_a - e_a = 0$$

$$b_1 = b \cdot 2^{-e_b},\ e_{b_1} = e_b - e_b = 0$$

- Calculate c_1 using the algorithm for double-extended and floating-point register file format values, and a local value of the FPSR, but with the user settings for rounding mode, precision and computation mode:

$$c_1 = \frac{a_1}{b_1} = c \cdot 2^{e_b - e_a}$$

  This is a normal floating-point number:

$$|c_1| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b + e_b - e_a} \in \left(\frac{1}{2}, 2\right)$$

- If the result is inexact, determine the *fpa* bit. Using a local value of the FPSR, with rounding to nearest and in floating-point register file format, calculate

$$d_1 = |b_1| \cdot |c_1| - |a_1|$$

  As c_1 is within 1 ulp of a_1/b_1 (no matter which rounding mode is used to calculate it), the calculation for d_1 will be exact. Also, d_1 will be normal due to the range of a_1. Two cases are possible (as the result is inexact):

  If $d_1 < 0$, then *fpa* = 0.

  If $d_1 > 0$, then *fpa* = 1.

The result depends now on whether an underflow or inexact trap occurs or not. Two cases that matter are the following:

(VIII a). If the underflow traps are enabled, the result of the divide has to be calculated with exponent modulo 2^{17}, and has to be delivered to the OS kernel, which will in turn pass it to the user trap handler. The computation steps are listed below.

- Calculate the biased exponent for $c = c_1 \cdot 2^{e_a - e_b}$:

$$e_c + bias = e_{c_1} + e_a - e_b + bias$$

- Set the result to $c^* = \sigma_c \cdot s_c \cdot 2^{(e_c + bias) mod 2^{17}}$, that will be delivered to the exception handler.
- Set $U = 1$ in the ISR code (as $e_c + bias \leq -2^{17} + 1$). If the result is inexact, set $I = 1$ and the value of *fpa* calculated above in the ISR code.
- Set $U = 1$ in FPSR. If the result is inexact, set $I = 1$ in the FPSR.
- Clear the output predicate of *frcpa*.
- Return FALSE to the OS kernel trap handler, indicating that a new (and different) exception has to be raised, and that it provides a result for the divide operation that has to be propagated to the user exception handler. Also indicate that a fault was converted to a floating-point trap (this will help the OS kernel to increment the instruction pointer correctly).

(VIII b). If the underflow traps are disabled:

- If the flush-to-zero mode is disabled (*ftz* = 0), calculate $c = c_1 \cdot 2^{e_a - e_b}$. As this is a ‘‘normal’’ with an exponent smaller than e_{min} when an unbounded exponent range is considered, *c* will have to be denormalized using the information in *fpa*, *rnd* = *I*, and *sticky* = 0 (where *I* is the inexact status flag from the calculation of c_1). (Denormalization consists in shifting the significand right and incrementing the exponent at each step until it reaches e_{min}. The result is then rounded to the destination precision.)
- Otherwise, if the flush-to-zero mode is enabled (*ftz* = 1), then |c| = 0, where c has the sign of c_1, and the result is inexact.
- If the result is inexact, set $U = 1$ and $I = 1$ in the FPSR.
- Clear the output predicate of *frcpa*.
- If the result is inexact and the inexact traps are enabled, set $I = 1$ and the value of the *fpa* bit in the ISR code (the value of *fpa* was calculated in the denormalization process). Return FALSE to the OS kernel trap handler, indicating that a new (and different) exception has to be raised, and that it provides a result for the divide operation that has to be propagated to the user exception handler. Also indicate that a fault was converted to a floating-point trap (this will help the OS kernel to increment the instruction pointer correctly).
- Otherwise, if the result *c* is exact or the inexact traps are disabled, return TRUE to the OS kernel, indicating that the SWA handler provides the result of the divide operation.

Case (IX) $e_a - e_{min} + 1 \leq e_b \leq e_a - e_{min} + N - 2$ (part of conditions (b), (d) and (e) for SWA)

It can be shown that:

$$\left|\frac{a}{b}\right| \leq MINFP - 1ulp$$

where *MINFP* – 1*ulp* is the smallest normal minus 1 ulp (1 ulp considered for the binade [2^{emin-1}, 2^{emin}).

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} \le \frac{s_a}{s_b} \cdot 2^{e_{min}-1} \le \frac{2 - 2^{-N+1}}{1} \cdot 2^{e_{min}-1}$$
$$= (2 - 2^{-N+1}) \cdot 2^{e_{min}-1} = MINFP - 1ulp$$

This shows that the infinitely precise *a*/*b* is tiny. Further, it can be shown that *a*/*b* is larger than the smallest denormal (which is not important anymore for the computation of the result of *a*/*b)*:

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} \ge \frac{s_a}{s_b} \cdot 2^{e_{min}-N+2} \ge \frac{1}{2 - 2^{-N+1}} \cdot 2^{e_{min}-N+2} > 2^{e_{min}-N+1}$$

where $2^{e_{min}-N+1}$ is equal in value to the smallest denormal that can be represented. This shows that no matter which rounding mode is used, *a*/*b* is a non-zero denormal, or possibly the smallest denormal (in absolute value).

Case (IX) is similar to Case (VIII) above, and it can be logically OR-ed with Case (VIII).

Case (X) $e_b = e_a - e_{min} + N - 1$ and $s_a \ge s_b$ (part of conditions (b), (d) and (e) for SWA)

In this case it can be shown that:

$$\left|\frac{a}{b}\right| < MINFP - 1ulp$$

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} = \frac{s_a}{s_b} \cdot 2^{e_{min}-N+1} \le \frac{2 - 2^{-N+1}}{1} \cdot 2^{e_{min}-N+1}$$
$$< 2^{e_{min}-N+2} < MINFP - 1ulp$$

This shows that *a*/*b* is tiny. Further, it can be shown that *a*/*b* is larger than the smallest denormal (which is not important anymore for the computation of the result of *a*/*b*):

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} = \frac{s_a}{s_b} \cdot 2^{e_{min}-N+1} \ge 2^{e_{min}-N+1}$$

where $2^{e_{min}-N+1}$ is equal in value to the smallest denormal that can be represented. This shows that no matter which rounding mode is used, the rounded value of *a*/*b* is a non-zero denormal.

Case (X) is similar to Case (VIII) above, and it can be logically OR-ed with Case (VIII).

Case (XI) $e_b = e_a - e_{min} + N - 1$ and $s_a < s_b$ (part of conditions (b), (d) and (e) for SWA)

It can be shown that *a*/*b* is smaller than the smallest denormal, minus 1 ulp (1 ulp considered for the binade $[2^{emin-N}, 2^{emin-N+1})$):

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} = \frac{s_a}{s_b} \cdot 2^{e_{min} - N + 1} < \frac{2 - 2^{-N+2}}{2 - 2^{-N+1}} \cdot 2^{e_{min} - N + 1}$$

In order to show that:

$$\left|\frac{a}{b}\right| < (2 - 2^{-N+1}) \cdot 2^{e_{min} - N}$$

where $(2 - 2^{-N+1}) \cdot 2^{e_{min} - N}$ is the smallest denormal minus 1 ulp, it is sufficient to show that:

$$\frac{2 - 2^{-N+2}}{2 - 2^{-N+1}} \cdot 2^{e_{min} - N + 1} < (2 - 2^{-N+1}) \cdot 2^{e_{min} - N} \Leftrightarrow$$

$$2 \cdot (2 - 2^{-N+2}) < (2 - 2^{-N+1})^2 \Leftrightarrow$$

$$4 - 2^{-N+3} < 4 - 2^{-N+3} + 2^{-2N+2}$$

which is true.

This shows that *a*/*b* is smaller than the smallest denormal number that can be represented in the floating-point register file format, even after rounding to the destination precision.

Case (XI) is similar from a computational point of view to Case (VIII) above, and it can be logically OR-ed with Case (VIII). In case (XI) though, the result will always be zero or the smallest normal in absolute value, provided no underflow trap is taken (it may still be zero if an inexact trap is taken).

Case (XII) $e_b \geq e_a - e_{min} + N$ (part of conditions (b), (d) and (e) for SWA)

It can be shown in this case too that *a*/*b* is smaller than or equal to the smallest denormal, minus 1 ulp (1 ulp considered for the binade $[2^{emin-N}, 2^{emin-N+1})$):

$$\left|\frac{a}{b}\right| = \frac{s_a}{s_b} \cdot 2^{e_a - e_b} \leq \frac{s_a}{s_b} \cdot 2^{e_{min} - N} \leq \frac{2 - 2^{-N+1}}{1} \cdot 2^{e_{min} - N}$$

$$= (2 - 2^{-N+1}) \cdot 2^{e_{min} - N}$$

where $(2 - 2^{-N+1}) \cdot 2^{e_{min} - N}$ is the smallest denormal minus 1 ulp.

Case (XII) too is similar from a computational point of view to Case (VIII) above, and it can be logically OR-ed with Case (VIII). In case (XII), just as in Case (XI), the result will always be zero or the smallest denormal in absolute value, provided no underflow trap is taken (it may still be zero if an inexact trap is taken).

Note that for coding purposes, cases (IX), (X), (XI), and (XII) can be logically OR-ed to form one condition, $e_b \geq e_a - e_{min} + 1$.

Case (XIII) Error

If all the tests for cases (I) through (XII) have failed, an architecturally mandated SWA fault or an Itanium processor specific SWA fault caused by a floating-point register file format denormal is not necessary. In the actual IA-64 Floating-point Emulation Library code, this case is reserved for Itanium processor specific SWA faults for floating-point data types with an exponent range smaller than that for 17 bits (8, 11, or 15 bits). In such cases, an approximation of the value of $1/b$ and an output predicate set to 1 are the results of the *frcpa* instruction. The SWA handler returns TRUE to the OS kernel, indicating that it provides the result to *frcpa*. The software algorithm that begins with *frcpa* will then calculate the result for the divide operation.

Note that the logical union of Cases (I) through (V) specified above covers condition (a) for software assistance from Section 4.1.1:

$$\{(a) \quad e_b \leq e_{min} - 1 \quad (y_i \text{ might be huge})$$

Cases (V) through (XII) cover condition (b):

$$\{(b) \quad e_b \geq e_{max} - 2 \quad (y_i \text{ might be tiny})$$

Cases (I) through (IV) cover condition (c):

$$\{(c) \quad e_a - e_b \geq e_{max} \quad (q_i \text{ might be huge})$$

Cases (VI) through (XII) cover condition (d):

$$\{(d) \quad e_a - e_b \leq e_{min} + 1 \quad (q_i \text{ might be tiny})$$

Finally, Cases (V) through (XII) cover condition (e):

$$\{(e) \quad e_a \leq e_{min} + N - 1 \quad (r_i \text{ might lose precision})$$

4.3 Frequency Estimation of the Architecturally Mandated SWA Faults for Floating-point Divide

An estimation of the total number of possible pairs of exponents for a and b, (e_a, e_b), is:

$$N_{tot} = [e_{max} - (e_{min} - 1) + 1]^2 = (e_{max} - e_{min} + 2)^2 = (2 \cdot e_{max} + 1)^2$$

where N is the number of bits in the significand. We counted e_{min} twice, in order to account for denormals (whose significands are of the form $0.b_1b_2...b_{N-1}$, with b_i binary digits). For the purposes of this estimation, unnormals other than denormals are considered to be already normalized.

The total number of possible pairs of exponents for *a* and *b*, (e_a, e_b), for which the architecturally mandated SWA does not happen, is:

$$N_{NOSWA} = [(e_{max} - 3) - e_{min} + 1] \cdot [e_{max} - (e_{min} + N) + 1]$$
$$-\frac{1}{2} \cdot (-e_{min} + 1) \cdot (-e_{min} + 2) - \frac{1}{2} \cdot (-e_{min} - N) \cdot (-e_{min} - N + 1)$$
$$= (2 \cdot e_{max} - 3) \cdot (2 \cdot e_{max} - N)$$
$$-\frac{1}{2} \cdot e_{max} \cdot (e_{max} + 1) - \frac{1}{2} \cdot (e_{max} - N - 1) \cdot (e_{max} - N)$$

(we subtracted the lower and the upper triangles respectively from the rectangle considered initially in Figure 4-1 of Section 4.2). The total number of possible pairs of exponents for *a* and *b*, (e_a, e_b), for which the architecturally mandated SWA does happen is then:

$$N_{SWA} = N_{tot} - N_{NOSWA}$$

For the floating-point register file format:

$$N_{tot} = (2 \cdot 65535 + 1)^2 = 17, 179, 607, 041$$
$$N_{NOSWA} = (2 \cdot 65535 - 3) \cdot (2 \cdot 65535 - 64) -$$
$$\frac{1}{2} \cdot 65535 \cdot 65536 - \frac{1}{2} \cdot (65535 - 64 - 1) \cdot (65535 - 64) =$$
$$131067 \cdot 131006 - 2, 147, 450, 880 - 2, 143, 193, 185 = 12, 879, 919, 337$$
$$N_{SWA} = 17, 179, 607, 041 - 12, 879, 919, 337 = 4, 299, 687, 704$$

From here:

$$N_{SWA}/N_{tot} = (4, 299, 687, 704/17, 179, 607, 041) \cdot 100 = 25.0278\ \%$$

This calculation assumes that all the possible exponents of *a* and *b* have equal probability of occurring when computing *a*/*b*.

