IA-64 Floating-Point Operations and the IEEE Standard for Binary Floating-Point Arithmetic

Marius Cornea-Hasegan, Microprocessor Products Group, Intel Corporation
Bob Norin, Microprocessor Products Group, Intel Corporation

Index words: IA-64 architecture, floating-point, IEEE Standard 754-1985

ABSTRACT

This paper examines the implementation of floating-point operations in the IA-64 architecture from the perspective of the IEEE Standard for Binary Floating-Point Arithmetic [1]. The floating-point data formats, operations, and special values are compared with the mandatory or recommended ones from the IEEE Standard, showing the potential gains in performance that result from specific choices.

Two subsections are dedicated to the floating-point divide, remainder, and square root operations, which are implemented in software. It is shown how IEEE compliance was achieved using new IA-64 features such as fused multiply-add operations, predication, and multiple status fields for IEEE status flags. Derived integer operations (the integer divide and remainder) are also illustrated.

IA-64 floating-point exceptions and traps are described, including the Software Assistance faults and traps that can lead to further IEEE-defined exceptions. The software extensions to the hardware needed to comply with the IEEE Standard's recommendations in handling floating-point exceptions are specified. The special case of the Single Instruction Multiple Data (SIMD) instructions is described. Finally, a subsection is dedicated to speculation, a new feature in IA processors.

INTRODUCTION

The IA-64 floating-point architecture was designed with three objectives in mind. First, it was meant to allow high-performance computations. This was achieved through a number of architectural features. Pipelined floating-point units allow several operations to take place in parallel. Special instructions were added, such as fused floating-point multiply-add, or SIMD instructions, which allow the processing of two subsets of floating-point operands in parallel. Predication allows skipping operations without taking a branch. Speculation allows speculative execution chains whose results are committed only if needed. In addition, a large floating-point register file (including a rotating subset) reduces the number of save/restore operations involving memory. The rotating subset of the floating-point register file enables software pipelining of loops, leading to significant gains in performance.

Second, the architecture aims to provide high floating-point accuracy. For this, several floating-point data types were provided, and instructions new to the Intel architecture, such as the fused floating-point multiply-add, were introduced.

Third, compliance with the IEEE Standard for Binary Floating-Point Arithmetic was sought. The environment that a numeric software programmer sees complies with the IEEE Standard and most of its recommendations as a combination of hardware and software, as explained further in this paper.

Floating-Point Numbers

Floating-point numbers are represented as a concatenation of a sign bit, an M-bit exponent field, and an N-bit significand field. In some floating-point formats, the most significant bit (integer bit) of the significand is not represented. Its assumed value is 1, except for denormal numbers, whose most significant bit of the significand is 0. Mathematically

$$f = \sigma \cdot s \cdot 2^e$$

where $\sigma = \pm 1$, $s \in [1,2)$, $s = 1 + k / 2^{N-1}$, $k \in \{0, 1, 2, \ldots, 2^{N-1}-1\}$, $e \in [e_{min}, e_{max}] \cap \mathbf{Z}$ ($\mathbf{Z}$ is the set of integers), $e_{min} = -2^{M-1} + 2$, and $e_{max} = 2^{M-1} - 1$.

The IA-64 architecture provides 128 82-bit floating-point registers that can hold floating-point values in various formats, and which can be addressed in any order.

Floating-point numbers can also be stored into or loaded from memory.

IA-64 FORMATS, CONTROL, AND STATUS

Formats

Three floating-point formats described in the IEEE Standard are implemented as required: single precision (M=8, N=24), double precision (M=11, N=53), and double-extended precision (M=15, N=64). These are the formats usually accessible to a high-level language numeric programmer. The architecture provides for several more formats, listed in Table 1, that can be used by compilers or assembly code writers, some of which employ the 17-bit exponent range and 64-bit significands allowed by the floating-point register format.

Format	Format Parameters
Single precision	M=8, N=24
Double precision	M=11, N=53
Double-extended precision	M=15, N=64
Pair of single precision floating-point numbers	M=8, N=24
IA-32 register stack single precision	M=15, N=24
IA-32 register stack double precision	M=15, N=53
IA-32 double-extended precision	M=15, N=64
Full register file single precision	M=17, N=24
Full register file double precision	M=17, N=53
Full register file double-extended precision	M=17, N=64

Table 1: IA-64 floating-point formats

The floating-point format used in a given computation is determined by the floating-point instruction (some instructions have a precision control completer *pc* specifying a static precision) or by the precision control field (pc), and by the widest-range exponent (wre) bit in the Floating-Point Status Register (FPSR). In memory, floating-point numbers can only be stored in single precision, double precision, double-extended precision, and register file format ('spilled' as a 128-bit entity, containing the value of the floating-point register in the lower 82 bits).

Rounding

The four IEEE rounding modes are supported: rounding to nearest, rounding to negative infinity, rounding to positive infinity, and rounding to zero. Some instructions have the option of using a static rounding mode. For example, fcvt.fx.trunc performs conversion of a floating-point number to integer using rounding to zero.

Some of the basic operations specified by the IEEE Standard (divide, remainder, and square root) as well as other derived operations are implemented using sequences of add, subtract, multiply, or fused multiply-add and multiply-subtract operations.

In order to determine whether a given computation yields the correctly rounded result in any rounding mode, as specified by the standard, the error that occurs due to rounding has to be evaluated. Two measures are commonly used for this purpose. The first is the error of an approximation with respect to the exact result, expressed in fractions of an ulp, or unit in the last place. Let $\mathbf{F}_N$ be the set of floating-point numbers with N-bit significands and unlimited exponent range. For the floating-point number $f = \sigma \cdot s \cdot 2^e \in \mathbf{F}_N$, one ulp has the magnitude

$$1 \text{ ulp} = 2^{e-N+1.}$$

An alternative is to use the relative error. If the real number x is approximated by the floating-point number a, then the relative error ε is determined by

$$a = x \cdot (1 + \varepsilon)$$

The Floating-Point Status Register

Several characteristics of the floating-point computations are determined by the contents of the 64-bit FPSR.

A set of six trap mask bits (bits 0 through 5) control enabling or disabling the five IEEE traps (invalid operation, divide-by-zero, overflow, underflow, and inexact result) and the IA-defined denormal trap [2]. In addition, four 13-bit subsets of control and status bits are provided: status fields sf0, sf1, sf2, and sf3. Multiple status fields allow different computations to be performed simultaneously with different precisions and/or rounding modes. Status field 0 is the user status field, specifying rounding-to-nearest and 64-bit precision by default. Status field 1 is reserved by software conventions for special operations, such as divide and square root. It uses rounding-to-nearest, the 64-bit precision, and the widest-range exponent (17 bits). Status fields 2 and 3 can be used in speculative

operations, or for implementing special numeric algorithms, e.g., the transcendental functions.

Each status field contains a 2-bit rounding mode control field (00 for rounding to nearest, 01 to negative infinity, 10 to positive infinity, and 11 toward zero), a 2-bit precision control field (00 for 24 bits, 10 for 53 bits, and 11 for 64 bits), a widest-range exponent bit (use the 17-bit exponent if wre = 1), a flush-to-zero bit (causes flushing to zero of tiny results if ftz = 1), and a traps disabled bit (overrides the individual trap masks and disables all traps if td = 1, except for status field 0, where this bit is reserved). Each status field also contains status flags for the five IEEE exceptions and for the denormal exception.

The register file floating-point format uses a 17-bit exponent range, which has two more bits than the double-extended precision format, for at least three reasons. The first is related to the implementation in software of the divide and square root operations in the IA-64 architecture. Short sequences of assembly language instructions carry out these computations iteratively. If the exponent range of the intermediate computation steps is equal to that of the final result, then some of the intermediate steps might overflow, underflow, or lose precision, preventing the final result from being IEEE correct. Software Assistance (SWA) will be necessary in these cases to generate the correct results, as explained in [4]. The two (or more) extra bits in the exponent range (17 versus 15 or less) prevent the SWA requests from occurring. The second reason for having a 17-bit exponent range is that it allows the common computation of $x^2 + y^2$ to be performed without overflow or underflow, even for the largest or smallest double-extended precision numbers. Third, the 17-bit exponent range is necessary in order to be able to represent the product of all double-extended denormal numbers.

