Software Implementations of Division and Square Root Operations for Intel® Itanium® Processors

Marius Cornea
Intel Corporation

Abstract

Division and square root are basic operations defined by the IEEE Standard 754-1985 for Binary Floating-Point Arithmetic [1], and are implemented in hardware in most modern processors. In recent years however, software implementations of these operations have become competitive. The first IEEE-correct implementations in software of the division and square root operations in a mainstream processor appeared in the 1980s [2]. Since then, several major processor architectures adopted similar solutions for division and square root algorithms, including the Intel® Itanium® Processor Family (IPF). Since the first software algorithms for division and square root were designed and used, improved algorithms were found and complete correctness proofs were carried out. It is maybe possible to improve these algorithms even further.

The present paper gives an overview of the IEEE-correct division and square root algorithms for Itanium processors. As examples, a few algorithms for single precision are presented and properties used in proving their IEEE correctness are stated. Non-IEEE variants, less accurate but faster, of the division, square root and also reciprocal and reciprocal square root operations are discussed. Finally, accuracy and performance numbers are given. The algorithms presented here are inlined by the Intel and other compilers for IPF, whenever division and square root operations are performed.

Introduction

One of the design goals for the Intel® Itanium® architecture, finalized in the late 1990s, was to achieve world-class performance in floating-point computations. For this reason, the floating-point architecture included many novel features for Intel processors: available 82-bit floating-point format (1-bit sign, 17-bit exponent, and 64-bit significand), 128 floating-point registers, rotating registers and other support for software pipelining, multiple status fields, flexible computation modes, and a floating-point multiply-add instruction with only one rounding error in the addition step [3][4][5]. Today it is a known fact that this goal was achieved: presently, 16 of the 17 top positions (1 through 7 and 9 through 17) in the SpecFP 2000 ranking list for speed of single processor systems are held by machines based on Itanium processors.

The floating-point multiply-add instruction `fma` was at the basis of efficient software implementations of the floating-point division and square root operations. An important application of this instruction is in the calculation of exact remainders. For example for a division a/b, where a and b are floating-point numbers, a sequence of increasingly better approximations q_0, q_1, … q_{i-1}, q_i of the quotient a/b can be calculated using the Newton-Raphson or another equivalent method. A final approximation q_i can be obtained that can be rounded correctly as specified by the IEEE Standard 754-1985, provided a correction term (remainder) r_{i-1} can be calculated exactly based on the penultimate approximation q_{i-1}:

$$r_{i-1} = a - b \cdot q_{i-1}$$

If the approximation q_{i-1} is good enough[1], it can be shown that r_{i-1} calculated with an `fma` instruction can be represented always exactly as a floating-point number. The floating-point multiply-add operation, which is not defined by the current IEEE Standard for Binary Floating-Point Arithmetic, is thus essential in calculating IEEE-correct results in software for division and square root in the three most widely used formats defined by the standard: single precision, double precision, and double-extended precision. A brief review of some of the IEEE floating-point formats available in the Itanium architecture is included here for reference.

In general, floating-point numbers are represented as a concatenation of a sign bit, an M-bit exponent field containing a biased exponent, and an N-bit significand field (in this context N = 24, 53, or 64). Mathematically:

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

where $\sigma = \pm 1$, $s \in [1,2)$, $e \in [e_{min}, e_{max}] \cap \mathbf{Z}^2$, $s = 1 + k/2^{N-1}$, $k \in \{0, 1, 2, \ldots, 2^{N-1}-1\}$, $e_{min} = -2^{M-1} + 2$, and $e_{max} = 2^{M-1} - 1$. Let $\mathbf{F}_N$ be the set of floating-point numbers with N-bit significands and unlimited exponent range (no special values such as zeros, infinities, or NaNs[3] are included). The main

---

[1] It suffices for q_{i-1} to be accurate to one *unit-in-the-last-place* (ulp). A unit-in-the-last-place represents the weight of the least significant digit of a floating-point number. For a floating-point number f with N bits in the significand, $f = b_0.b_1b_2\ldots b_{N-1} \cdot 2^e$, the value of one ulp is $1\ ulp(f) = 2^{e-N+1}$.