Note that in the computation performed in the SWA handler for the floating-point register file format, the conditions for Itanium processor specific SWA faults are covered by the conditions for the architecturally mandated SWA faults (this was achieved also by modifying condition (a), as explained above).

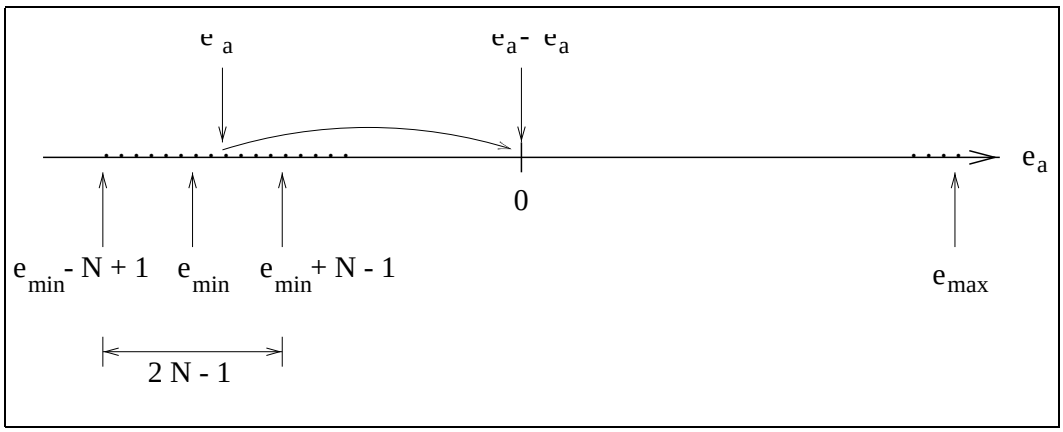
4.4 Algorithms for SWA Faults for Floating-point Square Root

When an architecturally mandated SWA fault is raised for *frsqrta*, the SWA handler, which is invoked by the operating system kernel, uses an alternate algorithm to calculate the result for the square root operation.

Figure 4-2 shows the regions of the e_a axis where IA-64 architecturally mandated SWA faults are required. Note that e_a is an integer number, which means that only points of integer coordinates on this axis have to be considered. We have $e_a \in [e_{min} - N + 1, e_{max}]$, as denormal values are also allowed. The condition that determines the necessity for architecturally mandated SWA faults is

$$e_a \leq e_{min} + N - 1$$

Figure 4-2. Architecturally Mandated SWA Condition for frsqrta



This condition will be examined next. Again, a floating-point number *x* will be represented as a product of its sign, significand, and a power of 2:

$$x = \sigma_x \cdot s_x \cdot 2^{e_x}$$

The four IEEE rounding modes are *rn* (rounding to nearest), *rm* (rounding to negative infinity), *rp* (rounding to positive infinity), and *rz* (rounding to zero). An unspecified rounding mode is denoted by *rnd*.

The two cases that follow are listed in the order in which they are checked for in the IA-64 Floating-point Emulation Library source code. Their sequence translates into an ‘‘if - else’’ construct. Note though that in any of the following two cases, if the input argument to *frsqrta* is unnormal and the denormal exceptions are enabled, a denormal fault will be taken (which implies setting the D flag in the ISR code, and leaving the FPSR unchanged).

Case (I) $e_a \leq e_{min} + N - 1$

The square root reduces approximately in half the exponent of the result, which means that $\sqrt{a}$ will be a normal floating-point number in register file format.

To calculate $s = \sqrt{a}$, the value of *a* is scaled to

$$a_1 = a \cdot 2^{-e_a} \quad if\ e_a\ is\ even,\ or$$

$$a_1 = a \cdot 2^{-e_a + 1} \quad if\ e_a\ is\ odd$$

After applying the square root algorithm, $s_1 = \sqrt{a_1}$ is obtained, which has to be scaled back:

$$s = s_1 \cdot 2^{e_a/2} \quad if\ e_a\ is\ even,\ or$$

$$s = s_1 \cdot 2^{(e_a - 1)/2} \quad if\ e_a\ is\ odd$$

The subsequent actions depend now on whether the result is inexact, and whether the inexact (I) traps are enabled.

(I a). If the inexact traps are disabled or the result is exact:

- Set the result for the square root operation to s
- If the result is inexact, set *I* = 1 (the inexact status flag) in the FPSR
- Clear the output predicate of *frsqrta*
- Return TRUE to the OS kernel, indicating that the SWA handler provides the result of the square root operation

(I b). If the inexact traps are enabled and the result of the square root operation is inexact, then the result has to be delivered to the OS kernel, which will in turn pass it to the user trap handler:

- Determine the *fpa* bit. Using a local value of the FPSR, with rounding to nearest and in floating-point register file format, calculate

$$d_1 = |s_1| \cdot |s_1| - |a_1|$$

- As s_1 is within 1 ulp of a_1/s_1 (no matter which rounding mode was used to calculate it), the computation of d_1 will be exact. Also, d_1 will be normal, due to the range of a_1.
- Two cases are possible:
  If $d_1 < 0$, set *I* = 1 and *fpa* = 0 in the ISR code
  If $d_1 > 0$, set *I* = 1 and *fpa* = 1 in the ISR code
- Set *I* = 1 (the inexact status flag) in the FPSR
- Set the result for the square root operation to s
- Clear the output predicate of *frsqrta*
- The SWA handler returns FALSE to the OS kernel, indicating that a new (and different) exception has to be raised, and that it provides a result for the square root operation that has to be propagated to the user exception handler. Also indicate that a fault was converted to a floating-point trap (this will help the OS kernel to increment the instruction pointer correctly).

Case (II) Error

If the test for case (I) has failed, it means that an architecturally mandated SWA fault or an Itanium processor specific SWA fault caused by a floating-point register file format unnormal is not

necessary. In the actual code IA-64 Floating-point Emulation Library, this case is reserved for Itanium processor specific SWA faults for floating-point data types with an exponent range smaller than that for 17 bits (8, 11, or 15 bits). In such cases, an approximation of the value $1/\sqrt{a}$ and an output predicate set to 1 are the results of the *frsqrta* instruction. The SWA handler returns TRUE to the OS kernel, indicating that it provides the result to *frsqrta*. The software algorithm that begins with *frsqrta* will then calculate the result for the square root operation.

4.5 Frequency Estimation of the Architecturally Mandated SWA Faults for Floating-point Square Root

An estimation of the total number of possible values of the exponent of *a* is:

$$N_{tot} = e_{max} - (e_{min} - 1) + 1 = 2 \cdot e_{max} + 1$$

where *N* is the number of bits in the significand, and the value of $e_{min} - 1$ was added to account for denormal values of *a*. For the purposes of this estimation, unnormals other than denormals are considered to be already normalized.

The number of values of the exponents e_a for which the architecturally mandated SWA faults occur, is:

$$N_{SWA} = (e_{min} + N - 1) - (e_{min} - 1) + 1 = N + 1$$

For floating-point register file format:

$$N_{SWA}/N_{tot} = (64 + 1)/(2 \cdot 65535 + 1) \cdot 100 = 0.04959$$

This calculation assumes that all the possible exponents of *a* have equal probability of occurring when computing $\sqrt{a}$.

Architecturally Mandated Pseudo-SWA Requests for Parallel Computations 5

The architecturally mandated software assistance requests are issued while executing the *frcpa* or *frsqrta* instructions, when an intermediate computation step in the calculation of a divide or square root result initiated by these instructions might overflow, underflow, or lose precision, thus potentially leading to an incorrect result, or to the incorrect raising of an IEEE exception. A SWA request is a floating-point exception handled by the IA-64 Floating-point Emulation Library.

Similar situations may arise for the *fprcpa* or *fprsqrta* instructions. In these cases, rather than issuing SWA requests, *fprcpa* and *fprsqrta* are raising ‘‘pseudo-SWA requests’’, by merely clearing their output predicate. The software assistance will have to come in this case from the user code, instead of a dedicated SWA handler (the IA-64 Floating-point Emulation Library). As the divide and square root sequences of instructions are usually inlined by compilers, code for processing pseudo-SWA requests will have to be inlined as well. The complimentary code will unpack the parallel (SIMD) operands of the instruction that issued the pseudo-SWA request, will normalize them, will perform two scalar calculations for the two halves of the result, and will pack the parallel result. This is possible, because performing the unpacked operations using the floating-point register file format for single precision operands will avoid any further SWA requests (this could not have been easily possible for SWA requests issued for register file format operands, as there is no higher precision to use for a simple alternate calculation).

There is a major advantage to this approach, as the pseudo-SWA requests from *fprcpa* and *fprsqrta* occur more frequently (on a relative scale) than those from *frcpa* and *frsqrta*. Therefore it is important to ensure faster processing of these requests in the application code, than what can be achieved using an interrupt handler.

The main disadvantage is that a clear output predicate for *fprcpa* and *fprsqrta* does not disambiguate the cases when the output register contains the result of the divide or square root operation (e.g. for 0/0 or $\sqrt{-1}$), from the cases when it contains just a reciprocal approximation (examples will be given below for each instruction). This constitutes an issue when floating-point exceptions that are enabled occur while computing divide or square root results (the detailed solution is not included here).

5.1 Architecturally Mandated Pseudo-SWA Conditions for Parallel Floating-point Divide

For the parallel divide, if a_1/b_1 and a_2/b_2 have to be calculated, the *fprcpa* instruction provides initial approximations of $1/b_1$ and $1/b_2$ that allow starting a Newton-Raphson or similar iterative process to compute the correctly rounded values of a_1/b_1 and a_2/b_2 as specified by the IEEE-754 Standard for Binary Floating-point Computations [3]. The sample algorithm presented below might not generate the IEEE correct result if some of the intermediate computation steps underflow, overflow, or lose precision. For simplicity, the computation is presented only for one set of single precision input values.

Consider the following algorithm for calculating a/b in single precision, where a, b, y_0, e_0, y_1, e_1, y_2, e_2, y_3, q_0, r_0, q_1, r_1, and q_2 are floating-point numbers with N_s-bit significands, y_0 is an 11-bit

approximation of 1/*b*, *rn* is the IEEE rounding to nearest mode, and *rnd* is any IEEE rounding mode. The precision of the calculation is indicated for each step.

1. $y_0 = 1/b \cdot (1 + \varepsilon_0),\ |\varepsilon_0| \le 2^{-m},\ m = 8.886$ — table lookup
2. $e_0 = (1 - b \cdot y_0)_{rn}$ — single precision
3. $y_1 = (y_0 + e_0 \cdot y_0)_{rn}$ — single precision
4. $e_1 = (1 - b \cdot y_1)_{rn}$ — single precision
5. $y_2 = (y_1 + e_1 \cdot y_1)_{rn}$ — single precision
6. $e_2 = (1 - b \cdot y_2)_{rn}$ — single precision
7. $y_3 = (y_2 + e_2 \cdot y_2)_{rn}$ — single precision
8. $q_0 = (a \cdot y_0)_{rn}$ — single precision
9. $r_0 = (a - b \cdot q_0)_{rn}$ — single precision
10. $q_1 = (q_0 + r_0 \cdot y_2)_{rn}$ — single precision
11. $r_1 = (a - b \cdot q_1)_{rn}$ — single precision
12. $q_2 = (q_1 + r_1 \cdot y_3)_{rn}$ — single precision

The algorithm generates $q_2 = (a/b)_{rnd}$, the correctly rounded single precision value of a/b.