Special Values

The various floating-point formats support the IEEE mandated representations for denormals, zero, infinities, quiet NaNs (QNaNs), and signaling NaNs (SNaNs). In addition, the formats that have an explicit integer bit in the significand can also hold other types of values. These formats are double-extended, with 15-bit exponents biased by 16383 (0x3fff), and all the register file formats, with 17-bit exponents biased by 65535 (0xffff). The exponents of these additional types of values are specified below for the register file format:

> *unnormalized numbers*: non-zero significand beginning with 0 and exponent from 0 to 0x1fffe, or pseudo-zeroes with a significand of 0, and exponent from 0x1 to 0x1fffe
>
> *pseudo-NaNs*: non-zero significand and exponent of 0x1ffff (unsupported by the architecture); the pseudo-QNaNs have the second most significant bit of the significand equal to 1; this bit is 0 for pseudo-SNaNs
>
> *pseudo-infinities*: significand of zero and exponent of 0x1ffff (unsupported by the architecture)

Note that one of the pseudo-zero values, encoded on 82 bits as 0x1fffe0000000000000000, is denoted as NaTVal ('not a value') and is generated by unsuccessful speculative load from memory operations (e.g. a speculative load, in the presence of a deferred floating-point exception). It is then propagated through the speculative chain to indicate in the end that no useful result is available.

Two special categories that overload other floating-point numbers in register file format are the SIMD floating-point pairs, and the canonical non-zero integers. Both have an exponent of 0x1003e (unbiased 63). The value of the canonical non-zero integers is equal to that of the unnormal or normal floating-point numbers that they overlap with. The exponent of 63 moves the binary point beyond the least significant bit, the resulting value being the integer stored in the significand. The SIMD floating-point numbers consist of two single-precision floating-point values encoded in the two halves of the 64-bit significand of a floating-point register, with the biased exponent set to 0x1003e. For example, the 82-bit value of 0x1003e 3f800000 3f800000 represents the pair (+1.0, +1.0). Note that all the arithmetic scalar floating-point instructions have SIMD counterparts that operate on two single-precision floating-point values in parallel.

IA-64 FLOATING-POINT OPERATIONS

All the floating-point operations mandated or recommended by the IEEE Standard are or can be implemented in IA-64 [2]. Note that most IA-64 instructions [2] are predicated by a 1-bit predicate (*qp*) from the 64-bit predicate register (predicate p0 is fixed, containing always the logical value 1). For example, the fused multiply-add operation is

> (*qp*) fma.*pc.sf f1 = f3, f4, f2*

The fma instruction is executed if *qp* = 1; otherwise, it is skipped. Two instruction completers select the precision control (*pc*) and the status field (*sf*) to be used. When the qualifying predicate is not represented, it is either not necessary, or it is assumed to be p0. When *qp* = 1, fma calculates $f3 \cdot f4 + f2$, where '*pc*' can be 's', 'd', or none. If the instruction completer '*pc*' is 's', fma.s.*sf* generates a result with a 24-bit significand. Similarly, fma.d.*sf*

generates a result with a 53-bit significand. The exponent in the two cases is in the 8-bit or 11-bit range respectively if *sf*.wre = 0, and in the 17-bit range if *sf*.wre = 1. If *'pc'* is none, the precision of the computation fma.*sf f1 = f3, f4, f2* is specified by the pc field of the status field being used, *sf*.pc. The exponent size is 15 bits if *sf*.wre = 0, and 17 bits if *sf*.wre = 1.

Addition and multiplication are implemented as pseudo-ops of the floating-point multiply-add operation. The pseudo-op for addition is fadd.*pc.sf f1 = f3, f2* obtained by replacing *f4* with register f1 that contains +1.0. The pseudo-op for multiplication is fmpy.*pc.sf f1 = f3, f4*, obtained by replacing *f2* with f0 that contains +0.0.

The reason for having a fused multiply-add operation is that it allows computation of $a \cdot b + c$ with only one rounding error. Assuming rounding to nearest, fma computes

$$(a \cdot b + c)_{rn} = (a \cdot b + c) \cdot (1 + \varepsilon)$$

where $|\varepsilon| < 2^{-N}$, and N is the number of bits in the significand. The relative error above (ε) is smaller in general than that obtained with pure add and multiply operations:

$$((a \cdot b)_{rn} + c)_{rn} = (a \cdot b\,(1 + \varepsilon_1) + c) \cdot (1 + \varepsilon_2)$$

where $|\varepsilon_1| < 2^{-N}$ and $|\varepsilon_2| < 2^{-N}$.

The benefit that arises from this property is that it enables the implementation of a whole new category of numerical algorithms, relying on the possibility of performing this combined operation with only one rounding error (see the subsections on divide and square root below).

Subtraction (fsub.*pc.sf f1 = f3, f2*) is implemented as a pseudo-op of the floating-point multiply-subtract, fms.*pc.sf f1 = f3, f4, f2* (which calculates $f3 \cdot f4 - f2$) where *f4* is replaced by f1. In addition to fma and fms, a similar operation is available for the floating-point negative multiply-add operation, fnma.*pc.sf f1 = f3, f4, f2*, which calculates $-f3 \cdot f4 + f2$.

A deviation from one of the IEEE Standard's recommendations is to allow higher precision operands to lead to lower precision results. However, this is a useful feature when implementing the divide, remainder, and square root operations in software.

For parallel computations, counterparts of fma, fms, and fnma are provided. For example, fpma.*pc.sf f1 = f3, f4, f2* calculates $f3 \cdot f4 + f2$. A pair of ones (1.0, 1.0) has to be loaded explicitly in a floating-point register to emulate the SIMD floating-point add.

Divide, square root, and remainder operations are not available directly in hardware. Instead, they have to be implemented in software as sequences of instructions corresponding to iterative algorithms (described below).

Rounding of a floating-point number to a 64-bit signed integer in floating-point format is achieved by the fcvt.fx.*sf f1 = f2* instruction followed by fcvt.xf *f2 = f1*. For 64-bit unsigned integers, the similar instructions are fcvt.fxu.*sf f1 = f2* and fcvt.xuf.*pc.sf f2 = f1*. Two variations of the instructions that convert floating-point numbers to integer use the rounding-to-zero mode regardless of the rounding control bits used in the FPSR status field (fcvt.fx.trunc.*sf f1 = f2* and fcvt.fxu.trunc.*sf f1 = f2*). They are useful in implementing integer divide and remainder operations using floating-point instructions. For example, the following instructions convert a single precision floating-point number from memory (whose address is in the general register r30) to a 64-bit signed integer in r8:

```
ldfs f6=[r30];; // load single precision fp number
fcvt.fx.trunc.s0 f7=f6;; // convert to integer
getf.sig r8=f7;;
```

(Note that stop bits (;;) delimit the instruction groups.) The biased exponent of the value in f7 is set by fcvt.fx.trunc.s0 to 0x1003e (unbiased 63) and the significand to the signed integer that is the result of the conversion. (If the conversion is invalid, the significand is set to the value of Integer Indefinite, which is -2^{63}.) Since rounding to zero is used by fcvt.fx.trunc, specifying the status field only tells which status flags to set if an invalid operation, denormal, or inexact result exception occurs (Exceptions and Traps are covered later in the paper.) For the conversion from a floating-point number to a 64-bit unsigned integer, fcvt.fx.trunc above has to be replaced by fcvt.fxu.trunc.

The opposite conversion, from a 64-bit signed integer in r32 to a register-file format floating-point number in f7, is performed by

```
setf.sig f6 = r32;; //sign=0 exp=0x1003e signif.=r32
fcvt.xf f7 = f6;; // sign=sign(r32); no fp exceptions
```

where the result is an integer-valued normal floating-point number. To convert further, for example to a single precision floating-point number, one more instruction is needed

```
fma.s.s0 f8=f7,f1,f0;;
```

where the single precision format is specified statically, and status field s0 is assumed to have wre = 0.