[2] $\mathbf{Z}$ is the set of integer numbers.

[3] NaN stands for not-a-number. NaNs are symbolic values encoded in floating-point format, used most often to cause or be the result of invalid operations.

parameters of the formats used in the software implementations discussed in the paper are shown in Table 1.

Table 1. Floating-Point Formats Available in the Itanium Architecture (subset)

Format	Precision (N)	Exponent Bits (M)	Exponent Range
Single	24	8	$-126 \le e \le 127$
Double	53	11	$-1022 \le e \le 1023$
Double extended	64	15	$-16382 \le e \le 16383$
Register single	24	17	$-65534 \le e \le 65535$
Register double	53	17	$-65534 \le e \le 65535$
Register	64	17	$-65534 \le e \le 65535$

The division and square root operations discussed here have in general two different implementations available for every format: one that minimizes latency, and one that maximizes throughput. The latency-optimized versions minimize the number of clock cycles elapsed from the beginning of the computation until the result is available. In most cases this is easy to determine, because the majority of floating-point instructions have a latency of 4 clock cycles on the Itanium 2 processor. The throughput-optimized versions minimize the number of clock cycles elapsed between the moments when two consecutive floating-point results are generated. The latter are intended for use in software-pipelined loops, and the resulting throughput depends on the number of functional units available. For example, the throughput-optimized single precision division algorithm uses 7 floating-point instructions, and possibly three memory access instructions. The limiting factor in this case is the number of floating-point instructions. On the Itanium 2 processor, which has two floating-point units available, it will take on average 7/2 = 3.5 clock cycles to generate a result with the throughput-optimized algorithm (but only if the loop is unrolled once, otherwise the throughput will be of 4 clock cycles/result).

IEEE-Correct Floating-Point Division

Division operations that comply with the IEEE Standard 754-1985 have a clearly defined result. In general (exceptions are the cases of underflow or overflow) this is the exact result rounded to the destination precision, using the IEEE rounding mode currently in effect (rounding to nearest, toward zero, toward positive infinity, or toward negative infinity). Division for Itanium processors is implemented based on iterative algorithms, starting with an 11-bit approximation y_0 of the denominator's reciprocal. This value is provided by a special instruction performing a table lookup, `frcpa`, and has a relative error of at most $2^{-8.886}$:

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

Multiplying this value by a, a first approximation of the quotient is obtained and its relative error e_0 can be calculated. The symbol *rn* denotes the IEEE round-to-nearest mode, and *rnd* represents any IEEE rounding mode.

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

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

This approximation can be further improved if the value of q_0 is multiplied by the polynomial $1 - \varepsilon_0 + \varepsilon_0^2 - \ldots + (-\varepsilon_0)^{k-1}$, derived from the identity

$$(1 + \varepsilon_0) \cdot (1 - \varepsilon_0 + \varepsilon_0^2 - \ldots + (-\varepsilon_0)^{k-1}) = 1 - (-\varepsilon_0)^k$$

The result (ignoring for now the rounding errors) will be:

$$q \approx a/b \cdot (1 - (-\varepsilon_0)^k)$$

In addition, an optimal way of calculating the product of this polynomial by $1+\varepsilon_0$ has to be determined for each division algorithm: with the lowest latency for latency-optimized operations, and with the lowest number of floating-point instructions for throughput-optimized operations.

Consider as a first example the latency-optimized single precision division algorithm.

Single precision division, optimized for latency

The following algorithm calculates the single precision value $q'_3 = (a/b)_{rnd}$, where a and b are single precision numbers. All the other intermediate results are 82-bit floating-point register format numbers. The precision used for each step is shown too. An approximate value of the result is also shown, calculated assuming that the rounding errors are negligible.