The conditions that might cause certain intermediate steps to overflow, underflow, or lose precision are the following, and they identify situations when the Itanium processor will have to issue a pseudo-software assistance (SWA) request, by clearing the output predicate of *fprcpa*:

$$\begin{cases} (a)\ \ ^{e}{}_{b} \le e_{min} - 1 \quad (y_i \text{ might be huge}) \\ (b)\ \ ^{e}{}_{b} \ge e_{max} - 2 \quad (y_i \text{ might be tiny}) \\ (c)\ \ e_a - e_b \ge e_{max} \quad (q_i \text{ might be huge}) \\ (d)\ \ e_a - e_b \le e_{min} + 1 \quad (q_i \text{ might be tiny}) \\ (e)\ \ e_a \le e_{min} + N - 1 \quad (r_i \text{ might lose precision}) \end{cases}$$

The same observations as in the case of *frcpa* can be made for conditions (a) and (b) (see Section 4.1.1, ‘‘Architecturally Mandated SWA Conditions for Divide’’), but in the case of *fprcpa* the action taken is that the output predicate is cleared, rather than raising a SWA fault.

When *fprcpa* asks for pseudo-SWA by clearing its output predicate, the floating-point output register will contain the approximation *fprcpa* can provide for the inverse of *b*. The code containing the *fprcpa* instruction will have to check the result predicate, if an IEEE correct answer is needed. Otherwise, for a non-IEEE divide one could simply multiply the approximation of 1/*b* by *a*. The values provided by *fprcpa* in such cases are:

- ∞, with the sign of the denominator, if $e_b \le e_{min} - 1$
- 0, with the sign of the denominator, if $e_b \ge e_{max} - 2$
- the 11-bit table approximation for the inverse of the denominator if $e_a - e_b \ge e_{max}$, $e_a - e_b \le e_{min} + 1$, or $e_a \le e_{min} + N - 1$

Note that if a pseudo-SWA request condition is met when one of the inputs is a denormal number and the denormal exceptions are enabled, then the OS kernel trap handler will first invoke the SWA handler (for a SWA fault). This will in turn convert the ISR code to that of a denormal exception

and will return it to the kernel. The operating system will then search for a corresponding user-registered floating-point exception handler.

Similar to the register file format divide operation, each of the 12 computation steps above translates into a parallel IA-64 assembly language instruction. The first and the last step use status field 0 from the FPSR (the user status field), while all the intermediate steps use status field 1 (reserved for special computations by software conventions). Steps (2) through the last are predicated by the output predicate of the *fprcpa* instruction (corresponding to step (1) above). Thus, when the output predicate of *fprcpa* (corresponding to step (1)) is cleared, steps (2) through the last will be skipped. Again, the *fprcpa* instruction and the last instruction in the sequence have to need the same output register.

Care has to be exercised when the output predicate of *fprcpa* is cleared, as there is no easy way to tell whether the floating-point output register contains the result of the parallel divide operation, or just approximations to the inverses of the two denominators. The solution is to split the parallel operands in two in both cases (result provided or approximation), to perform two scalar operations, and to pack the two halves of the result in the end. This means re-calculating the two components of the result in cases when it was already provided by *fprcpa*. This solution is preferred to the one that would check the input operands and decide whether the result is already there when the output predicate is cleared - this would only lengthen the execution time in all cases. The sequence of steps inlined by the compiler for a parallel divide operation is outlined below (but the actual implementation may be different):

	$y_0 = 1/b \cdot (1+\varepsilon_0),\ \lvert\varepsilon_0\rvert \le 2^{-m}$	*fprcpa* table lookup
	intermediate parallel computation steps	predicated by *fprcpa* output predicate
	…	
	$q_2 = (q_1 + r_1 \cdot y_3)_{rnd}$	result = q_2; predicated by *fprcpa*
	go to “done” if *fprcpa* output predicate is 1	branch predicated by *fprcpa* output predicate
	unpack parallel operands into scalar operands	
	normalize scalar operands	
	$y_0 = 1/b \cdot (1+\varepsilon_0),\ \lvert\varepsilon_0\rvert \le 2^{-m}$	*frcpa* table lookup for the low half
	…	
	intermediate scalar computation steps - low half	predicated by *frcpa* output predicate
	…	
	$q'_3 = (q_3)_{rnd}$	low result = q'_3; predicated by the *frcpa* output predicate
	$y_0 = 1/b \cdot (1+\varepsilon_0),\ \lvert\varepsilon_0\rvert \le 2^{-m}$	*frcpa* table lookup for the high half
	…	
	…	
	intermediate scalar computation steps - high half	predicated by *frcpa* output predicate
	…	
	$q''_3 = (q_3)_{rnd}$	high result = q''_3; predicated by *frcpa* output predicate
	pack the low half and high half of the result	
done :	store result	

Note though that the sequence outlined above has to be implemented so as to handle correctly all the floating-point exceptions.

An observation has to be made regarding the denormal floating-point exceptions raised when computing the result of a divide operation. In the case of the parallel divide, the last operation in the sequence (using the user status field 0) may receive a denormal input generated by the previous computation step (using status field 1), even though none of the original input operands *a* and *b* was denormal. This will cause raising the denormal operand exception (setting the D status flag in status field 0 if the denormal exceptions are disabled, or taking a denormal trap if they are enabled). This situation can be resolved by scaling the input operands at the beginning, and by scaling the result accordingly at the end of the computation.

5.2 Frequency Estimation of the Architecturally Mandated Pseudo-SWA Faults for Parallel Floating-point Divide

For the parallel single precision format, even though there is no architecturally mandated SWA, an evaluation similar to that for the floating-point register file format can be made for the number of points in the (e_a, e_b) plane, for which *fprcpa* will return the best approximation it can provide for the reciprocal 1 / b, but will clear the output predicate:

$$N_{tot} = (2 \cdot 127 + 1)^2 = 65025$$

$$N_{NOSWA} = (2 \cdot 127 - 3) \cdot (2 \cdot 127 - 24) - \frac{1}{2} \cdot 127 \cdot 128 - \frac{1}{2} \cdot (127 - 24 - 1) \cdot (127 - 24)$$

$$= 251 \cdot 230 - 127 \cdot 64 - 51 \cdot 103 = 57730 - 8128 - 5253 = 44349$$

$$N_{SWA} = 65025 - 44349 = 20676$$

where N_{tot} represents the total number of possible pairs of exponents for a and b, N_{NOSWA} is the number of points for which user level software assistance is not needed, and N_{SWA} denotes the number of points for which user level software assistance (pseudo-SWA) is needed.

From here:

$$N_{SWA} / N_{tot} = (20676 / 65025) \cdot 100 = 31.797\ \%$$

This calculation assumes that all the possible exponents of *a* and *b* have equal probability of occurring when computing *a*/*b*.

5.3 Architecturally Mandated Pseudo-SWA Conditions for Parallel Floating-point Square Root

For the parallel square root, if $\sqrt{a_1}$ and $\sqrt{a_2}$ have to be calculated, the *fprsqrta* instruction provides initial approximations of $1/\sqrt{a_1}$ and $1/\sqrt{a_2}$ that allow starting a Newton-Raphson or similar iterative process to compute the correctly rounded values of $\sqrt{a_1}$ and $\sqrt{a_2}$ as specified by the IEEE-754 Standard for Binary Floating-point Computations [3]. The sample algorithm presented below might not generate the IEEE correct result if some of the intermediate computation steps lose precision. For simplicity, the computation is presented only for one set of single precision input values.

Consider the following algorithm for calculating $\sqrt{a}$ in single precision, where y_0 is an 11-bit approximation of $1/\sqrt{a}$, the values *a*, *h*, t_1, t_2, t_4, y_1, *S*, *H*, D, S_1, H_1, d_1, and *R* are floating-point

numbers with $N_s = 24$ bits in the significand (single precision), *rn* is the IEEE rounding to nearest mode, and *rnd* is any IEEE rounding mode. The precision of the calculation is indicated for each step.

1. $y_0 = 1/\sqrt{a} \cdot (1 + \varepsilon_0),\ \lvert\varepsilon_0\rvert \le 2^{-m},\ m = 8.831$	table lookup
2. $h = (1/2 \cdot y_0)_{rn}$	single precision
3. $t_1 = (a \cdot y_0)_{rn}$	single precision
4. $t_2 = (1/2 - t_1 \cdot h)_{rn}$	single precision
5. $y_1 = (y_0 + t_2 \cdot y_0)_{rn}$	single precision
6. $S = (a \cdot y_1)_{rn}$	single precision
7. $H = (1/2 \cdot y_1)_{rn}$	single precision
8. $d = (a - S \cdot S)_{rn}$	single precision
9. $t_4 = (1/2 - S \cdot H)_{rn}$	single precision
10. $S_1 = (S + d \cdot H)_{rn}$	single precision
11. $H_1 = (H + t_4 \cdot H)_{rn}$	single precision
12. $d_2 = (a - S_1 \cdot S_1)_{rn}$	single precision
13. $R = (S_1 + d_1 \cdot H_1)_{rnd}$	single precision

The algorithm generates $R = (\sqrt{a})_{rnd}$, the correctly rounded single precision value of $\sqrt{a}$.

The condition that might cause certain intermediate steps to lose precision is the following, and it identifies situations when the Itanium processor will have to issue a pseudo-software assistance (SWA) request, by clearing the output predicate of *fprsqrta*:

$$\{\ e_a \le e_{\min} + N - 1\ (d_i \text{ might lose precision})$$

When *fprsqrta* asks for pseudo-SWA by clearing its output predicate, the floating-point output register will contain the 11-bit approximation for the inverse of $\sqrt{a}$. The code containing the *fprsqrta* instruction will have to check the result predicate, if an IEEE correct answer is needed. Otherwise, for a non-IEEE square root one could simply multiply the approximation of $1/\sqrt{a}$ by *a*.

Note that if a pseudo-SWA request condition is met when the input is a denormal number and the denormal exceptions are enabled, then the OS kernel trap handler will first invoke the SWA handler (for a SWA fault). This will in turn convert the ISR code to that of a denormal exception and will return it to the kernel. The operating system will then search for a corresponding user-registered floating-point exception handler.

Similar to the register file format square root operation, each of the 13 computation steps above translates into a parallel IA-64 assembly language instruction. The first and the last step use status field 0 from the FPSR (the user status field), while all the intermediate steps use status field 1 (reserved for special computations by software conventions). Steps (2) through the last are predicated by the output predicate of the *fprsqrta* instruction (corresponding to step (1) above). Thus, when the output predicate of *fprsqrta* (corresponding to step (1)) is cleared, steps (2) through the last will be skipped. The *fprsqrta* instruction and the last instruction in the sequence need to have the same output register.

Care has to be exercised when the output predicate of *fprsqrta* is cleared, as there is no easy way to tell whether the floating-point output register contains the result of the parallel square root operation, or just approximations to the inverse square roots of the two arguments. The solution is to split the parallel operands in two in both cases (result provided or approximation), to perform two scalar operations, and to pack the two halves of the result in the end. This means re-calculating

the two components of the result in cases when it was already provided by *fprsqrta*. This solution is preferred to the one that would check the input operands and decide whether the result is already there when the output predicate is cleared – this would only lengthen the execution time in all cases. The sequence of steps inlined by the compiler for a parallel square root operation is outlined below (but the actual implementation may be different):

	$y_0 = 1/\sqrt{a} \cdot (1+\varepsilon_0),\ \lvert\varepsilon_0\rvert \le 2^{-m}$	*fprsqrta* table lookup
	…	
	intermediate parallel computation steps	predicated by *fprsqrta* output predicate
	…	
	$R = (S_1 + d_1 \cdot H_1)_{rnd}$	result = *R*; predicated by *fprsqrta* output predicate
	go to “done” if *fprsqrta* output predicate is 1	branch predicated by *fprsqrta* output predicate
	unpack parallel operands into scalar operands	
	normalize scalar operands	
	$y_0 = 1/\sqrt{a} \cdot (1+\varepsilon_0),\ \lvert\varepsilon_0\rvert \le 2^{-m}$	*frsqrta* table lookup for the low half
	…	
	intermediate scalar computation steps - low half	predicated by *frsqrta* output predicate
	…	
	$R' = (S_1 + d_1 \cdot H_1)_{rnd}$	low result = R' ; predicated by *frsqrta* output predicate
	$y_0 = 1/\sqrt{a} \cdot (1+\varepsilon_0),\ \lvert\varepsilon_0\rvert \le 2^{-m}$	*frsqrta* table lookup for the high half
	…	
	intermediate scalar computation steps - low half	predicated by *frsqrta* output predicate
	…	
	$R'' = (S_1 + d_1 \cdot H_1)_{rnd}$	high result = R''; predicated by *frsqrta* output predicate
	pack the low and high half of the result	
done:	store result	

Note that the sequence outlined above has to be implemented so as to handle correctly all the floating-point exceptions.