For 64-bit unsigned integers, the similar conversion is

setf.sig f6 = r32;; // sign=0 exp=0x1003e signif.=r32

fcvt.xuf.s0 f7 = f6

where fcvt.xuf.*pc*.*sf* f7 = f6 is actually a pseudo-op for fma.*pc*.*sf* f7 = f6, f1, f0, and a synonym of fnorm.*pc*.*sf* f7 = f6 (it is assumed that status field s0 has pc = 0x3). The result is thus a normalized integer-valued floating-point number. This is important to know, since floating-point operations on unnormalized numbers lead to Software Assistance faults (as explained further in the paper), thereby slowing down performance unnecessarily.

Conversions between the different floating-point formats are achieved using floating-point load, store, or other operations. For example, the following sequence converts a single precision value from memory to double precision format, also in memory (r29 contains the address of the single precision source, and r30 that of the double precision destination):

ldfs f6 = [r29];;

fma.d.s0 f7=f6,f1,f0;;

stfd [r30] = f7

This conversion could trigger the invalid exception (for a signaling NaN operand) or the denormal operand exception. These can happen on the fma instruction, but the conversion will be correct numerically even without this instruction, as all the single precision values can be represented in the double precision format.

The opposite conversion is shown below (it is assumed that status field s0 has wre = 0):

ldfd f6=[r29];;

fma.s.s0 f7=f6,f1,f0;;

stfs [r30]=f7;;

The role of the fma.s.s0 is to trigger possible invalid, denormal, underflow, overflow, or inexact exceptions on this conversion.

Other conversions between floating-point and integer formats can be achieved with short sequences of instructions. For example, the following sequence converts a single precision floating-point value in memory to a 32-bit signed integer (correct only if the result fits on 32 bits):

ldfs f6 = [r30];; // load f6 with fp value from memory

fcvt.fx.trunc.s0 f7=f6;; // convert to signed integer

getf.sig r29 = f7;; // move the 64-bit integer to r29

st4 [r28] = r29;; // store as 32-bit integer in memory

The opposite conversion, from a 32-bit integer in memory to a single precision floating-point number in memory, is performed by

ld4 r29 = [r30];; // load r29 with 32-bit int from mem

sxt4 r28=r29;; // sign-extend

setf.sig f6 = r28;; // 32-bit integer in f6; exp=0x1003e

fcvt.xf f7=f6;; // convert to normal floating-point

fma.s.s0 f8 = f7,f1,f0;; // trigger I exceptions if any

stfs [r27] = f8;; // store single prec. value in memory.

Floating-point compare operations can be performed directly between numbers in floating-point register file format, using the fcmp instruction. For other memory formats, a conversion to register format is required prior to applying the floating-point compare instruction. From the 26 functionally distinct relations specified by the IEEE Standard, only the six mandatory ones are implemented (four directly, and two as pseudo-ops):

fcmp.eq.*sf p1, p2 = f2, f3* (test for '=')

fcmp.lt.*sf p1, p2 = f2, f3* (test for '<')

fcmp.le.*sf p1, p2 = f2, f3* (test for '<=')

fcmp.gt.*sf p1, p2 = f2, f3* (test for '>')

fcmp.ge.*sf p1, p2 = f2, f3* (test for '>=')

fcmp.unord.*sf p1, p2 = f2, f3* (test for '?')

The result of a compare operation is written to two 1-bit predicates in the 64-bit predicate register. Predicate *p1* shows the result of the comparison, while *p2* is its opposite. An exception is the case when at least one input value is NaTVal, when *p1* = *p2* = 0. A variant of the fcmp instruction is called 'unconditional' (with respect to the qualifying predicate). The difference is that if *qp* = 0, the unconditional compare

(*qp*) fcmp.eq.unc.*sf p1, p2 = f2, f3*

clears both output predicates, while

(*qp*) fcmp.eq.*sf p1, p2 = f2, f3*

leaves them unchanged.

Six more compare relations are implemented, as pseudo-ops of the above, to test for the opposite situations (neq, nlt, nle, ngt, nge, and ord). The remaining 14 comparison relations specified by the IEEE Standard can be performed based on the above.

A special type of compare instruction is fclass.*fcrel.fctype p1,p2=f2,fclass9*, that allows classification of the contents of *f2* according to the class specifier *fclass9*. The *fcrel* instruction completer can be

'm' (if *f2* has to agree with the pattern specified by *fclass9*), or 'nm' (*f2* has to disagree). The *fctype* completer can be *none* or 'unc' (as for fcmp). *fclass9* can specify one of {NaTVal, QNaN, SNaN} OR none, one or both of {positive, negative} AND none, one or several of {zero, unnormal, normal, infinity} (nine bits correspond to the nine classes that can be selected, with the restrictions specified on the possible combinations).

IA-64 FLOATING-POINT OPERATIONS DEFERRED TO SOFTWARE

A number of floating-point operations defined by the IEEE Standard are deferred to software by the IA-64 architecture in all its implementations:

- floating-point divide (integer divide, which is based on the floating-point divide operation, is also deferred to software)
- floating-point square root
- floating-point remainder (integer remainder, based on the floating-point divide operation, is also deferred to software)
- binary to decimal and decimal to binary conversions
- floating-point to integer-valued floating-point conversion
- correct wrapping of the exponent for single, double, and double-extended precision results of floating-point operations that overflow or underflow, as described by the IEEE Standard

In addition, the IA-64 architecture allows virtually any floating-point operation to be deferred to software through the mechanism of Software Assistance (SWA) requests, which are treated as floating-point exceptions. Software Assistance is discussed in detail in the sections describing the divide operation, the square root operation, and the exceptions and traps.

IA-64 FLOATING-POINT DIVIDE AND REMAINDER

The floating-point divide algorithms for the IA-64 architecture are based on the Newton-Raphson iterative method and on polynomial evaluation. If a/b needs to be computed and the Newton-Raphson method is used, a number of iterations first calculate an approximation of 1/b, using the function

$$f(y) = b - 1/y$$

The iteration step is

$$e_n = (1 - b \cdot y_n)_{rn} \approx 0$$

$$y_{n+1} = (y_n + e_n \cdot y_n)_{rn} \approx 1/b$$

where the subscript *rn* denotes the IEEE rounding to the nearest mode.

Once a sufficiently good approximation y of 1/b is determined, $q = a \cdot y$ approximates a/b. In some cases, this might need further refinement, which requires only a few more computational steps.

In order to show that the final result generated by the floating-point divide algorithm represents the correctly rounded value of the infinitely precise result a/b in any rounding mode, it was proved (by methods described in [3] and [4]) that the exact value of a/b and the final result q* of the algorithm before rounding belong to the same interval of width 1/2 ulp, adjacent to a floating-point number. Then

$$(a/b)_{rnd} = (q^*)_{rnd}$$

where *rnd* is any IEEE rounding mode.

The algorithms proposed for floating-point divide (as well as for square root) are designed, as seen from the Newton-Raphson iteration step shown above, based on the availability of the floating-point multiply-add operation, fma, that performs both the multiply and add operations with only one rounding error.

Two variants of floating-point divide algorithms are provided for single precision, double precision, double-extended and full register file format precision, and SIMD single precision. One achieves minimum latency, and one maximizes throughput.

The minimum latency variant minimizes the execution time to complete one operation. The maximum throughput variant performs the operation using a minimum number of floating-point instructions. This variant allows the best utilization of the parallel resources of the IA-64, yielding the minimum time per operation when performing the operation on multiple sets of operands.

Double Precision Floating-Point Divide Algorithm

The double precision floating-point divide algorithm that minimizes latency was chosen to illustrate the implementation of the mathematical algorithm in IA-64 assembly language. The input values are the double precision operands a and b, and the output is a/b.

All the computational steps are performed in full register file double-extended precision, except for steps (11) and (12), which are performed in full register file double precision, and step (13), performed in double-precision.

The approximate values are shown on the right-hand side.