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

(2) $q_0 = (a \cdot y_0)_{rn} = a/b \cdot (1+\varepsilon_0)$
82-bit floating-point register format

(3) $e_0 = (1 - b \cdot y_0)_{rn} = -\varepsilon_0$
82-bit floating-point register format

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

(5) $e_1 = (e_0 \cdot e_0)_{rn} \approx \varepsilon_0^2$
82-bit floating-point register format

(6) $q_2 = (q_1 + e_1 \cdot q_1)_{rn} \approx a/b \cdot (1-\varepsilon_0^4)$
82-bit floating-point register format
(7) $e_2 = (e_1 \cdot e_1)_{rn} \approx \varepsilon_0^4$
82-bit floating-point register format
(8) $q_3 = (q_2 + e_2 \cdot q_2)_{rn} \approx a/b \cdot (1-\varepsilon_0^8)$
17-bit exponent, 53-bit significand
(9) $q'_3 = (q_3)_{rnd} \approx a/b \cdot (1-\varepsilon_0^8)$
single precision

This shows that the intermediate approximations q_0, q_1, q_2, and q_3 are getting increasingly closer to a/b. The last step is needed to reduce the precision of the result to 24 bits, for the single precision format. As steps (2) and (3), (4) and (5), and (6) and (7) respectively can be executed in parallel, the total latency on the Itanium 2 processor will be of 6 x 4 = 24 clock cycles. In software-pipelined form, this algorithm could generate on average a result every 9/2=4.5 clock cycles. However, an algorithm can be found that has better throughput characteristics.

Single precision division, optimized for throughput

The first idea was to modify the latency-optimized algorithm so that the first five steps generate increasingly better approximations y_1 and y_2 of 1/b, rather than q_1 and q_2. The subsequent steps would be to calculate

$$q_0 = (a \cdot y_2)_{rn}$$

then an exact remainder

$$r_0 = (a - b \cdot q_0)_{rn}$$

in the penultimate step, and the correctly rounded result

$$q_1 = (q_0 + r_0 \cdot y_2)_{rnd} \approx a/b \cdot (1-\varepsilon_0^8)$$

in the last step. This would result in a latency of 7 x 4 = 28 clock cycles, which is worse than that of the previous algorithm, but a better throughput of 8/2 = 4 clock cycles/result. However, an even better algorithm could be found after noticing that $\varepsilon_0^8 < 2^{-71.088}$ leads to a value q_1 before rounding that is more accurate than needed for an IEEE-correct single precision result. The relative error incurred when rounding a real number to single precision is less than 2^{-24}, and about twice as much accuracy should be enough (as shall be seen in the subsection on Correctness Proofs). It suffices for example to calculate $q \approx a/b \cdot (1-\varepsilon_0^6)$ where $\varepsilon_0^6 < 2^{-53.316}$. The best throughput-optimized algorithm is thus:

(1) $y_0 = 1/b \cdot (1+\varepsilon_0)$, $|\varepsilon_0| < 2^{-8.886}$
table lookup
(2) $e_0 = (1 - b \cdot y_0)_{rn} = -\varepsilon_0$
82-bit floating-point register format
(3) $e_1 = (e_0 + e_0 \cdot e_0)_{rn} \approx -\varepsilon_0+\varepsilon_0^2$
82-bit floating-point register format
(4) $y_1 = (y_0 + e_1 \cdot y_0)_{rn} \approx 1/b \cdot (1+\varepsilon_0^3)$
82-bit floating-point register format
(5) $q_1 = (a \cdot y_1)_{rn} \approx a/b \cdot (1+\varepsilon_0^3)$
17-bit exponent, 24-bit significand
(6) $r_1 = (a - b \cdot q_1)_{rn} = -a \cdot \varepsilon_0^3$
82-bit floating-point register format
(7) $q = (q_1 + r_1 \cdot y_1)_{rnd} \approx a/b \cdot (1-\varepsilon_0^6)$
single precision

In software-pipelined form, this algorithm can generate on average one result every 3.5 clock cycles. However, for this the loop would have to be unrolled once, so that it will contain an even number of floating-point instructions. Then two results will be generated on average every 14/2=7 clock cycles.

Similar algorithms were designed for double and double-extended precision division operations. In each case, the optimal sequence was selected that would still afford sufficient accuracy in the final result $q \approx a/b \cdot (1 - (-\varepsilon_0)^k)$ to allow for correct IEEE rounding in all cases. Of all possible sequences, the one that minimized the number of clock cycles was chosen for latency-optimized algorithms, and the one with the lowest number of floating-point instructions for throughput-optimized algorithms. The complete set of IEEE-correct algorithms for the division operation can be found in [6], where source code for all the IPF division algorithms can also be obtained.