An observation has to be made regarding the denormal floating-point exceptions raised when computing the result of a square root operation. In the case of the parallel square root, the last operation in the sequence (using the user status field 0) may receive a denormal input generated by the previous computation step (using status field 1), even though the original input operand *a* was not denormal. This will cause raising the denormal operand exception (setting the D status flag in status field 0 if the denormal exceptions are disabled, or taking a denormal trap if they are enabled). Just as for the parallel divide operation, this situation is resolved by scaling the input operand at the beginning, and by scaling the result accordingly at the end of the computation.

5.4 Frequency Estimation of the Architecturally Mandated Pseudo-SWA Faults for Parallel Floating-point Square Root

For the parallel single precision format, even though there is no architecturally mandated SWA, a similar evaluation can be made for the number of points on the e_a axis, for which *fprsqrta* will return the best approximation it can provide for $1/\sqrt{a}$, but will clear the output predicate:

$$N_{SWA} / N_{tot} = (24 + 1) / (2 \cdot 127 + 1) \cdot 100 = 9.8039\ \%$$

where N_{SWA} denotes the number of points for which user level software assistance (pseudo-SWA) is needed, and N_{tot} is the total number of possible exponents for a.

This calculation assumes that all the possible exponents of a have equal probability of occurring when computing $\sqrt{a}$.

Examples of Floating-point Software Assistance Requests 6

Examples for Itanium processor specific as well as IA-64 architecturally mandated SWA requests are given in the following subsections.

Examples are also given of parallel floating-point operations that lead first to SWA faults or traps, and then to other enabled floating-point exceptions. First, they are processed by the SWA handler (IA-64 Floating-point Emulation Library), and then, by an optional Floating-point IEEE Filter and by a user exception handler.

6.1 Examples of Itanium™ Processor Specific Software Assistance Requests

Itanium processor specific software assistance requests appear in two forms: Itanium processor specific software assistance faults and Itanium processor specific software assistance traps. These two cases will be illustrated separately.

6.1.1 Itanium™ Processor Specific Software Assistance Faults

Itanium processor specific software assistance faults are raised when an input operand to an IA-64 instruction is unnormal or denormal (for parallel instructions unnormals other than denormals are not representable), with the restrictions specified in Table 3-2. Software assistance in this case is necessary because the Itanium processor cannot handle input operands that are unnormal. One exception occurs for the *fnorm* operation, which multiplies its input operand by 1.0, and adds 0.0 to the result. The exception is that the *fnorm* instruction with an unnormal input will raise a SWA fault only if the exponent of the input operand is 0, or if the denormal exceptions are enabled. In this latter case, the SWA handler (the IA-64 Floating-point Emulation Library), will raise a denormal operand (D) exception, and the operating system will look for a user defined handler for denormal exceptions.

A few examples of operations raising (or not raising) Itanium processor specific SWA faults follow. The examples, illustrating only the *fma* operation, are meant to cover a variety of situations that are possible. In all the cases when a SWA fault is raised, it is because one of the operands is unnormal (or denormal).

Example 1 fma.s0 f1 = f3, f4, f2
with rc = 0x00 (rounding to nearest), pc = 0x11 (64-bit significand), wre = 1 (17-bit exponent), exceptions disabled, and:

$$f3 = 1.0 \cdot 2^{0}$$

$$f4 = 0.0111\ldots1 \cdot 2^{e_{min} + 7} = 0.0111\ldots1 \cdot 2^{-65534 + 7}$$

$$f2 = 0.0$$

The *fma* instruction raises an Itanium processor specific SWA fault because f4 contains an unnormal.

Floating-point status flags set: D, indicating an unnormal operand.
The result, normal and exact, is:

$$f1 = f3 \cdot f4 + f2 = 1.11\ldots100 \cdot 2^{e_{min}+5} = 1.11\ldots100 \cdot 2^{-65534+5}$$

Note that if the denormal exceptions are enabled, the SWA handler raises a denormal exception. Compare this example with Example 7 below, for *fnorm*.

Example 2 fma.s0 f1 = f3, f4, f2
with rc = 0x00 (rounding to nearest), pc = 0x11 (64-bit significand), wre = 1 (17-bit exponent), exceptions disabled, and:

$$f3 = 1.0 \cdot 2^{0}$$
$$f4 = 0.0111\ldots1 \cdot 2^{e_{min}} = 0.0111\ldots1 \cdot 2^{-65534}$$
$$f2 = 0.0$$

The *fma* instruction raises an Itanium processor specific SWA fault because f4 contains a denormal.
Floating-point status flags set: D, indicating a denormal operand.
The result, tiny and exact, is:

$$f1 = f3 \cdot f4 + f2 = 0.0111\ldots1 \cdot 2^{e_{min}} = 0.0111\ldots1 \cdot 2^{-65534}$$

Note that if the denormal exceptions are enabled, the SWA handler raises a denormal exception. If the denormal exceptions are disabled but the underflow exceptions are enabled, an underflow trap is taken. Compare this example with Example 10 below, for *fnorm* (from Section 6.1.2).

Example 3 fma.s0 f1 = f3, f4, f2
with rc = 0x00 (rounding to nearest), pc = 0x11 (64-bit significand), wre = 1 (17-bit exponent), exceptions disabled, and:

$$f3 = 0.011\ldots1 \cdot 2^{e_{min}} = 0.011\ldots1 \cdot 2^{-65534}$$
$$f4 = 1.0 \cdot 2^{-3}$$
$$f2 = 0.0$$

The *fma* instruction raises an Itanium processor specific SWA fault because f3 contains a denormal.
Floating-point status flags set: D, indicating a denormal operand; U, indicating a result that is tiny and inexact; I, indicating a result that is inexact.
The result, tiny and inexact, is:

$$f1 = f3 \cdot f4 + f2 = 0.000100\ldots0 \cdot 2^{e_{min}} = 0.000100\ldots0 \cdot 2^{-65534}$$

Note that if the denormal exceptions are enabled, the SWA handler raises a denormal exception. If the denormal exceptions are disabled but the underflow exceptions are enabled, an underflow trap is taken. If the denormal and underflow exceptions are disabled but the inexact exceptions are enabled, an inexact trap is taken.

Example 4 fma.s0 f1 = f3, f4, f2
with rc = 0x00 (rounding to nearest), pc = 0x11 (64-bit significand), wre = 1 (17-bit exponent), exceptions disabled, and:

$$f3 = 0.011\ldots1000 \cdot 2^{e_{min}} = 0.011\ldots1000 \cdot 2^{-65534}$$

$$f4 = 1.0 \cdot 2^{-3}$$

$$f2 = 0.0$$

The *fma* instruction raises an Itanium processor specific SWA fault because f3 contains a denormal.
Floating-point status flags set: D, indicating a denormal operand.
The result, tiny and exact, is:

$$f1 = f3 \cdot f4 + f2 = 0.000011\ldots1 \cdot 2^{e_{min}} = 0.000011\ldots1 \cdot 2^{-65534}$$

Note that if the denormal exceptions are enabled, the SWA handler raises a denormal exception. If the denormal exceptions are disabled but the underflow exceptions are enabled, an underflow trap is taken.

Example 5 fma.s0 f1 = f3, f4, f2
with rc = 0x00 (rounding to nearest), pc = 0x00 (24-bit significand), wre = 0 (15-bit exponent), exceptions disabled, and:

$$f3 = 1.0 \cdot 2^{0}$$

$$f4 = 0.01111111111111111111111100\ldots0 \cdot 2^{e_{min}}$$

$$0.01111111111111111111111100\ldots0 \cdot 2^{-16382}$$

$$f2 = 0.0$$

Note that f4 is represented as an IA-32 stack single real denormal (1-bit sign + 15-bit exponent + 24-bit significand). In floating-point register file format, it will have a biased exponent of 0 instead of 0xc001.
The *fma* instruction raises an Itanium processor specific SWA fault because f4 contains a denormal.
Floating-point status flags set: D, indicating a denormal operand.
The result, tiny and exact, is:

$$f1 = f3 \cdot f4 + f2 = 0.01111111111111111111111100\ldots0 \cdot 2^{e_{min}}$$

$$0.01111111111111111111111100\ldots0 \cdot 2^{-16382}$$

Note that if the denormal exceptions are enabled, the SWA handler raises a denormal exception. If the denormal exceptions are disabled but the underflow exceptions are enabled, an underflow trap is taken. Compare this example with Example 8 below, for *fnorm*.

Example 6 fma.s0 f1 = f3, f4, f2
with rc = 0x00 (rounding to nearest), pc = 0x10 (53-bit significand), wre = 0 (15-bit exponent), exceptions disabled, and:

$$f3 = 1.0 \cdot 2^{0}$$

$$f4 = 0.011111\ldots100000000000 \cdot 2^{e_{min}} = 0.011111\ldots100000000000 \cdot 2^{-16382}$$

$$f2 = 0.0$$

Note that f4 is represented as an IA-32 stack double real denormal (1-bit sign + 15-bit exponent + 53-bit significand). In floating-point register file format, it will have a biased exponent of 0 instead of 0xc001.
The *fma* instruction raises an Itanium processor specific SWA fault because f4 contains a denormal.
Floating-point status flags set: D, indicating a denormal operand.
The result, tiny and exact, is:

$$f1 = f3 \cdot f4 + f2 = 0.0111\ldots100000000000 \cdot 2^{e_{min}} = 0.0111\ldots100000000000 \cdot 2^{-16382}$$

Note that if the denormal exceptions are enabled, the SWA handler raises a denormal exception. If the denormal exceptions are disabled but the underflow exceptions are enabled, an underflow trap is taken.

Example 7 fnorm.s0 f1 = f3
with rc = 0x00 (rounding to nearest), pc = 0x11 (64-bit significand), wre = 1 (17-bit exponent), exceptions disabled, and:

$$f3 = 0.0111\ldots1 \cdot 2^{e_{min}+7} = 0.0111\ldots1 \cdot 2^{-65534+7}$$

Floating-point status flags set: D, indicating a denormal operand.
The result, normal and exact, is:

$$f1 = f3 \cdot 1.0 + 0.0 = 1.11\ldots100 \cdot 2^{e_{min}+5} = 1.11\ldots100 \cdot 2^{-65534+}$$

The *fnorm* instruction (pseudo-op for *fma*) does not raise a SWA fault, even though f3 contains an unnormal, because its biased exponent in floating-point register file format is not 0, and the denormal exceptions are disabled. To compute the result, which is normal and exact, the hardware shifts left the significand and decrements the exponent. Note that if the denormal exceptions are enabled, the SWA handler raises a denormal exception. Compare this example with Example 1 above, for *fma*.

Example 8 fnorm.s0 f1 = f3
with rc = 0x00 (rounding to nearest), pc = 0x00 (24-bit significand), wre = 0 (15-bit exponent), exceptions disabled, and:

$$f3 = 0.01111111111111111111111100\ldots0 \cdot 2^{e_{min}} = 0.01111111111111111111111100\ldots0 \cdot 2^{-16382}$$

Note that f3 is represented as an IA-32 stack single real denormal (1-bit sign + 15-bit exponent + 53-bit significand). In floating-point register file format, it will have a biased exponent of 0 instead of 0xc001.

The *fnorm* instruction (pseudo-op for *fma*) raises an Itanium processor specific SWA fault because f3 contains a denormal, and its biased exponent in floating-point register file format is 0.

Floating-point status flags set: D, indicating a denormal operand.

The result, tiny and exact, is:

$$f1 = f3 \cdot 1.0 + 0.0 = 0.01111111111111111111111100\ldots0 \cdot 2^{e_{min}} = 0.01111111111111111111111100\ldots0 \cdot 2^{-16382}$$

If the denormal exceptions are enabled, then a SWA fault is raised, but the SWA handler raises a denormal exception. If the denormal and the underflow exceptions are disabled, the result, tiny and exact, will leave unchanged the U status flag in the appropriate status field of the FPSR. If the underflow exceptions are enabled, the tiny result causes an underflow trap to be raised. Compare this example with Example 5 above, for *fma*.