(1) $y_0 = 1/b \cdot (1 + \varepsilon_0)$, $|\varepsilon_0| \le 2^{-m}$, m=8.886

(2) $q_0 = (a \cdot y_0)_m = a/b \cdot (1 + \varepsilon_0)$

(3) $e_0 = (1 - b \cdot y_0)_{rn} \approx -\varepsilon_0$

(4) $y_1 = (y_0 + e_0 \cdot y_0)_{rn} \approx 1/b \cdot (1 - \varepsilon_0^2)$

(5) $q_1 = (q_0 + e_0 \cdot q_0)_{rn} \approx a/b \cdot (1 - \varepsilon_0^2)$

(6) $e_1 = (e_0^2)_{rn} \approx \varepsilon_0^2$

(7) $y_2 = (y_1 + e_1 \cdot y_1)_{rn} \approx 1/b \cdot (1 - \varepsilon_0^4)$

(8) $q_2 = (q_1 + e_1 \cdot q_1)_{rn} \approx a/b \cdot (1 - \varepsilon_0^4)$

(9) $e_2 = (e_1^2)_{rn} \approx \varepsilon_0^4$

(10) $y_3 = (y_2 + e_2 \cdot y_2)_{rn} \approx 1/b \cdot (1 - \varepsilon_0^8)$

(11) $q_3 = (q_2 + e_2 \cdot q_2)_{rn} \approx a/b \cdot (1 - \varepsilon_0^8)$

(12) $r_0 = (a - b \cdot q_3)_{rn} \approx a \cdot \varepsilon_0^8$

(13) $q_4 = (q_3 + r_0 \cdot y_3)_{rnd} \approx a/b \cdot (1 - \varepsilon_0^{16})$

The first step is a table lookup performed by frcpa, which gives an initial approximation y_0 of 1/b, with known relative error determined by m = 8.886. Steps (3) and (4), (6) and (7), and (9) and (10) represent three iterations that generate increasingly better approximations of 1/b in y_1, y_2, and y_3. Note that step (2) above is exact: y0 has 11 bits (read from a table), and a has 53 bits in the significand, and thus the result of the multiplication has at most 64 bits that fit in the significand. Steps (5), (8), and (11) calculate three increasingly better approximations q_1, q_2 and q_3 of a/b. Evaluating their relative errors and applying other theoretical properties [4], it was shown that $q_4 = (a/b)_{rnd}$ in any IEEE rounding mode *rnd*, and that the status flag settings and exception behavior are IEEE compliant. Assuming that the latency of all floating-point operations is the same, the algorithm takes seven fma latencies: steps (2) and (3) can be executed in parallel, as can steps (4), (5), (6); then (7), (8), (9) and also (10) and (11).

The implementation of this algorithm in assembly language is shown next.

```
(1)  frcpa.s0 f8,p6=f6,f7;;  // y0=1/b in f8
(2)  (p6) fma.s1 f9=f6,f8,f0  // q0=a*y0 in f9
(3)  (p6) fnma.s1 f10=f7,f8,f1;;  // e0=1-b*y0 in f10
(4)  (p6) fma.s1 f8=f10,f8,f8  // y1=y0+e0*y0 in f8
(5)  (p6) fma.s1 f9=f10,f9,f9  // q1=q0+e0*q0 in f9
(6)  (p6) fma.s1 f11=f10,f10,f0;; // e1=e0*e0 in f11
(7)  (p6) fma.s1 f8=f11,f8,f8  // y2=y1+e1*y1 in f8
(8)  (p6) fma.s1 f9=f11,f9,f9  // q2=q1+e1*q1 in f9
(9)  (p6) fma.s1 f10=f11,f11,f0;; // e2=e1*e1 in f10
(10) (p6) fma.s1 f8=f10,f8,f8  // y3=y2+e2*y2 in f8
(11) (p6) fma.d.s1 f9=f10,f9,f9;;//q3=q2+e2*q2in f9
(12) (p6) fnma.d.s1 f6=f7,f9,f6;;  // r0=a-b*q3 in f6
(13) (p6) fma.d.s0 f8=f6,f8,f9;;// q4=q3+r0*y3 in f8
```

Note that the output predicate p6 of instruction (1) (frcpa) predicates all the subsequent instructions. Also, the output register of frcpa (f8) is the same as the output register of the last operation (in step (13)). If the frcpa instruction encounters an exceptional situation such as unmasked division by 0, and an exception handler provides the result of the divide, p6 is cleared and no other instruction from the sequence is executed. The result is still provided where it is expected. Another observation is that the first and last instructions in the sequence use the user status field (sf0), which will reflect exceptions that might occur, while the intermediate computations use status field 1 (sf1, with wre = 1). This implementation behaves like an atomic double precision divide, as prescribed by the IEEE Standard. It sets correctly all the IEEE status flags (plus the denormal status flag), and it signals correctly all the possible floating-point exceptions if unmasked (invalid operation, denormal operand, divide by zero, overflow, underflow, or inexact result).

Floating-Point Remainder

The floating-point divide algorithms are the basis for the implementation of floating-point remainder operations. Their correctness is a direct consequence of the correctness of the floating-point divide algorithms. The remainder is calculated as $r = a - n \cdot b$, where n is the integer closest to the infinitely precise a/b. The problem is that n might require more bits to represent than available in the significand for the format of a and b. The solution is to implement an iterative algorithm, as explained in [5] for FPREM1 (all iterations but the last are called 'incomplete'). The implementation (not shown here) is quite straightforward. The rounding to zero mode for divide can be set in status field sf2 (otherwise identical to the user status field sf0). Status field sf0 will only be used by the first frcpa (which may signal the invalid, divide by zero, or denormal exceptions) and by the last fnma (computing the remainder). The last fnma may also signal the underflow exception.

Software Assistance Conditions for Scalar Floating-Point Divide

The main issue identified in the process of proving the IEEE correctness of the divide algorithms [4] is that there are cases of input operands for *a/b* that can cause overflow, underflow, or loss of precision of an intermediate result. Such operands might prevent the sequence from generating correct results, and they require alternate algorithms implemented in software in order to avoid this. These special situations are identified by the following conditions that define the necessity for IA-64 Architecturally Mandated Software Assistance for the scalar floating-point divide operations:

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

(b) $e_b \ge e_{max} - 2$ (y_i might become tiny)

(c) $e_a - e_b \ge e_{max}$ (q_i might become huge)

(d) $e_a - e_b \le e_{min} + 1$ (q_i might become tiny)

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

where e_a is the (unbiased) exponent of a; e_b is the exponent of b; e_{min} is the minimum value of the exponent in the given format; e_{max} is its maximum possible value; and N is the number of bits in the significand. When any of these conditions is met, frcpa issues a Software Assistance (SWA) request in the form of a floating-point exception instead of providing a reciprocal approximation for *1/b*, and clears its output predicate. An SWA handler provides the result of the floating-point divide, and the rest of the iterative sequence for calculating *a/b* is predicated off. The five conditions above can be represented to show how the two-dimensional space containing pairs (e_a, e_b) is partitioned into regions (Figure 4 of [4]). Alternate software algorithms had to be devised to compute the IEEE correct quotients for pairs of numbers whose exponents fall in regions satisfying any of the five conditions above. Note though that due to the extended internal exponent range (17 bits), the single precision, double precision, and double-extended precision calculations will never require architecturally mandated software assistance. This type of software assistance might be required only for floating-point register file format computations with floating-point numbers having 17-bit exponents.

When an architecturally mandated software assistance request occurs for the divide operation, the result is provided by the IA-64 Floating-Point Emulation Library, which has the role of an SWA handler, as described further.

The parallel reciprocal approximation instruction, fprcpa, does not signal any SWA requests. When any of the five conditions shown above is met, fprcpa merely clears its output predicate, in which case the result of the parallel divide operation has to be computed by alternate algorithms (typically by unpacking the parallel operands, performing two single precision divide operations, and packing the results into a SIMD result).

IA-64 FLOATING-POINT SQUARE ROOT

The IA-64 floating-point square root algorithms are also based on Newton-Raphson or similar iterative computations. If $\sqrt{a}$ needs to be computed and the Newton-Raphson method is used, a number of Newton-Raphson iterations first calculate an approximation of $1/\sqrt{a}$, using the function

$$f(y) = 1/y^2 - a$$

The general iteration step is

$$e_n = (1/2 - 1/2 \cdot a \cdot y_n^2)_{rn}$$

$$y_{n+1} = (y_n + e_n \cdot y_n)_{rn}$$

where the subscript *rn* denotes the IEEE rounding to the nearest mode. The first computation above is rearranged in the real algorithm in order to take advantage of the fma instruction capability.