Correctness Proofs

Proofs were developed to show that the results of the division algorithms proposed for single, double, and double-extended computations are IEEE-correct for any combination of operands and for any of the four IEEE rounding modes. This included showing also that the floating-point exception status flags are always set correctly, and that unmasked exceptions trap as specified in the IEEE Standard (using the user status field sf0 only in the first and last computation steps and the reserved status field sf1 in the intermediate steps helps ensure correct IEEE exception behavior; note that Itanium processors have four status fields available). To prove that the results are always numerically correct, three properties were used [7]. (The values N of concern in this context are N = 24, N = 53, and N = 64.)

Theorem 1. Let $a, b \in \mathbf{F}_N$, such that $a/b \notin \mathbf{F}_N$, $q^* \in \mathbf{R}$, and $N_1 \in \mathbf{N}$[4], $N_1 \geq 2 \cdot N + 1$.

If q^* is within 1 ulp of a/b in $\mathbf{F}_{N1}$, then

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

Theorem 2. Let $b \in \mathbf{F}_N$, with the restriction that the significand of b is not 1.11…1. Let $y \in \mathbf{F}_N$ be an approximation of 1/b within 1 ulp of 1/b in $\mathbf{F}_N$. Then the computation:

$$e = (1 - b \cdot y)_{rn}$$
$$y' = (y + e \cdot y)_{rn}$$

[4] **R** is the set of real numbers, and **N** is the set of natural numbers.

yields $y' = (1/b)_{rn}$.

Theorem 3. Let $a, b \in \mathbf{F}_N$. If $y \in \mathbf{R}^*$ is within 1/2 ulp of 1/b in $\mathbf{F}_N$, $q \in \mathbf{F}_N$, and $q \cong a/b$ is within 1 ulp of a/b in $\mathbf{F}_N$, then the computation

$$r = (a - b \cdot q)_{rn}$$

$$q' = (q + r \cdot y)_{rnd}$$

yields $q' = (a/b)_{rnd}$.

Theorem 1 was applied in proving correctness of the latency-optimized single precision division algorithm. Relative error evaluations for steps (1) through (8) showed that q_3 is within 1 ulp of a/b in $\mathbf{F}_{49}$. Theorem 1 proves that in step (9), $q'_3 = (a/b)_{rnd}$ (i.e. a/b is correctly rounded, as specified by the IEEE Standard).

Theorem 3 was applied in proving correctness of the throughput-optimized single precision division algorithm. First it was shown that y_1 is within 1/2 ulp of 1/b in $\mathbf{F}_{24}$ and q_1 is within 1 ulp of a/b in $\mathbf{F}_{24}$. Theorem 3 states that steps (6) and (7):

$$(6)\ r_1 = (a - b \cdot q_1)_{rn}$$

$$(7)\ q = (q_1 + r_1 \cdot y_1)_{rnd}$$

yield $q = (a/b)_{rnd}$.

Theorem 2 was needed only for the double-extended division algorithms, where the operands and the result have the same precision as the intermediate calculations. This makes it more difficult to rely just on simple relative error evaluations to show for example that y in the last step is within 1/2 ulp of 1/b as required by Theorem 3, but Theorem 2 makes this possible. One special case had to be treated separately, when the significand of b is 1.11…1 (but for this case it could be checked directly that $y' = (1/b)_{rn}$).

The mathematical proofs of correctness were checked further using an automatic proof checker written in HOL [8].

Non-IEEE Floating-Point Division

There are applications where strict IEEE accuracy for floating-point computations may not be required, and instead faster basic operations would be of more benefit. To cover such needs, non-IEEE floating-point division algorithms were derived from the IEEE-correct versions, with relative errors not exceeding 1 ulp (the IEEE-correct operations have relative errors of at most 0.5 ulp). Non-IEEE algorithms were designed also for reciprocal operations, which are not defined by the IEEE Standard. The division operations performed by the non-IEEE algorithms are thus slightly less accurate, but faster than their equivalent IEEE-correct algorithms.