Example 9 fnorm.s0 f1 = f3

with rc = 0x00 (rounding to nearest), pc = 0x11 (64-bit significand), wre = 0 (15-bit exponent), exceptions disabled, and:

$$f3 = 0.0111\ldots1 \cdot 2^{e_{min}} = 0.0111\ldots1 \cdot 2^{-16382}$$

Note that f3 is represented as an IA-32 stack double-extended real denormal (1-bit sign + 15-bit exponent + 64-bit significand). In floating-point register file format, it will have a biased exponent of 0 instead of 0xc001.

The *fnorm* instruction (pseudo-op for *fma*) raises an Itanium processor specific SWA fault because f3 contains a denormal, and its biased exponent in floating-point register file format is 0.

Floating-point status flags set: D, indicating a denormal operand.

The result, tiny and exact in double-extended precision, is:

$$f1 = f3 \cdot 1.0 + 0.0 = 0.0111\ldots1 \cdot 2^{e_{min}} = 0.0111\ldots1 \cdot 2^{-16382}$$

If the denormal exceptions are enabled, then a SWA fault is raised, but the SWA handler raises a denormal exception. If the denormal and the underflow exceptions are disabled, the result, tiny and exact, will leave unchanged the U status flag in the appropriate status field of the FPSR. If the underflow exceptions are enabled, the tiny result causes an underflow trap to be raised.

Sample source code for the examples above is included in Section 6.1.3.

6.1.2 Itanium™ Processor Specific Software Assistance Traps

Itanium processor specific software assistance traps are raised when the underflow traps are disabled, if the result of a floating-point instruction is tiny (tininess is evaluated after rounding to the destination precision, with unbounded exponent range), and the flush-to-zero mode is disabled. This means that the result requires denormalization, i.e. shifting right of the significand and incrementing the exponent (in the simplest case), operations that the hardware cannot perform.

A few examples of operations raising (or not) Itanium processor specific SWA traps follow.

Example 10 fnorm.s0 f1 = f3
with rc = 0x00 (rounding to nearest), pc = 0x11 (64-bit significand), wre = 1 (17-bit exponent), exceptions disabled, and:

$$f3 = 0.0111\ldots1 \cdot 2^{e_{min}}$$

Floating-point status flags set: D, indicating a denormal operand.
The result, tiny and exact, will be:

$$f1 = f3 \cdot 1.0 + 0.0 = 0.0111\ldots1 \cdot 2^{e_{min}}$$

In the example above, the *fnorm* instruction (pseudo-op for *fma*) does not raise a SWA fault, even though f3 contains a denormal, because its biased exponent in floating-point register file format is not 0, and the denormal exceptions are disabled. If the denormal exceptions are enabled, then a SWA fault is raised, but the SWA handler raises further a denormal exception. If the denormal and underflow exceptions are disabled, the result, tiny and exact, will leave unchanged the U status flag in the appropriate status field of the FPSR, and will cause raising an Itanium processor specific SWA trap. If the underflow exceptions are enabled, the tiny result will cause raising of an underflow trap. Compare this example with Example 2 above, for *fma*.

Example 11 fma.s0 f1 = f3, f4, f2
with rc = 0x00 (rounding to nearest), pc = 0x11 (64-bit significand), wre = 1 (17-bit exponent), exceptions disabled, and:

$$f3 = 1.11\ldots100 \cdot 2^{e_{min}} = 1.11\ldots100 \cdot 2^{-65534}$$
$$f4 = 1.0 \cdot 2^{-5}$$
$$f2 = 0.0$$

The *fma* instruction raises a SWA trap because its result is a tiny floating-point number.
Floating-point status flags set: U, indicating a result that is tiny and inexact; I, indicating a result that is inexact.
The result, tiny and inexact, is:

$$f1 = f3 \cdot f4 + f2 = 0.000100\ldots0 \cdot 2^{e_{min}} = 0.000100\ldots0 \cdot 2^{-65534}$$

Note that if the underflow exceptions are enabled, the SWA handler raises an underflow exception. If the underflow exceptions are disabled, but the inexact exceptions are enabled, then the SWA handler raises an inexact exception.

Example 12 fma.s0 f1 = f3, f4, f2
with rc = 0x00 (rounding to nearest), pc = 0x11 (64-bit significand), wre = 1 (17-bit exponent), exceptions disabled, and:

$$f3 = 1.11\ldots100000 \cdot 2^{e_{min}} = 1.11\ldots100000 \cdot 2^{-65534}$$
$$f4 = 1.0 \cdot 2^{-5}$$
$$f2 = 0.0$$

The *fma* instruction raises a SWA trap because its result is a tiny floating-point number.
Floating-point status flags set: none.

The result, tiny and exact, is:

$$f1 = f3 \cdot f4 + f2 = 0.0000111\ldots1 \cdot 2^{e_{min}} = 0.0000111\ldots1 \cdot 2^{-65534}$$

Note that if the underflow exceptions are enabled, the SWA handler raises an underflow exception.

Example 13 fma.s0 f1 = f3, f4, f2

with rc = 0x01 (rounding to minus infinity), pc = 0x00 (24-bit significand), wre = 1 (17-bit exponent), exceptions disabled, and:

$$f3 = 1.11\ldots100000 \cdot 2^{e_{min}} = 1.11\ldots100000 \cdot 2^{-65534}$$

$$f4 = 1.0 \cdot 2^{-5}$$

$$f2 = 0.0$$

The *fma* instruction raises a SWA trap because its result is tiny.

Floating-point status flags set: U, indicating a result that is tiny and inexact; I, indicating a result that is inexact.

The result, tiny and inexact, is:

$$f1 = f3 \cdot f4 + f2 = 0.0000111111111111111111100\ldots0 \cdot 2^{e_{min}}$$

$$0.0000111111111111111111100\ldots0 \cdot 2^{-65534}$$

Note that if the underflow exceptions are enabled, the SWA handler raises an underflow exception. If the underflow exceptions are disabled, but the inexact exceptions are enabled, then the SWA handler raises an inexact exception.

Example 14 fma.s0 f1 = f3, f4, f2

with rc = 0x10 (rounding to plus infinity), pc = 0x00 (24-bit significand), wre = 0 (15-bit exponent), exceptions disabled, and:

$$f3 = 1.11\ldots100000 \cdot 2^{e_{min}(17\ bits)} = 1.11\ldots100000 \cdot 2^{-65534}$$

$$f4 = 1.0 \cdot 2^{-5}$$

$$f2 = 0.0$$

(Note that the value in f3 is outside the range of numbers representable with 15-bit exponents.)

The *fma* instruction raises a SWA trap because its result is tiny.

Floating-point status flags set: U, indicating a result that is tiny and inexact; I, indicating a result that is inexact.

The result, tiny and inexact, is:

$$f1 = f3 \cdot f4 + f2 =$$

$$0.00000000000000000000100\ldots0 \cdot 2^{e_{min}} =$$

$$0.00000000000000000000100\ldots0 \cdot 2^{-16382}$$

Note that if the underflow exceptions are enabled, the SWA handler raises an underflow exception. If the underflow exceptions are disabled, but the inexact exceptions are enabled, then the SWA handler raises an inexact exception.

Sample source code for the examples above is included also in Section 6.1.3 below.

6.1.3 Sample Code for Examples of Itanium™ Processor Specific Software Assistance Faults and Traps

Sample source code for the examples presented in the two previous subsections is included next. A simple test driver (written in C), calls the IA-64 assembly routines run_fma () and run_fnorm () to execute the *fma* and *fnorm* operations with given input operands and Floating-point Status Register (run_fnorm () is not shown).

main.c:

```
#include <stdio.h>

typedef struct {
        __int64 LowPart;
        __int64 HighPart;
} FLOAT128;

void run_fma (unsigned __int64 *fpsr,
        FLOAT128 *d, FLOAT128 *a, FLOAT128 *b, FLOAT128 *c); // d = a * b + c
void run_fnorm (unsigned __int64 *fpsr, FLOAT128 *d, FLOAT128 *a);
        // d = a * 1.0 + 0.0

void
main ()
{
        FLOAT128 a, b, c, d;
        unsigned __int64 fpsr;
        int *p;

        p = (int *)&d;

        // ********************************************************************* //
        // ************ ITANIUM PROCESSOR SPECIFIC SWA FAULTS *************** //
        // ********************************************************************* //

        // Example 1

        // unnormal operand(s), traps disabled, rn
        // fpsr = 0x003bf - sf0: rc=00, pc=11, wre=1, traps dis.
        // 1.0 * 2^0 * 0.011...1 * 2^(e_min_17_bits + 7) + 0.0 =
        //             1.11...100 * 2^(e_min_17_bits + 5)
        fpsr = (unsigned __int64)0x03bf;
        a.HighPart = 0x000000000000ffff; a.LowPart = 0x8000000000000000;
        b.HighPart = 0x0000000000000008; b.LowPart = 0x3fffffffffffffff;
        c.HighPart = 0x0000000000000000; c.LowPart = 0x0000000000000000;
        run_fma (&fpsr, &d, &a, &b, &c);
        printf ("RN d [HH, HL, LH, LL] = %8x %8x %8x %8x PUOZDI = %2x\n"
                p[3], p[2], p[1], p[0], (short int)((fpsr >> 13) & 0x3f));
        ...

        // Example 7

        // unnormal operand(s), traps disabled, rn
        // fpsr = 0x003bf - sf0: rc=00, pc=11, wre=1, traps dis.
        // 0.011...1 * 2^(e_min_17_bits + 7) * 1.0 * 2^0 + 0.0 =
```

```
        //              1.11...100 * 2^(e_min_17_bits + 5)
        fpsr = (unsigned __int64)0x03bf;
        a.HighPart = 0x0000000000000008; a.LowPart = 0x3fffffffffffffff;
        b.HighPart = 0x000000000000ffff; b.LowPart = 0x8000000000000000;
        c.HighPart = 0x0000000000000000; c.LowPart = 0x0000000000000000;
        run_fnorm (&fpsr, &d, &a);
        printf ("RN d [HH, HL, LH, LL] = %8x %8x %8x %8x PUOZDI = %2x\n",
               p[3], p[2], p[1], p[0], (short int)((fpsr >> 13) & 0x3f));
        ...

        // ************************************************************** //
        // *************** ITANIUM PROCESSOR SPECIFIC SWA TRAPS ******************
//
        // ************************************************************** //

        // Example 10

        // unnormal operand(s), traps disabled, rn
        // fpsr = 0x003bf - sf0: rc=00, pc=11, wre=1, traps dis.
        // 0.011...1 * 2^(e_min_17_bits) * 1.0 * 2^0 + 0.0 =
        //              0.011...1 * 2^(e_min_17_bits)
        fpsr = (unsigned __int64)0x03bf;
        a.HighPart = 0x0000000000000001; a.LowPart = 0x3fffffffffffffff;
        b.HighPart = 0x000000000000ffff; b.LowPart = 0x8000000000000000;
        c.HighPart = 0x0000000000000000; c.LowPart = 0x0000000000000000;
        run_fnorm (&fpsr, &d, &a);
        printf ("RN d [HH, HL, LH, LL] = %8x %8x %8x %8x PUOZDI = %2x\n",
               p[3], p[2], p[1], p[0], (short int)((fpsr >> 13) & 0x3f));

        // Example 11

        // normal operands, unnormal result, traps disabled, rn
        // fpsr = 0x003bf - sf0: rc=00, pc=11, wre=1, traps dis.
        // 1.11...100 * 2^(e_min_17_bits) * 1.0 * 2^(-5) + 0.0 =
        //              0.00001...1 * 2^(e_min_17_bits)
        fpsr = (unsigned __int64)0x03bf;
        a.HighPart = 0x0000000000000001; a.LowPart = 0xfffffffffffffffc;
        b.HighPart = 0x000000000000fffa; b.LowPart = 0x8000000000000000;
        c.HighPart = 0x0000000000000000; c.LowPart = 0x0000000000000000;
        run_fma (&fpsr, &d, &a, &b, &c);
        printf ("RN d [HH, HL, LH, LL] = %8x %8x %8x %8x PUOZDI = %2x\n"
               p[3], p[2], p[1], p[0], (short int)((fpsr >> 13) & 0x3f));
        ...
}
```

run_fma.s

```
        .file "run_fma.s"
        .section .text
        .align 32
        .proc run_fma#
        .global run_fma#
        .align 32

run_fma:
{ .mmi
```

```
        alloc r31=ar.pfs,5,2,0,0 // r32, r33, r34, r35, r36, r37, r38
        // &fpsr is in r32
        // &fr1 (output) is in r33
        // &fr2 (input) is in r34
        // &fr3 (input) is in r35
        // &fr4 (input) is in r36

        // save old FPSR in r37
        mov r37 = ar40
        nop.i 0;;
} { .mmi
        // load new fpsr in r38
        ld8 r38 = [r32];;
        // set new value of FPSR
        mov ar40 = r38
        nop.i 0;;
} { .mmi
        // load first input argument into f8
        ldf.fill f8 = [r34]
        // load second input argument into f9
        ldf.fill f9 = [r35]
        nop.i 0;;
} { .mmi
        // load third input argument into f10
        ldf.fill f10 = [r36]
        nop.m 0
        nop.i 0;;
} { .mfi
        nop.m 0
        (p0) fma.s0 f11 = f8, f9, f10 // f11 = f8 * f9 + f10
        nop.i 0;;
} { .mmi
        // store result
        stf.spill [r33] = f11
        // save new FPSR in r38
        mov r38 = ar40
        nop.i 0;;
} { .mmi
        // store new fpsr from r38
        st8 [r32] = r38
        // restore FPSR
        mov ar40 = r37
        nop.i 0;;
} { .mib
        nop.m 0
        nop.i 0
        // return
        br.ret.sptk b0
}
        .endp run_fma
```

6.2 Examples of IA-64 Architecturally Mandated Software Assistance Requests

Divide

1) Condition (a):

$$e_b \leq e_{min} - 1$$

Example: The exponents of a = $1.11 \cdot 2^{-30000}$ and b = $0.00000101 \cdot 2^{-65534} = 1.01 \cdot 2^{-65540}$ satisfy condition (a), but no other condition for architecturally mandated software assistance.