Once a sufficiently good approximation y of $1/\sqrt{a}$ is determined, $S = a \cdot y$ approximates $\sqrt{a}$. In certain cases, this too might need further refinement.

In order to show that the final result generated by a floating-point square root algorithm represents the correctly rounded value of the infinitely precise result $\sqrt{a}$ in any rounding mode, it was proved (by methods described in [3] and [4]), that the exact value of $\sqrt{a}$ and the final result R* of the algorithm before rounding belong to the same interval of width 1/2 ulp, adjacent to a floating-point number. Then, just as for the divide operation

$$(\sqrt{a})_{rnd} = (R^*)_{rnd}$$

where *rnd* is any IEEE rounding mode.

Floating-point square root algorithms are provided for single precision, double precision, double-extended and full register file format precision, and for SIMD single precision, in two variants. One achieves minimum latency, and one maximizes throughput.

SIMD Floating-Point Square Root Algorithm

We next present as an example the algorithm that allows computing the SIMD single precision square root, and which is optimized for throughput, having a minimum

number of floating-point instructions. The input operand is a pair of single precision numbers (a_1, a_2). The output is the pair ($\sqrt{a_1}$, $\sqrt{a_2}$). All the computational steps are performed in single precision. The algorithm is shown below as a scalar computation. The approximate values shown on the right-hand side are computed assuming no rounding errors occur, and they neglect some high order terms that are very small.

(1) $y_0 = 1/\sqrt{a} \cdot (1 + \varepsilon_0)$, $|\varepsilon_0| \le 2^{-m}$, m=8.831

(2) $h = (1/2 \cdot y_0)_{rn} \approx (1 / (2 \cdot \sqrt{a})) \cdot (1 + \varepsilon_0)$

(3) $t_1 = (a \cdot y_0)_{rn} \approx \sqrt{a} \cdot (1 + \varepsilon_0)$

(4) $t_2 = (1/2 - t_1 \cdot h)_{rn} \approx -\varepsilon_0 - 1/2 \cdot \varepsilon_0^2$

(5) $y_1 = (y_0 + t_2 \cdot y_0)_{rn} \approx 1/\sqrt{a} \cdot (1 - 3/2 \cdot \varepsilon_0^2)$

(6) $S = (a \cdot y_1)_{rn} \approx \sqrt{a} \cdot (1 - 3/2 \cdot \varepsilon_0^2)$

(7) $H = (1/2 \cdot y_1)_{rn} \approx (1 / (2 \cdot \sqrt{a})) \cdot (1 - 3/2 \cdot \varepsilon_0^2)$

(8) $d = (a - S \cdot S)_{rn} \approx a \cdot (3 \cdot \varepsilon_0^2 - 9/4 \cdot \varepsilon_0^4)$

(9) $t_4 = (1/2 - S \cdot H)_{rn} \approx 3/2 \cdot \varepsilon_0^2 - 9/8 \cdot \varepsilon_0^4$

(10) $S_1 = (S + d \cdot H)_{rn} \approx \sqrt{a} \cdot (1 - 27/8 \cdot \varepsilon_0^4)$

(11) $H_1 = (H + t_4 \cdot H)_{rn} \approx (1 / (2 \cdot \sqrt{a})) \cdot (1 - 27/8 \cdot \varepsilon_0^4)$

(12) $d_1 = (a - S_1 \cdot S_1)_{rn} \approx a \cdot (27/4 \cdot \varepsilon_0^4 - 729/64 \cdot \varepsilon_0^8)$

(13) $R = (S_1 + d_1 \cdot H_1)_{rnd} \approx \sqrt{a} \cdot (1 - 2187/128 \cdot \varepsilon_0^8)$

The first step is a table lookup performed by fprsqrta, which gives an initial approximation of ($1/\sqrt{a_1}$,$1/\sqrt{a_2}$) with known relative error determined by m = 8.831. The following steps implement a Newton-Raphson iterative algorithm. Specifically, step (5) improves on the approximation of ($1/\sqrt{a_1}$,$1/\sqrt{a_2}$). Steps (3), (6), (10) and (13) calculate increasingly better approximations of ($\sqrt{a_1}$,$\sqrt{a_2}$). The algorithm was proved correct as outlined in [3] and [4]. The final result (R_1,R_2) equals (($\sqrt{a_1}$)$_{rnd}$,($\sqrt{a_2}$)$_{rnd}$) for any IEEE rounding mode *rnd*, and the status flag settings and exception behavior are IEEE compliant.

The assembly language implementation is as follows (only the floating-point operations are numbered):

```
      movl r3 = 0x3f0000003f000000;; // +1/2,+1/2
      setf.sig f7=r3 // +1/2,+1/2 in f7
(1)   fprsqrta.s0 f8,p6=f6;;          // y0=1/sqrt(a) in f8
(2)   (p6) fpma.s1 f9=f7,f8,f0        // h=1/2*y0 in f9
(3)   (p6) fpma.s1 f10=f6,f8,f0;; // t1=a*y0 in f10
(4)   (p6) fpnma.s1 f9=f10,f9,f7;;// t2=1/2-t1*h in f9
(5)   (p6) fpma.s1 f8=f9,f8,f8;;  // y1=y0+t2*y0 in f8
(6)   (p6) fpma.s1 f9=f6,f8,f0    // S=a*y1 in f9
(7)   (p6) fpma.s1 f8=f7,f8,f0;;  // H =1/2*y1 in f8
(8)   (p6) fpnma.s1 f10=f9,f9,f6 // d=a-S*S in f10
(9)   (p6) fpnma.s1 f7=f9,f8,f7;; // t4=1/2-S*H in f7
(10)  (p6) fpma.s1 f10=f10,f8,f9// S1=S+d*H in f10
(11)  (p6) fpma.s1 f7=f7,f8,f8;;  // H1=H+t4*H in f7
(12)  (p6) fpnma.s1 f9=f10,f10,f6;;// d1=a-S1^2 in f9
(13)  (p6) fpma.s0 f8=f9,f7,f10;;//R=S1+d1*H1 in f8
```

Software Assistance Conditions for Scalar Floating-Point Square Root

Just as for divide, cases of special input operands were identified in the process of proving the IEEE correctness of the square root algorithms [4]. The difference with respect to divide is that only loss of precision of an intermediate result can occur in an iterative algorithm calculating the floating-point square root. Such operands might prevent the sequence from generating correct results, and they require alternate algorithms implemented in software in order to avoid this. These special situations are identified by the following condition that defines the necessity for IA-64 Architecturally Mandated Software Assistance for the scalar floating-point square root operation:

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

where e_a is the (unbiased) exponent of a, e_{min} is the minimum value of the exponent in the given format, and N is the number of bits in the significand. When this condition is met, frsqrta issues a Software Assistance request in the form of a floating-point exception, instead of providing a reciprocal approximation for $1/\sqrt{a}$, and it clears its output predicate. The result of the floating-point square root operation is provided by an SWA handler, and the rest of the iterative sequence for calculating $\sqrt{a}$ is predicated off. Due to the extended internal exponent range (17 bits), the single precision, double precision, and double-extended precision calculations will never require architecturally mandated software assistance. This type of software assistance might be required only for floating-point register file format computations with floating-point numbers having 17-bit exponents.

When an architecturally mandated software assistance request occurs for the square root operation, the result is provided by the IA-64 Floating-Point Emulation Library.

Just as for the parallel divide, the parallel reciprocal square root approximation instruction, fprsqrta, does not signal any SWA requests. When the condition shown

above is met, fprsqrta merely clears its output predicate, in which case the result of the parallel square root operation has to be computed by alternate algorithms (typically by unpacking the parallel operands, performing two single precision square root operations, and packing the results into a SIMD result).

DERIVED OPERATIONS: INTEGER DIVIDE AND REMAINDER

The integer divide and remainder operations are based on floating-point operations. They are not specified in the IEEE Standard [1], but their implementation is so close to that of the floating-point operations mandated by the standard, that it is worthwhile mentioning them here.

A 64-bit integer divide algorithm can be implemented based on the double-extended precision floating-point divide. A 32-bit integer divide algorithm can use the double precision divide. The 16-bit and 8-bit integer divide can use the single precision divide. But the desired computation can be performed in each case by shorter instruction sequences. For example, 24 bits of precision are not needed to implement the 16-bit integer divide.