For example, the non-IEEE single precision division algorithm is:

(1) $y_0 = 1/b \cdot (1+\varepsilon_0),\ |\varepsilon_0| < 2^{-8.886}$
table lookup
(2) $q_0 = (a \cdot y_0)_{rn} = a/b \cdot (1+\varepsilon_0)$
82-bit floating-point register format
(3) $e_0 = (1 - b \cdot y_0)_{rn} = -\varepsilon_0$
82-bit floating-point register format
(4) $e_1 = (e_0 + e_0 \cdot e_0)_{rn} \approx -\varepsilon_0 + \varepsilon_0^2$
82-bit floating-point register format
(5) $q_1 = (q_0 + e_1 \cdot q_0)_{rnd} \approx a/b \cdot (1+\varepsilon_0^3)$
single precision

The same algorithm can be used both in latency-optimized as well as throughput-optimized code.

Only a limited correctness proof is required in this case. The maximum relative error of the result has to be determined, and it has to be proved that overflow and underflow conditions occur reasonably close to those for the IEEE-correct algorithm. The exception status flag for precision is not checked in this case.

The algorithm for calculating the non-IEEE single precision reciprocal is even simpler:

(1) $y_0 = 1/b \cdot (1+\varepsilon_0),\ |\varepsilon_0| < 2^{-8.886}$
table lookup
(2) $e_0 = (1 - b \cdot y_0)_{rn} = -\varepsilon_0$
82-bit floating-point register format
(3) $e_1 = (e_0 \cdot e_0 + e_0)_{rn} \approx -\varepsilon_0 + \varepsilon_0^2$
82-bit floating-point register format
(4) $y_1 = (y_0 + e_1 \cdot y_0)_{rnd} \approx 1/b \cdot (1+\varepsilon_0^3)$
single precision

Similar algorithms for non-IEEE double precision division and reciprocal are given in [9], together with source code.

Latency, Throughput, and Accuracy for Division and Reciprocal Operations

Latency and throughput values for the single, double, and double-extended IEEE-correct and non-IEEE division and reciprocal operations on the Itanium 2 processor are given in Tables 2a and 2b.

Table 2a. Latency, Throughput, and Accuracy for IPF IEEE Division and Reciprocal Operations

Operation	Latency (clock cycles)	Throughput (clock cycles/ result)	Accuracy (ulps)
Single Precision Division	24	3.5	0.50
Double Precision Division	28	5.0	0.50
Double-Extended Precision Division	32	7.0	0.50
Single Precision Reciprocal	24	3.5	0.50
Double Precision Reciprocal	28	5.0	0.50

Theoretical error bounds for the non-IEEE operations are given in Table 2b. These are guaranteed upper bounds, but might not be reached in some cases. For this reason, maximum errors observed in testing are also included in the table.

Table 2b. Latency, Throughput, and Accuracy for IPF Non-IEEE Division and Reciprocal Operations

Operation	Latency (clock cycles)	Throughput (clock cycles/ result)	Theoretical Accuracy (ulps)	Observed Accuracy (ulps)
Single Precision Division	16	2.5	0.6585	0.6524
Double Precision Division	20	4.0	0.5018	0.5010
Double-Extended Precision Division	NA	NA	NA	NA
Single Precision Reciprocal	16	2.0	0.6585	0.6487
Double Precision Reciprocal	20	3.5	0.5010	0.5003

IEEE-Correct Floating-Point Square Root

Square root operations that comply with the IEEE Standard 754-1985 return the exact result rounded to the destination precision, using the IEEE rounding mode currently in effect. The square root operation for Itanium processors is implemented based on iterative algorithms as well, starting with an 11-bit approximation y_0 of the reciprocal square root. This value is provided by a special instruction performing a table lookup, `frsqrta`, and has a relative error of at most $2^{-8.831}$:

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

Multiplying by a, a first approximation of the square root is obtained and its relative error δ can be calculated:

$$S_0 = (a \cdot y_0)_{rn} = \sqrt{a} \cdot (1+\varepsilon_0)$$
$$\delta = 1/2 \cdot (1 - S_0 \cdot y_0)_{rn} = -\varepsilon_0 - 1/2 \cdot \varepsilon_0^2$$