2) Condition (b):

$$e_b \geq e_{max} - 2$$

Example:

The pair

$$a = 1.0 \cdot 2^{1000}$$

$$b = 1.111\ldots1 \cdot 2^{e_{max}-2} = 1.111\ldots1 \cdot 2^{65533}$$

satisfies condition (b), but no other condition for software assistance.

As an exercise, the precision loss in y_i can be evaluated in this case. Without a software assistance request from *frcpa*, the algorithm for double-extended and floating-point register file format inputs generates the value y_1* (the value of the second approximation y_1 of 1/*b* before rounding) that is shown below:

$$y_1^* = (2^{64} - 2^{40} + 1 + 2^{-9} + 2^{-20}) \cdot 2^{-65533-65} =$$
$$(2^{63} + 2^{62} + \ldots + 2^{40} + 1 + 2^{-9} + 2^{-20}) \cdot 2^{-65533-65} =$$
$$(1 + 2^{-1} + \ldots + 2^{-24} + 2^{-63} + 2^{-72} + 2^{-83}) \cdot 2^{-65535}$$

But this is tiny, as e_{min} = –65534, and will be represented as a denormal. When denormalizing, the least significant bit which is 1 (the 64-th bit above) is shifted out, and by rounding to nearest, the previous bit, which was 0, becomes 1:

$$y_1 = 0.111\ldots1100\ldots01 \cdot 2^{-65534}$$

where the fraction contains 24 consecutive 1's, followed by 39 consecutive 0's, followed by a 1. Represented in hexadecimal on 82 bits as a floating-point register file format number (17-bit exponent and 64-bit significand) this becomes:

00001 7*ffff*80000000001

The effect of this accuracy loss in y_1 is that the final result for *a/b* might be incorrect (in reality, for this particular case, the precision loss is not catastrophic, i.e. the final result would not be affected; other examples can be found for condition (b) where the precision loss really matters).

Without the architecturally mandated SWA requests, the operation producing y_1 (an *fma*) would create a denormal, and it would lead to an Itanium processor specific SWA trap. The operation consuming y_1 (also an *fma*), would then raise a SWA fault. This may further happen also for y_2, y_3, and y_4. Therefore (not mentioning accuracy), the performance is better if *frcpa* asks directly for SWA in this case.

3) Condition (c):

$$e_a - e_b \geq e_{max}$$

Example: The exponents of a = $1.01 \cdot 2^{65533}$ and b = $1.11 \cdot 2^{-65530}$ satisfy condition (c), but no other condition for architecturally mandated software assistance.

4) Condition (d):

$$e_a - e_b \leq e_{min} + 1$$

Example: The exponents of a = $1.01 \cdot 2^{-65436}$ and b = $1.11 \cdot 2^{65530}$ satisfy condition (d), but no other condition for architecturally mandated software assistance.

5) Condition (e):

$$e_a \leq e_{min} + N - 1$$

Example 1: if

$$a = 1.111\ldots110001 \cdot 2^{e_{min} + N - 17} = 1.111\ldots110001 \cdot 2^{-65487}$$

and

$$b = 1.111\ldots1110000 \cdot 2^{-10}$$

(with 64-bit significands), the algorithm for double-extended precision and register file format floating-point numbers produces $a/b = 1.0 \cdot 2^{-65477}$, instead of the correctly rounded result $a/b = 1.0...01 \cdot 2^{-65477}$.

Example 2: The exponents of a = $1.01 \cdot 2^{-65530}$ and b = $1.11 \cdot 2^{-65530}$ satisfy condition (e), but no other condition for architecturally mandated software assistance.

Note that all the cases when *a* or *b* are denornal in floating-point register file format, are covered by the conditions for architecturally mandated SWA. In such cases, the SWA handler for *frcpa* will return the correct result for the divide. For *fprcpa*, if any input is denormal, the SWA handler will just return an approximation for 1 / b (but not necessarily one that can be used to start a Newton-Raphson or similar iterative calculation for the result of the divide operation).

Square Root

The condition for architecturally mandated SWA is:

$$e_a \leq e_{min} + N - 1$$

This tells that if $e_a > e_{min} + N - 1$, a valid normal value will be returned by *frsqrta* or *fprsqrta* as an approximation to $1/\sqrt{a}$.

Example 1: If *a* is a floating-point register file format value:

$$a = 1.111\ldots11 \cdot 2^{e_{min} + N - 1} = 1.111\ldots11 \cdot 2^{-65471}$$

the algorithm for double-extended precision and register file format floating-point numbers produces $\sqrt{a} = 1.0 \cdot 2^{-32735}$, instead of the correct result $\sqrt{a} = 1.11\ldots11 \cdot 2^{-32736}$. The IEEE correct value of $\sqrt{a}$ will be calculated by the SWA handler.

Example 2: The exponent of $a = 0.00000101 \cdot 2^{-65534} = 1.01 \cdot 2^{-65540}$ satisfies the condition for architecturally mandated software assistance, and the IEEE correct value of $\sqrt{a}$ will be calculated by the SWA handler.

IA-64 Floating-point Emulation Library 7

The first section describes the method for installing the IA-64 Floating-point Emulation Library as an EFI driver. The second section describes the API defined for the IA-64 Floating-point Emulation Library. The API is defined to be independent of the operating system. The last subsection illustrates the integration of IA-64 Floating-point Emulation Library with the operating system.

It is assumed that the reader is familiar with the Extensible Firmware Interface (EFI) Specification [4]. Where necessary, apprpropriate references will be made to the relevant sections of the EFI spec.

7.1 EFI Floating-point SWA Driver

This section outlines a method for incorporating the Intel provided IA-64 Floating-point Emulation Library (Floating-point Software Assistance Handler, or FP SWA handler) on an Itanium processor based platform.

The Extensible Firmware Interface (EFI) provides services for loading runtime driver images into memory. The FP SWA handler is encapsulated as an EFI runtime driver image. This allows the loading of the FP SWA handler before an operating system is loaded. The topics discussed include:

- Installing an FP SWA driver.
- Updating an FP SWA driver.
- Loading an FP SWA driver.
- Identifying an FP SWA driver from an OS Loader or an OS Kernel.

7.1.1 Introduction to EFI Drivers

The following is a brief introduction to EFI drivers. For more details, please refer to the EFI Specification [4].

EFI drivers can be coded in most high level languages, including C, and are relocatable. As with any EFI driver image, it may be loaded through several mechanisms:
1) loaded from firmware storage device
2) loaded from a file on an EFI System Partition, or
3) loaded from any location that is accessible by EFI (Network, add-in device's option ROM, etc.).

An EFI driver can also be replaced with a newer revision of the same driver. The update is performed either from the EFI firmware or under control of an OS.

Once a runtime EFI driver is loaded, the EFI firmware marks the memory used by the driver as not available for OS use and further indicates the range as requiring an OS virtual address mapping. This allows the OS to make calls into the runtime driver from within its native OS virtual mapping, and if it is appropriate for the function call to be performed with interrupts enabled. Before the OS uses any EFI components in virtual mode, it supplies EFI with the set of virtual mappings required. EFI notifies all runtime components, thus mapping the images to the new virtual addresses.

7.1.2 FP SWA EFI Driver

The FP SWA driver may be included in non-volatile firmware storage device (e.g. flash memory) along with the rest of firmware components (PAL and SAL), and/or it may be in a directory on an EFI system partition (on hard disk). To support the identification and serviceability goals, the FP SWA driver needs to be locatable and identifiable such that newer versions may automatically be applied by custom setup or OS setup functions. To accomplish this:

- When installed on an EFI System Partition, the FP SWA driver image will be called "`fpswa.efi`", and will be installed in the "`Intel Firmware`" directory within the EFI directory. The file path for the FP SWA driver from the root of an EFI System Partition is "`\EFI\Intel Firmware\fpswa.efi`".

  This allows for custom or OS setup and utility functions to locate the FP SWA image if the image resides on the EFI System Partition.

- The driver will have a monotonically increasing major and minor revision number assigned to it (these will be stored in the high, and low 16 bits of the Revision field of the FpswaInterface variable of type FPSWA_INTERFACE respectively). This revision number and the FP SWA GUID (Guaranteed Unique Identifier) will be included in a PE (portable executable file format) [5] resource on the image.

  This allows for custom or OS setup and utility functions to automatically determine if they have a new copy and the EFI system to determine if it has a newer version of the driver at load image time.

- The driver will have provision to be loaded at an operating system prescribed virtual address. The FP SWA driver implementation allows for the relocation of the entry point and demonstrates its usage.

  This allows for the OS to call the driver from within the context of the kernel, thus saving the need for heavy-weight mode transitioning operations (e.g., switch to physical mode and flushing TLB's, synchronizing processors in a multiprocessor system, etc.). See Chapter 3 of the EFI Specification [4] for more details.

In addition, the presence of an EFI System Partition install option allows providing tools that maintain the FP SWA component on the platform. These tools can update the existing FP SWA driver, even if the driver is in the ROM. The key data item to which an OSV must pay attention is the globally defined variable that prescribes the loading of the FP SWA driver. For the load order, the `DriverOrder` global variable and the associated `DriverXXXX` variable will be stored in NVRAM; please refer to Chapter 17 of the EFI Specification [4]. EFI supports overriding the built-in ROM version of a driver with one on the EFI System Partition.

7.1.2.1 OEM Requirements

OEMs must ship the platform with the Intel provided FP SWA EFI driver installed on the platform (either on firmware storage media or on EFI system partition). This ensures that the platform will have the necessary FP SWA EFI driver in the event that an OS does not have the driver or has an older revision of FP SWA EFI driver. This also provides a path for OEM's to upgrade the FP SWA EFI driver.

7.1.2.2 Operating System Vendor (OSV) Requirements

OSVs must include the latest version of the Intel provided FP SWA EFI driver with the OS and related service packs at the time of shipment. During OS or service-pack installation, the OS setup procedure shall :

1. Check for the presence of an FP SWA EFI driver on the platform.*

2. Compare the version of the FP SWA EFI driver included with the OS with the version found on the platform.
3. If the OS FP SWA EFI driver version is more recent than the version found on the platform, the OS must update the platform with the most recent version.

*If no FPSWA EFI driver is found on the platform, the OS must install the FPSWA EFI driver contained on the OS media.

For instances when the OS is installed on a machine that has the FP SWA EFI driver missing, the OS Loader shall, on failure to detect the FP SWA Protocol interface, use the EFI **`LoadImage()`** operation to dynamically load the FP SWA driver included with the OS media before transferring control to the OS setup procedure. The OS setup procedure must install the FP SWA driver on the EFI System Partition, and update the `DriverOrder` and `DriverXXXX` environment variables.

7.1.2.3 FP SWA EFI Driver Functionality

Providing the FP SWA handler as an EFI driver does not mean that code within the driver that provides the useful FP SWA functionality actually utilizes the EFI mechanism in any way other than to load the binary code into memory. The FP SWA handler code does not use the EFI services. This is similar to other EFI runtime drivers that keep their functionality lightweight. Most of the EFI core services only materialize during the boot-services phase and are not available during runtime.