As examples, the signed and then unsigned 16-bit integer divide algorithms are presented. They are both based on the same core (all four steps below are performed in full register file double-extended precision):

(1) $y_0 = 1/b \cdot (1 + \varepsilon_0)$, $|\varepsilon_0| \le 2^{-m}$, $m=8.886$

(2) $q_0 = (a \cdot y_0)_{rn} = a/b \cdot (1 + \varepsilon_0)$

(3) $e_0 = (1 + 2^{-17} - b \cdot y_0)_{rn} \approx -\varepsilon_0$ (adding 2^{-17} ensures correctness of the final result)

(4) $q_1 = (q_0 + e_0 \cdot q_0)_{rn} \approx a/b \cdot (1 - \varepsilon_0^2)$

The assembly language implementation of the 16-bit signed integer divide algorithm follows. It is assumed that the 16-bit operands are received in r32 and r33, and the result is returned in r8.

```
sxt2 r2=r32    // sign-extend dividend
sxt2 r3=r33;;  // sign-extend divisor
setf.sig f8=r2  // integer dividend in f8
setf.sig f9=r3  // integer divisor in f9
movl r9=0x8000400000000000;;// 1 + 2^-17 in r9
setf.sig f10=r9  // (1 + 2^-17) · 2^63 in f10
fcvt.xf f6=f8   // normal fp dividend in f6
fcvt.xf f7=f9;; // normal fp divisor in f7
fmerge.se f10=f1,f10 // 1 + 2^-17 in f10
(1) frcpa.s1 f8,p6=f6, f7;; // y0 in f8
(2) (p6) fma.s1 f9=f6, f8, f0 // q0 = a * y0 in f9
(3) (p6) fnma.s1 f10=f8,f7,f10;; //e0=(1+2^-17)-b*y0
(4) (p6)  fma.s1 f8=f9,f10,f9;;// q1=q0+e0*q0 in f8
    fcvt.fx.trunc.s1 f8=f8;; // integer quotient in f8
    getf.sig r8=f8;;  // integer quotient in r8
```

The 16-bit unsigned integer divide is similar, but uses the zero-extend instead of the sign-extend instruction from 2 bytes to 8 bytes (zxt2 instead of sxt2), conversion from unsigned integer to floating-point for the operands (fcvt.xuf instead of fcvt.xf), and conversion from floating-point to unsigned integer for the result (fcvt.fxu.trunc instead of fcvt.fx.trunc).

The integer remainder algorithms are implemented as extensions of the corresponding integer divide algorithms. The 16-bit signed integer remainder algorithm is almost identical to the 16-bit signed integer divide, with the last instruction replaced by the following sequence that is needed to calculate $r = a - (a/b) \cdot b$:

```
fcvt.xf f8=f8;; // convert to fp and normalize
fnma.s1 f8=f8, f7, f6;; // r = a - (a/b) · b in f8
fcvt.fx.trunc.s1 f8=f8;;// integer remainder in f8
getf.sig r8=f8;;  // integer remainder in r8
```

EXCEPTIONS AND TRAPS

IA-64 arithmetic floating-point instructions [2] may signal all of the five IEEE-specified exceptions and also the Intel Architecture defined exception for denormal operands. Invalid operation, denormal operand, and divide-by-zero are pre-computation exceptions (floating-point faults). Overflow, underflow, and inexact result are post-computation exceptions (floating-point traps).

In addition to these user visible exceptions, Software Assistance (SWA) faults and traps can be signaled. They do not surface to the user level, and cannot be disabled (masked). The SWA requests are handled by a system SWA handler, the IA-64 Floating-Point Emulation Library.

The status flags in a given status field can be cleared using the fclrf.*sf* instruction. Control bits may be set using the fsetc.*sf amask7, omask7* instruction, which initializes the seven control bits of the specified status field to the value obtained by logically AND-ing the sf0.controls (seven bits) and *amask7,* and logically OR-ing with *omask7.* Alternatively, a 64-bit unsigned

integer value can be moved to or from the FPSR (application register 40): mov ar40 = *r1*, or mov *r1* = ar40.

Exception handlers can be registered, disabled, saved, or restored with software support (from the operating system and/or compiler) as specified by the IEEE Standard.

IA-64 Software Assistance Faults and Traps

The IA-64 architecture allows virtually any floating-point operation to be deferred to software through the mechanism of Software Assistance requests, which are treated as floating-point exceptions, always unmasked, and resolved without reaching a user handler. On Itanium™, the first implementation of the IA-64 architecture, SWA requests may be signaled in three forms:

- IA-64 architecturally mandated SWA faults. These occur for certain combinations of operands of the floating-point divide and square root operations, and only for frcpa and frsqrta (scalar reciprocal approximation instructions).
- Itanium-specific SWA faults. They occur whenever a floating-point instruction has a denormal input. All the arithmetic floating-point instructions on Itanium signal this exception, except for fma.*pc*.*sf f1 = f2*, f1, f0 (fnorm.*pc*.*sf f1=f2*), fms.*pc*.*sf f1 = f2*, f1, f0, and fnma.*pc*.*sf f1 = f2*, f1, f0. They signal Itanium-specific SWA faults only when the input is a canonical double-extended denormal value (i.e., when the input has a biased exponent of 0x00000 and a most significant bit of the non-zero significand equal to 0).
- Itanium-specific SWA traps. They occur whenever a floating-point instruction has a tiny result (smaller in magnitude than the smallest normal floating-point number that can be represented in the destination format). These exceptions only occur for fma, fms, fnma, fpma, fpms, and fpnma.

The IA-64 Floating-Point Emulation Library

When an unmasked floating-point exception occurs, the hardware causes a branch to the interruption vector (Floating-Point Fault or Trap Vector) and then to a low-level OS handler. From here, handling of the floating-point exception is propagated higher in the operating system, and an exception handler is invoked that decides whether to provide a result for the excepting instruction and allow execution of the application to continue.

SWA requests are treated like regular floating-point exceptions, but they are always ‘unmasked’ and handled by an SWA handler represented by the IA-64 Floating-Point Emulation Library. The library is able to calculate the result for any IA-64 arithmetic floating-point instruction. When an SWA fault or trap occurs, it is processed and the result is provided to the operating system kernel. The execution continues in a transparent manner for the user. In addition to satisfying the SWA requests, the SWA handler filters all other unmasked floating-point exceptions that occur, passing them to the operating system kernel that will continue to search for an appropriate user-provided exception handler.

Figure 1 depicts the control flow that occurs when an application running on an IA-64 processor signals an unmasked floating-point exception. The IA-64 Floating-Point Emulation Library is shown as part of the operating system kernel, but this is implementation dependent. If an unmasked floating-point exception other than an SWA fault or trap occurs, a user handler must have already been registered in order to resolve it. The user handler can be called directly by the operating system, receiving ‘raw’ information about the exception, or through an optional IEEE filter (as shown in Figure 1) that processes the information about the exception, thereby allowing a less complex handler to resolve the situation.

An example of an SWA trap is illustrated in Figure 2. The computation generates a result that is tiny and inexact (sufficient to trigger an underflow or an inexact trap if any was unmasked). As traps are masked, an Itanium™-specific SWA trap occurs, propagated from the application code to the floating-point emulation library via the OS kernel trap handler in steps 1 and 2. The result generated by the emulation library is then passed back in steps 3 and 4.