7.1.2.4 FP SWA EFI Driver Implementation

The following header file example defines the **`EFI_INTEL_FPSWA`** Protocol. This includes a GUID (Guaranteed Unique Identifier) and a protocol interface structure. The GUID is used to identify the FP SWA image and FP SWA Protocol interface structure. The protocol interface structure only contains a revision field and a single entry point into the FP SWA handler.

```
#define EFI_INTEL_FPSWA
 {c41b6531-97b9-11d3-9a29-0090273fc14d}

typedef struct _FPSWA_INTERFACE {
  UINT32     Revision;
  UINT32     Reserved;
  EFI_FPSWAFpswa;
} FPSWA_INTERFACE;

typedef struct _FPSWA_RET {
  UINT64     status;
  UINT64     err1;
  UINT64     err2;
  UINT64     err3;
} FPSWA_RET;
```

```
typedef FPSWA_RET (EFIAPI *EFI_FPSWA) (
  IN UINTN         TrapType,
  IN OUT VOID      *Bundle,
  IN OUT UINT64    *pipsr,
  IN OUT UINT64    *pfsr,
  IN OUT UINT64    *pisr,
  IN OUT UINT64    *ppreds,
  IN OUT UINT64    *pifs,
  IN OUT VOID      *fp_state
);
```

On initialization, the EFI driver checks to see if there is an FP SWA driver already installed by utilizing the **`LocateHandle()`** and **`HandleProtocol()`** functions. If there is a such a driver, its *`Revision`* is checked against that of the current driver. If the current driver is newer, it unloads the previous driver. If the current driver is not newer, an error code is returned from its initialization procedure and the driver is unloaded. After all of the FP SWA drivers that have been registered with the EFI firmware have been initialized, there will be only one FP SWA driver in memory, and this driver will be the one with the highest *`Revision`*.

7.1.3 OS Loader / OS Initialization Requirements

An OS Loader or an OS Kernel is required to perform the following steps to make use of the FP SWA handler.

- Call **`LocateHandle()`** to find the handle to the FP SWA driver
- Call **`HandleProtocol()`** to retrieve the FP SWA Protocol instance.
- Save the physical address of the FP SWA Protocol interface
- Call **`ExitBootServices()`**
- Call **`SetVirtualAddressMap()`**
- Use the physical address of the FP SWA Protocol interface to retrieve the virtual address of the FP SWA entry point.
- Enable the FP SWA handler

The following code fragment shows the code required before **`ExitBootServices()`** is called.

```
#define EFI_INTEL_FPSWA                         \
    { 0xc41b6531, 0x97b9, 0x11d3,               \
  0x9a, 0x29, 0x0, 0x90, 0x27, 0x3f, 0xc1, 0x4d }

EFI_GUID FpswaId = EFI_INTEL_FPSWA;

  EFI_STATUS       Status;
```

```
UINTN             BufferSize;
  EFI_HANDLE      *HandleBuffer;
EFI_HANDLE        FpswaHandle;
  FPSWA_INTERFACE *Fpswa;

BufferSize   = sizeof(EFI_HANDLE);
Status = BS->LocateHandle(ByProtocol,
                          &FpswaId,
                          NULL,
                          &BufferSize,
                          &FpswaHandle
                          );
if(EFI_ERROR(Status))
       return(Status);/* return error */

Status =
       SystemTable->BootServices->HandleProtocol(
                         FpswaHandle,
                         &FpswaId,
                         &Fpswa
                         );
if(EFI_ERROR(Status))
       return(Status);/* return error */
```

The value of **Fpswa** is a physical pointer to the FP SWA Protocol interface. This value must be saved so that the FP SWA handler entry point can be extracted at a later time.

Once the physical address of the FP SWA interface is obtained by the OS loader or OS initialization code, the OS loader must call the ExitBootServices() function and setup the appropriate virtual mapping for the FP SWA using SetVirtualAddressMap() function. Once **ExitBootServices()** and **SetVirtualAddressMap()** have been called, the FP SWA driver will be at its new virtual address, and the FP SWA Protocol interface structure **Fpswa** will contain the virtual address of the FP SWA handler entry point. The OS can use this FP SWA handler entry point to service floating-point software assistance requests (floating-point SWA faults and floating-point SWA traps).

7.2 Floating-point SWA Handler API - API for the IA-64 Floating-point Emulation Library

There is only one top-level function in the IA-64 Floating-point Emulation Library (FP SWA handler), *Fpswa*(), which resolves all the cases of architecturally mandated SWA faults and of Itanium processor specific SWA faults and traps. The Floating-point Emulation Library also includes part of FP82 Floating-point Reference Library, that is invoked for cases of Itanium processor specific SWA faults. The pseudo-code for the FP82 Floating-point Reference Library functions used in floating-point emulation is shown in the ‘‘IA-64 Software Developer’s Guide’’ [1].

It is assumed that:

1. The FP SWA handler is in LE (little endian) mode only.
2. The FP SWA handler uses registers from the register stack.
3. The FP SWA handler uses standard C calling conventions.
4. The FP SWA handler does not call back to the OS.
5. The FP SWA handler will look at the first two fields in the data structure pointed at by its last parameter (see below), for the type of the floating-point state passed to it. If the state is less than the full floating-point register state, it will use the actual physical floating-point registers that are not present in the floating-point state for any updates.

Based on these assumptions, the operating system kernel should call the FP SWA handler as:

```
FPSWA_RET Fpswa (
        int                     trap_type,
        BUNDLE                  *pbundle,
        __int64                 *pipsr,
        __int64                 *pfpsr,
        __int64                 *pisr,
        __int64                 *ppreds,
        __int64                 *pifs,
        FP_STATE                *fp_state
);
```

where *__int64* stands for the 64-bit integer data type.

The list of parameters, their type (input and/or output), and their usage are the following (for parameters which are pointers, the input or output attribute refers to the object pointed at):

- trap_type - (input) type of exception (0 for trap, 1 for fault, invalid otherwise); this is necessary because examination of the ISR code is not sufficient to distinguish between floating-point faults and traps
- pbundle - (input) pointer to the 128-bit quantity that contains the IA-64 bundle; the OS must guarantee that the FP SWA handler will be able to read the instruction bundle; the bundle contains the floating-point instruction that caused the SWA request, and it is used for reading the opcode of the instruction; the *BUNDLE* data type is defined as follows:

```
typedef         struct  bundle_s {
        __int64         bundle_low64;
        __int64         bundle_high64;
} BUNDLE;
```

- pipsr - (output) pointer to the 64-bit quantity that contains the IPSR value (Intrerruption Processor Status Register); used to set in it the *mfl* and/or *mfh* bits that indicate whether a ‘‘low’’ floating-point register was modified (f2 to f31), or respectively a ‘‘high’’ one (f32 to f127); upon return, pipsr points to the updated interruption processor status register (IPSR) Note: the OS can clear the *mfl* and *mfh* bits in the 64-bit value pointed at by pipsr before calling Fpswa(), and can examine them upon return; if set, they indicate that some floating-point registers have been written by the FP SWA handler; whether these registers are in the save area (with pointers in fp_state) or physical floating-point registers, depends on the bit masks passed to Fpswa() in fp_state; the OS can also use the processor *mfl* and *mfh* bits, knowing that the FP SWA handler uses only floating-point registers f6 through f11
- pfpsr - (input/output) pointer to the 64-bit quantity that contains the FPSR value (Floating-point Status Register); used to read control bit settings, and to write new status flag values; upon return, pfpsr points to the updated floating-point status register (FPSR); only the

status flags of the status field used by the instruction that caused the floating-point SWA fault or trap may be changed upon return from the call to Fpswa()

- pisr - (input/output) pointer to the 64-bit quantity that contains the ISR value (Intrerruption Status Register); it is read to determine the type of an incoming floating-point exception; upon return, if a floating-point exception has to be raised by the OS to the user level, pisr points to the updated interruption status register (ISR) code for a (possibly) new floating-point exception
- ppreds - (input/output) pointer to the 64-bit quantity that contains the predicate register value; used to read the qualifying predicate of the excepting instruction; upon return, ppreds points to the updated value of the 64-bit predicate register, but only if the instruction that caused the floating-point SWA fault or trap has at least one output that is a predicate register (otherwise, ppreds points to an unchanged value)
- pifs - (input) pointer to the 64-bit quantity that contains the IFS value (Intrerruption Function State); used to read the value of CFM (the Current Frame Marker), and to extract the values of the rotating register bases for floating-point and predicate registers
- fp_state - pointer to floating-point state area of type FP_STATE, containing the saved values of the floating-point registers (the current floating-point register state); the definition of FP_STATE is as follows:

```
typedef struct fp_state_s {
        __int64  bitmask_low64; /* bitmask of FP regs f63-f2 */
        __int64  bitmask_high64; /* bitmask of FP regs f127-f64 */
        FP_STATE_LOW_PRESERVED  *fp_state_low_preserved; /* f2-f5 */
        FP_STATE_LOW_VOLATILE  *fp_state_low_volatile; /* f6-f15 */
        FP_STATE_HIGH_PRESERVED *fp_state_high_preserved; /* f16-f31 */
        FP_STATE_HIGH_VOLATILE  *fp_state_high_volatile; /* f32-f127 */
} FP_STATE;

typedef struct fp_state_low_preserved_s {
        __int128 fp_lp[4]; /* contains FP registers f2-f5 */
} FP_STATE_LOW_PRESERVED;

typedef struct fp_state_low_volatile_s {
        __int128 fp_lv[10]; /* contains FP registers f6-f15 */
} FP_STATE_LOW_VOLATILE;

typedef  struct fp_state_high_preserved_s {
        __int128 fp_hp[16]; /* contains FP registers f16-f31 */
} FP_STATE_HIGH_PRESERVED;

typedef struct fp_state_high_volatile_s {
        __int128 fp_hv[96]; /* contains FP registers f32-f127 */
} FP_STATE_HIGH_VOLATILE;
```

The bitmask_low64 and bitmask_high64 fields of FP_STATE specify bit masks for all the floating-point registers that are valid in FP_STATE. The bitmask_low64 field specifies the bit mask for registers F0-F63, and bitmask_high64 specifies the bit mask for registers F64-F127. A value of 1 in the bit mask means that the corresponding floating-point register is valid in FP_STATE, and a value of 0 means that the corresponding floating-point register is not valid in the FP_STATE, and the FP SWA handler must use the corresponding hardware floating-point register. For example, if bit 2 of bitmask_low64 is set (1), state register F2 is valid in the FP_STATE structure and is available in fp_state_low_preserved[0], which correspond to f2. An important observation is that saving and restoring floating-point registers has to be made as atomic

operations. Interruptions have to be disabled during this timeframe, otherwise a context switch could occur, another thread could write to some of the floating-point registers, and upon resuming the interrupted thread, some floating-point registers could contain incorrect data.

The return value from Fpswa is of type RET_FPSWA, which is a structure of 32 bytes maximum (contained in r8-r11). The first eight bytes of the structure indicate if the call to the FP SWA handler succeeded or failed. The return value definition is:

```
typedef struct {
        __int64 status;         // r8
        unsigned __int64 err1; // r9
        unsigned __int64 err2; // r10
        unsigned __int64 err3; // r11
} FPSWA_RET;
```

If the return value (status) indicates an error (status < 0), the OS will print an error message and the hex contents of some or all of r8, r9, r10, and r11, which contain implementation specific error codes. A return value of 0 (status == 0) indicates that the FP SWA handler has successfully emulated the instruction that caused the SWA fault or trap. A positive value (status > 0) indicates that the floating-point exception is not a SWA fault or trap, or that the FP SWA handler has converted it to another unmasked (enabled) floating-point exception that must be propagated to the user. In the latter two cases, the FP SWA handler may update the floating-point state, the IPSR, the FPSR, the ISR (only if a floating-point exception has to be propagated to the user level), and the predicate registers in order to reflect the correct state information.