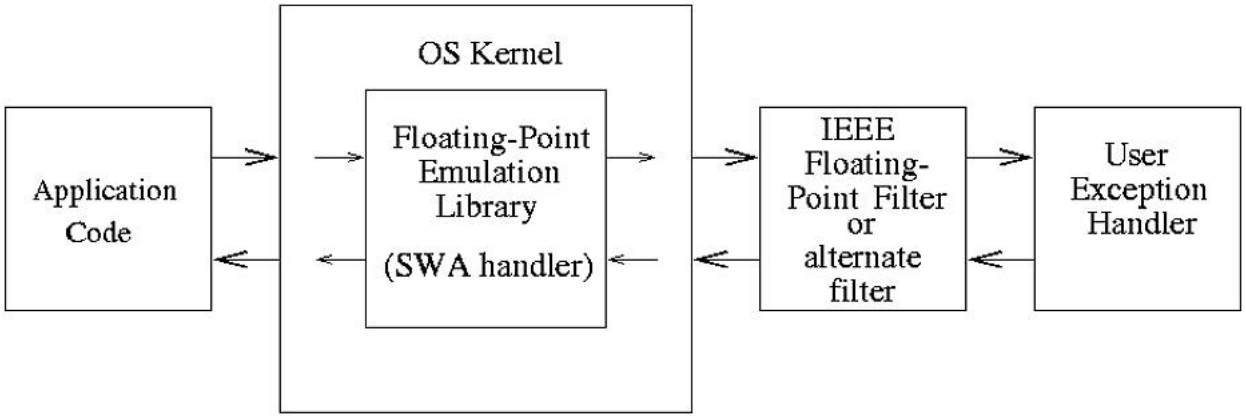

Figure 1: Flow of control for IA-64 floating-point exceptions

DIVIDE EXAMPLE: (1.0...0100 * 2^emin) / (1.0 * 2^4) = 0.0001 * 2^emin
(U traps disabled; SWA trap)

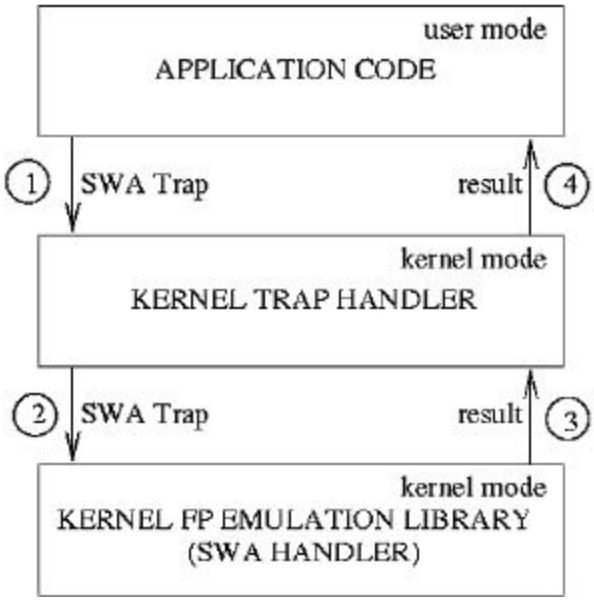

Figure 2: Flow of control for handling an SWA trap signaled by an IA-64 floating-point instruction

FLOATING-POINT EXCEPTION HANDLING

The floating-point exception priority is documented in [2], but for a given implementation of the architecture (Itanium™ in this case), a distinction can be made regarding the source of an exception. This can be signaled by the hardware, or from software, by the IA-64 Floating-Point Emulation Library.

For example, on Itanium, denormal faults are signaled by software (the IA-64 Floating-Point Emulation Library) after they are reported initially by the hardware as Itanium-specific SWA faults. SWA faults that are not converted to denormal faults (because denormal faults are masked) cause the result to be calculated by software. Whether the result of a floating-point instruction is calculated in hardware or in software, it can further signal other floating-point exceptions (traps).

For example, architecturally mandated SWA faults might lead to overflow, underflow, or inexact exceptions signaled from the IA-64 Floating-Point Emulation Library.

Another example is that of the SWA traps, that are always raised from hardware. They have to be resolved in software, but this computation might further lead to inexact exceptions signaled from the IA-64 Floating-Point Emulation Library.

The information that is relevant to a floating-point user exception handler is passed to it through a register file save area, the excepting instruction pointer and opcode, the Floating-Point Status Register, and a set of specialized registers.

The IA-64 IEEE Floating-Point Filter

The floating-point exception handling mechanism of an operating system raises portability issues, as exception handling is almost always implemented using proprietary data structures and procedures. A solution that can be adopted is to implement an IEEE Floating-Point Filter that preprocesses the exception information provided by the operating system kernel before passing it on to the user handler (see Figure 1). The filter, which can be viewed as part of the user handler, helps in the processing of all the IEEE floating-point exceptions (invalid operation, divide-by-zero, overflow, underflow, and inexact result) and also in the processing of the denormal exceptions that are IA specific. The interface between the operating system and the IEEE filter should be almost identical to that of the IA-64 Floating-Point Emulation Library, as they both process exception information. The IEEE filter also accomplishes the correct wrapping of the exponents when overflow or underflow traps are taken, as required by the IEEE Standard [1] (operation deferred to software by the IA-64 architecture).

An important advantage is that the IEEE Floating-Point Filter simplifies greatly the task of the user handler. All the complexities of reading operating system-specific information, decoding operation codes, and reading and writing floating-point or predicate registers are abstracted away by the filter. Also, exceptions generated by parallel (SIMD) instructions will appear to the user handler as originating in scalar instructions. The following two examples illustrate some of these benefits.

The example in Figure 3 illustrates the case of a scalar divide operation that signals an SWA fault, and then an underflow trap (underflow traps are assumed to be unmasked). The SWA fault is signaled by an frcpa instruction that jumpstarts the iterative computation calculating the quotient. The sequence of steps performed in handling the exception is numbered from 1 to 10 in the figure. As the result is provided by the user exception handler for 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 if present). The clear output predicate disables the iterative computation following frcpa, as the result is already in the correct floating-point register (the iterative computation is assumed to be automatically inlined by the compiler).

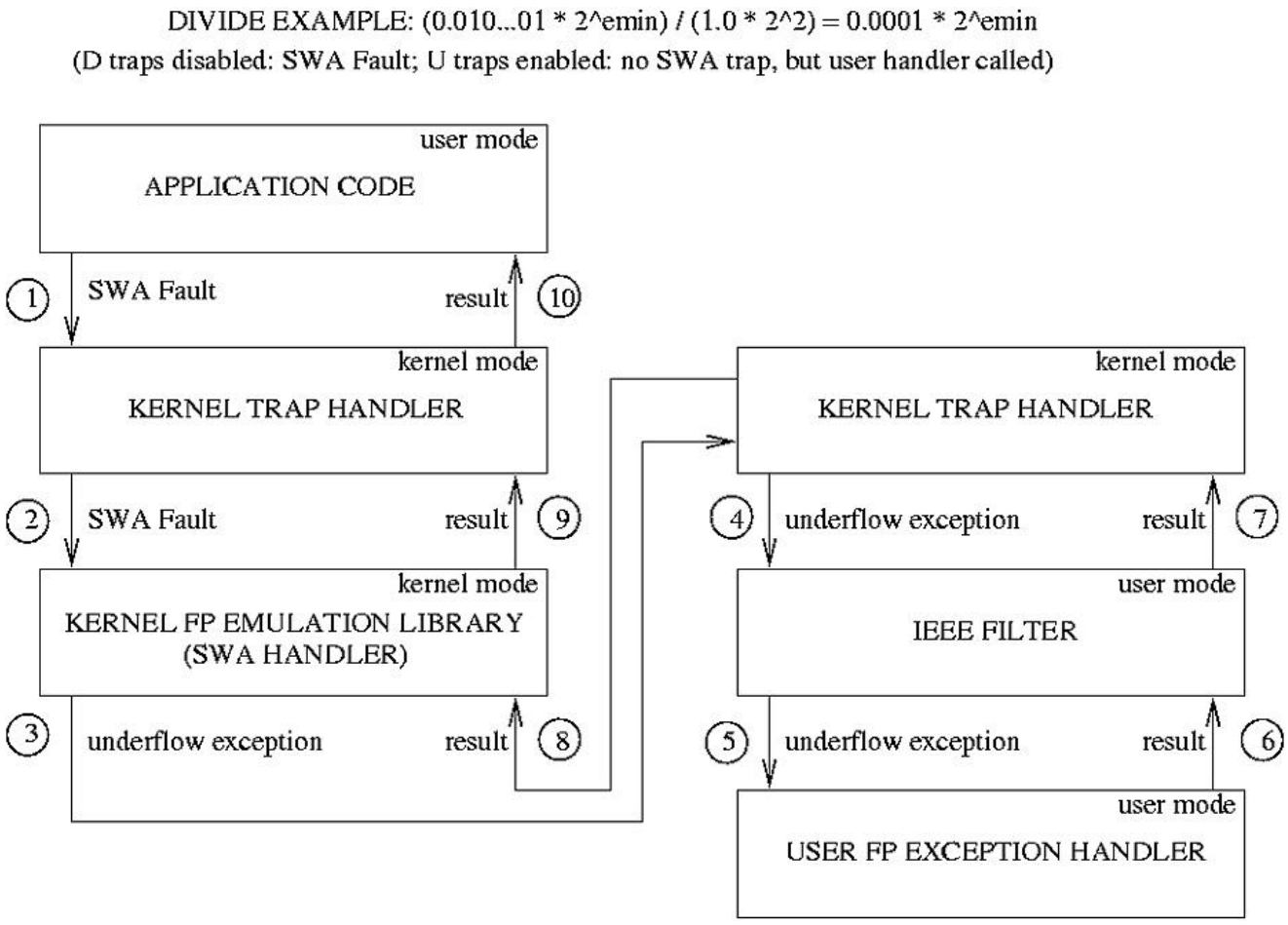

Figure 3: Flow of control for handling an SWA fault signaled by a divide operation, followed by an underflow trap

The example in Figure 4 illustrates the case of a parallel instruction that signals 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 unmasked (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). The sequence of steps executed while handling these exceptions is numbered from 1 to 12 in the figure.

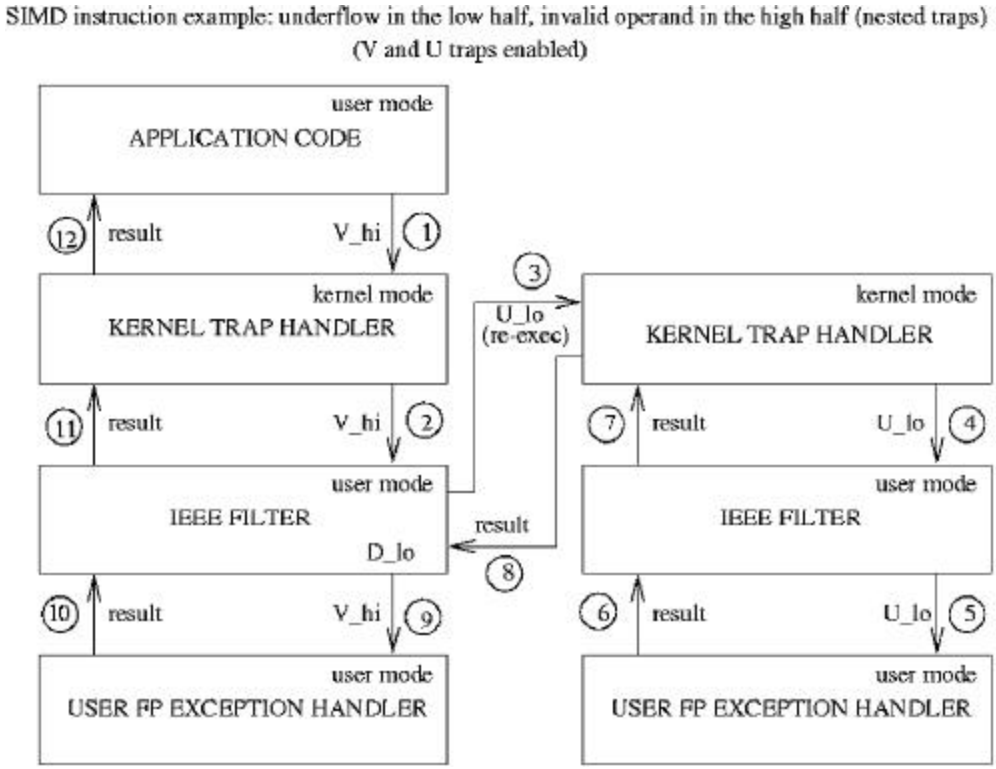

Figure 4: Flow of control for handling an invalid fault in the high half (V high) and an underflow trap in the low half (U low) of a parallel IA-64 instruction

SPECULATION FOR FLOATING-POINT COMPUTATIONS

Control speculation refers to a performance optimization technique where a sequence of instructions is executed before it is known that the dynamic control flow of the program will actually reach the point in the program where the sequence of instructions is needed. Control speculation in floating-point computations on IA-64 processors is possible, as loads to general or floating-point registers have both non-speculative (e.g., ldf, ldfp), and speculative (e.g., ldf.s, ldfp.s) variants. All instructions that write their results to general or floating-point registers are speculative.

A speculative floating-point computation uses status fields sf2 or sf3. The computation is considered to have failed if it signals a floating-point exception that is unmasked in the user status field sf0, or if it sets a status flag that is clear in sf0. This is checked for with the floating-point check flags instruction, fchkf.*sf target25*: the status flags in *sf* are compared with the status flags in sf0. If any flags in *sf* are set and the corresponding traps are enabled, or if any flags are set in *sf* that are not set in sf0, then a branch is taken to *target25*, which should be the address of the recovery code for the failed speculative floating-point computation. The compliance with the IEEE Standard is thus preserved even for speculative chains of computation.

The following example shows original code without control speculation. It is assumed that the contents of f9 are not used at the destination of the branch.

```
        (p6) br.cond some_label ;;
        fma.s0 f9=f8,f7,f6  // Do f9=f8*f7+f6
continue_label:
```

This code sequence can be rewritten using control speculation with sf2 to move the fma ahead of the branch as follows:

```
        fma.s2 f9=f8,f7,f6  // Speculative f9=f8*f7+f6
        // other instructions
        (p6) br.cond some_label ;;
        fchkf.s2 recovery_label   // Check speculation
continue_label:
```

If sf0 and sf2 do not agree, then the recovery code must be executed to cause the actual exception with sf0.

```
recovery_label:
        fma.s0 f9=f8,f7,f6  // Do real f9=f8*f7+f6
        br continue_label
```

CONCLUSION

Compliance with the IEEE Standard for Binary Floating-Point Arithmetic [1] is important for any modern processor. In this paper, we have shown how various facets of the standard are implemented or reflected in the IA-64 architecture, which is fully compliant with the IEEE Standard. In addition, we have highlighted features of the floating-point architecture that allow high-accuracy and high-performance computations, while abiding by the IEEE Standard.

ACKNOWLEDGMENTS

The authors thank Roger Golliver, Gautam Doshi, John Harrison, Shane Story, Ted Kubaska, and Cristina Iordache from Intel Corporation, and Peter Markstein from Hewlett-Packard* Company for their contributions, support, ideas, and/or feedback regarding various parts of this paper.

REFERENCES

[1] ANSI/IEEE Std 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE, New York, 1985.

[2] *IA-64 Application Developer's Architecture Guide*, Intel Corporation, 1999.

[3] Cornea-Hasegan, M., "Proving IEEE Correctness of Iterative Floating-Point Square Root, Divide, and Remainder Algorithms," *Intel Technology Journal*, Q2, 1998 at http://developer.intel.com/technology/itj/q21998.htm

[4] Cornea-Hasegan, M. and Golliver, R., "Correctness Proofs Outline for Newton-Raphson Based Floating-Point Divide and Square Root Algorithms," *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, 1999, IEEE Computer Society, Los Alamitos, CA, pp. 96-105.

[5] *Intel Architecture Software Developer's Manual*, Intel Corporation, 1997.

AUTHORS' BIOGRAPHIES

Marius Cornea-Hasegan is a senior staff software engineer with Intel Corporation in Hillsboro, Oregon. He holds an M.Sc. degree in electrical engineering from the Polytechnic Institute of Cluj, Romania, and a Ph.D. degree in computer science from Purdue University, in West Lafayette, IN. His interests include floating-point architecture and algorithms as parts of a computing system. His e-mail is marius.cornea@intel.com.

Bob Norin joined Intel in 1994. Currently he is manager of the MSL Numerics Group in MPG. Bob received his Ph.D. degree in electrical engineering from Cornell University. He has over 25 years experience developing software for high-performance computers. His technical interests include optimizing the performance of floating-point applications and developing mathematical library software. His e-mail is bob.norin@intel.com.