The return code of the FP SWA handler, status, is as follows:

- *0* - the floating-point instruction was successfully emulated, and a result is being provided
- *-1* - the floating-point instruction emulation was unsuccessful (due to incorrect parameters to the floating-point emulation function); this represents an internal error condition, and should not occur if the FP SWA handler is invoked with correct parameters by the OS
- *>0* - indicates that the floating-point emulation was not successful, and a new floating-point exception is reported by the IA-64 Floating-point Emulation Library; up to three bits in the value of the status field are set to indicate the status of the emulation; the following status bits are defined:
  - bit 0 - if set, indicates a new floating-point exception to be raised, that needs to be delivered to the user; the actual floating-point exception type is indicated in the updated ISR register.
  - bit 1 - if set, an incoming floating-point fault was converted to an outgoing floating-point trap by the IA-64 Floating-point Emulation Library, otherwise the type of the incoming exception (fault or trap) is maintained for the new exception to be raised; this bit is only tested if bit 0 is set
  - bit 2 - if set, it indicates that a parallel (SIMD) instruction has caused the exception; this bit is only tested if bit 0 is set; an OS will use this bit only if it wants to provide more information to the user handler than that in the ISR code (e.g. it could indicate the occurrence of a floating-point fault, of a floating-point trap, of multiple floating-point faults, or of multiple floating-point traps; note though that such information - SIMD instruction or not - can be derived also from the opcode of the instruction that caused the exception)
  - bits [3-63] - reserved

A template for interpreting the encoded error message when the return value from Fpswa() to the OS kernel, fpswa_ret, contains fpswa_ret.status == -1, is presented below (note that the calls to fprintf() will have to be replaced as needed for every OS).

```
#include <stdio.h>
#define FP_EMUL_ERROR -1

void FPSWA_error_print (FPSWA_RET fpswa_ret)
{

        unsigned int err_nr; // error number
        unsigned int qp; // qualifying predicate
        unsigned __int64 OpCode; // instruction opcode
        unsigned int rc; // rounding control
        unsigned int significand_size; // significand size: 24, 53, or 64
        unsigned int ISRlow; // ISR code
        unsigned int f1; // result floating-point register index
        unsigned int sign; // sign bit of the result
        unsigned int exponent; // exponent of the result
        unsigned __int64 significand; // significand of the result
        unsigned int new_trap_type;
                // indicates that a new floating-point exception has to be raised
        if (fpswa_ret.status != (__int64)FP_EMUL_ERROR) return;
        err_nr = (unsigned int)(fpswa_ret.err1 >> 56);
                // err_nr in err1, bits 63-56
        if (err_nr == 1) {
                fprintf (stderr, "Fpswa () Internal Error 1: template FXX is "
                        "invalid\n");
        } else if (err_nr == 2) {
                fprintf (stderr, "Fpswa () Internal Error 2: instruction slot 3 "
                        "is not valid\n");
        } else if (err_nr == 3) {
                qp = (unsigned int)fpswa_ret.err1; // // qp in err1, bits 31-0 (5-0)
                fprintf (stderr, "Fpswa () Internal Error 3: qualifying predicate "
                        "PR[%ud] = 0\n", qp);
        } else if (err_nr == 4) {
                OpCode = fpswa_ret.err2; // OpCode in err2, bits 63-0
                fprintf (stderr, "Fpswa () Internal Error 4-%ud: instruction opcode"
                        " %8x%8x not recognized\n", (unsigned int)fpswa_ret.err3,
                        (unsigned int)(OpCode >> 32), (unsigned int)OpCode);
        } else if (err_nr == 5) {
                rc = (unsigned int)fpswa_ret.err1; // rc in err1, bits 31-0 (1-0)
                fprintf (stderr, "Fpswa () Internal Error 5: invalid rc = %ud\n",
                        rc);
        } else if (err_nr == 6) {
                fprintf (stderr, "Fpswa () Internal Error 6: cannot determine "
                        "the computation model\n");
        } else if (err_nr == 7) {
                significand_size = (unsigned int)(fpswa_ret.err1 >> 32);
                        // significand_size in err1, bits 55-32
                ISRlow = (unsigned int)fpswa_ret.err1; // ISRlow in err1, bits 31-0
                f1 = (unsigned int)(fpswa_ret.err2 >> 32); // f1 in err2, bits 63-32
                sign = (unsigned int) (fpswa_ret.err2 >> 17) & 0x01;
                        // tmp_fp.sign in err2, bit 17
                exponent = (unsigned int)fpswa_ret.err2 & 0x1ffff;
                        // tmp_fp.exponent in err2, bits 16-0
                significand = fpswa_ret.err3; // tmp_fp.significand in err3
                fprintf (stderr, "Fpswa () Internal Error 7: incorrect significand"
                        " size %ud for ISRlow = %4.4x and FR[%ud] = %1.1x %5.5x "
```

```
                "%8x%8x\n", significand_size, ISRlow, f1, sign,exponent,
                (unsigned int)(significand >> 32),
                (unsigned int)significand);
} else if (err_nr == 8) {
        fprintf (stderr, "Fpswa () Internal Error 8: non-tiny result for "
                "SWA trap\n");
} else if (err_nr == 9) {
        significand_size = (unsigned int)fpswa_ret.err1;
                // significand_size in err1, bits 31-0
        fprintf (stderr, "Fpswa () Internal Error 9: incorrect significand"
                " size %ud\n", significand_size);
} else if (err_nr == 10) {
        rc = (unsigned int)fpswa_ret.err1; // rc in err1, bits 31-0
        fprintf (stderr, "Fpswa () Internal Error 10: invalid rc = %ud for"
                " non-SIMD F1 instruction\n", rc);
} else if (err_nr == 11) {
        ISRlow = (unsigned int)fpswa_ret.err1;
                // ISRlow & 0x0ffff in err1, bits 31-0
        fprintf (stderr, "Fpswa () Internal Error 11: SWA trap code "
                "invoked with F1 instruction, with invalid ISR.code = %x\n",
                ISRlow);
} else if (err_nr == 12) {
        ISRlow = (unsigned int)fpswa_ret.err1;
                // ISRlow & 0x0ffff in err1, bits 31-0
        fprintf (stderr, "Fpswa () Internal Error 12: SWA trap code "
                "invoked with SIMD F1 instruction, w/o O or U set in"
                " ISR.code = %x\n", ISRlow);
} else if (err_nr == 13) {
        fprintf (stderr, "Fpswa () Internal Error 13: non-tiny result "
                "low\n");
} else if (err_nr == 14) {
        rc = (unsigned int)fpswa_ret.err1; // rc in err1, bits 31-0
        fprintf (stderr, "Fpswa () Internal Error 14-%ud: invalid rc = %ud"
                "for SIMD F1 instruction\n", (unsigned int)fpswa_ret.err2,
                rc);
} else if (err_nr == 15) {
        fprintf (stderr, "Fpswa () Internal Error 15: non-tiny result "
                "high\n");
} else if (err_nr == 16) {
        OpCode = fpswa_ret.err2;// OpCode in err2, bits 63-0
        fprintf (stderr, "Fpswa () Internal Error 16: instruction opcode "
                "%8x%8x not valid for SWA trap\n",
                (unsigned int)(OpCode >> 32), (unsigned int)OpCode);
} else if (err_nr == 17) {
        OpCode = fpswa_ret.err2; // OpCode in err2, bits 63-0
        ISRlow = (unsigned int)fpswa_ret.err3; // ISRlow in err3, bits 31-0
        fprintf (stderr, "Fpswa () Internal Error 17: Fpswa () called w/o "
                "trap_type FPFLT or FPTRAP, OpCode = %8x%8x, and ISR code "
                "= %x\n", (unsigned int)(OpCode >> 32),
                (unsigned int)OpCode, ISRlow);
} else if (err_nr == 18) {
        ISRlow = (unsigned int)fpswa_ret.err2;
        fprintf (stderr, "Fpswa () Internal Error 18: SWA fault repeated, "
                "fault_ISR_code = %x\n",ISRlow);
} else if (err_nr == 19) {
        new_trap_type = (unsigned int)fpswa_ret.err1;
```

```
                        // new_trap_type in err1, bits 31-0
                fprintf (stderr, "Fpswa () Internal Error 19: new_trap_type = %x\n",
                        new_trap_type);
        } else { // error
                fprintf (stderr, "Incorrect err_nr = %8x%8x from Fpswa ()\n",
                        (unsigned int)(fpswa_ret.err1 >> 32),
                        (unsigned int)fpswa_ret.err1);
        }
}
```

7.3 FP SWA Handler Integration with the Operating System

The operating system provides interface code to the IA-64 Floating-point Emulation library. The library is called not only for SWA faults or traps, but for all the enabled (unmasked) floating-point exceptions. The SWA faults and traps are filtered and handled by the floating-point emulation library, but other exceptions are returned to the operating system kernel for being passed on to a user-registered floating-point exception handler. New floating-point exceptions can also be raised by the floating-point emulation library while processing SWA faults or traps.

All the necessary information about the floating-point exception is passed from the kernel to the floating-point emulation library as defined by the FP SWA API, which may update necessary processor state upon return. Sample interface code is shown below (entities not defined previously are self-explanatory). The OS needs to get the FP SWA entry point from the OS loader or OS initialization code as described above.

```
#define FLTTOTRAP               2

Trap (...) {
        FP_STATE                fp_state;
        FPSWA_RET               fpswa_ret;
        ...
        Switch (trap_type) {
                Case FPFAULT:

                        // get floating-point instruction bundle pointer
                        bundle = get_fp_instruction();

                        // create proper fp_state information
                        // example for f6-f15 used by kernel
                        fp_state.bitmask_low64 = 0xffc0; // bit6..bit15
                        fp_state.bitmask_high64 = 0;

                        // f6_f15 structure contains the state of f6-f15
                        fp_state.fp_state_low_volatile = &f6_f15;

                        // call IA-64 Floating-Point Emulation Library
                        fpswa_ret = fpswa_interface->fpswa(1, &bundle, &ipsr, &fpsr,
                                &isr, &preds, &ifs, &fp_state);

                        if (fpswa_ret.status == 0) {
                                // update IPSR and IIP to execute next instruction
                                // on return
                                increment_iip (pisr, pipsr, piip);
```

```
                        return;
                    } else if (fpswa_ret.status == -1) {
                        FPSWA_error_print (fpswa_ret);
                        panic ();
                    } else {
                        if (fpswa_ret.status & FLTTOTRAP) {
                            // next exception is trap
                        increment_iip (pisr, pipsr, piip);
                    }
                    raise_exception (new exception); // raise new exception
                    return;

                Case FPTRAP:

                    // get floating-point instruction bundle pointer
                    bundle = get_fp_instruction ();

                    // create proper fp_state information
                    // example for f6-f15 used by kernel
                    fp_state.bitmask_low64 = 0xffc0; // bit6..bit15
                    fp_state.bitmask_high64 = 0;

                    // f6_f15 structure contains the state of f6-f15
                    fp_state.fp_state_low_volatile = &f6_f15;

                    // call IA-64 Floating-Point Emulation Library
                    fpswa_ret = fpswa_interface->fpswa(0, &bundle, &ipsr, &fpsr,
                        &isr, &preds, &ifs, &fp_state);

                    if (fpswa_ret.status == 0) {
                        return;
                    } else if (fpswa_ret.status == -1) {
                        FPSWA_error_print (fpswa_ret);
                        panic ();
                    } else {
                        raise_exception (new exception);
                        // raise new exception
                    }
                    return;
            }

        }
```

Sample code for increment_iip () is shown next.

```
increment_iip (pisr, pipsr, piip) {
        __int64 *pisr; // pointer to ISR
        __int64 *pipsr; // pointer to IPSR
        __int64 *piip; // pointer to IIP
        int ei; // excepting instruction slot number in bundle (0, 1, 2)
        ei = (*pisr >> 41) & 0x03;
        // advance instruction pointer
        if (ei == 0) { // no template for this case
                *pipsr = *pipsr & (__int64)0xfffff9fffffffff;
                *pipsr = *pipsr | (__int64)0x0000020000000000;
        } else if (ei == 1) { // templates: MFI, MFB
                *pipsr = *pipsr & (__int64)0xfffff9fffffffff;
```

```
                *pipsr = *pipsr | (__int64)0x0000040000000000;
            } else { // if (ei == 2) - templates: MMF
                *pipsr = *pipsr & (__int64)0xfffff9fffffffff;
                *piip = *piip + (__int64)0x10;
            }
}
```

References 8

[1] *Intel® IA-64 Architecture Software Developer's Guide*, Intel Corporation, 2000

[2] *Pentium® Pro Family Developer's Manual*, Intel Corporation, 1996

[3] *IEEE-754 Standard for Binary Floating-point Computations*, 1985

[4] *Extensible Firmware Interface Specification, Revision 0.92,* http://developer.intel.com/technology/efi/index.htm, 1999

[5] *Microsoft Portable Executable and Common Object File Format Specification, Version 0.6,* http://www.microsoft.com/hwdev/efi