Note that $1 - 2\,\delta = (1+\varepsilon_0)^2$. Just as for division, the approximation S_0 can be improved further if it is multiplied by $1 - \varepsilon_0 + \varepsilon_0^2 - \ldots + (-\varepsilon_0)^{k-1}$. The result (ignoring the rounding errors) will be:

$$S \approx \sqrt{a} \cdot (1 - (-\varepsilon_0)^k)$$

A complication in this case is the fact that the relative error δ calculated for S_0 is not equal to $-\varepsilon_0$, as it was for the division operation. In order to use the identity:

$$(1 + \varepsilon_0) \cdot (1 - \varepsilon_0 + \varepsilon_0^2 - \ldots + (-\varepsilon_0)^{k-1}) = 1 - (-\varepsilon_0)^k$$

a polynomial in δ has to be found, that approximates sufficiently well

$$1 - \varepsilon_0 + \varepsilon_0^2 - \ldots + (-\varepsilon_0)^{k-1} + \ldots = 1/(1 + \varepsilon_0)$$

For this, the value of ε_0 is calculated from $\delta = -\varepsilon_0 - 1/2 \cdot \varepsilon_0^2$:

$$\varepsilon_0 = -1 + \sqrt{(1 - 2 \cdot \delta)}$$

The McLaurin series expansion for $1/(1 + \varepsilon_0) = 1/\sqrt{(1 - 2 \cdot \delta)}$ is:

$$1 - \varepsilon_0 + \varepsilon_0^2 - \varepsilon_0^3 + \varepsilon_0^4 - \ldots = 1 + \delta + 3/2 \cdot \delta^2 + 5/2 \cdot \delta^3 + 35/8 \cdot \delta^4 + 63/8 \cdot \delta^5 + 231/16 \cdot \delta^6 + \ldots$$

An approximation of the expansion in δ consisting of a few terms can be used to design an algorithm converging toward the square root value. The coefficients of some of the higher degree terms in this approximation can even be modified to make the calculation easier. Because of the truncation, the result will be

$$S \approx \sqrt{a} \cdot (1 + O(\varepsilon_0^k))$$

instead of

$$S \approx \sqrt{a} \cdot (1 - (-\varepsilon_0)^k)$$

where $O(\varepsilon_0^k)$ denotes a polynomial containing terms in ε_0 of degree k or higher.

In addition, an optimal way of calculating the product of this polynomial by $1+\varepsilon_0$ has to be determined for each square root algorithm: with the lowest latency for latency-optimized operations, and with the lowest number of floating-point instructions for throughput-optimized operations.

Consider as a first example the latency-optimized single precision square root algorithm.

Single precision square root, optimized for latency

The following algorithm calculates $S = (\sqrt{a})_{rnd}$ in single precision, where a is a single precision number. An approximate value of the result is also shown, calculated assuming that the rounding errors are negligible. The approximation is expressed in terms of ε_0 and/or δ, as convenient:

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

(2) $H_0 = (0.5 \cdot y_0)_{rn} = 1/(2 \cdot \sqrt{a}) \cdot (1+\varepsilon_0)$
82-bit floating-point register format

(3) $S_0 = (a \cdot y_0)_{rn} = \sqrt{a} \cdot (1+\varepsilon_0)$
82-bit floating-point register format

(4) $d = (0.5 - S_0 \cdot H_0)_{rn} = -\varepsilon_0 + 1/2 \cdot \varepsilon_0^2 = \delta$
82-bit floating-point register format

(5) $e = (1 + 1.5 \cdot d)_{rn} \approx 1 + 3/2 \cdot \delta$
82-bit floating-point register format

(6) $T_0 = (d \cdot S_0)_{rn} \approx = \sqrt{a} \cdot \delta \cdot (1+\varepsilon_0)$
82-bit floating-point register format

(7) $G_0 = (d \cdot H_0)_{rn} \approx 1/(2 \cdot \sqrt{a}) \cdot \delta \cdot (1+\varepsilon_0)$
82-bit floating-point register format

(8) $S_1 = (S_0 + e \cdot T_0)_{rn} \approx$

$\sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \cdot \delta^2)$
17-bit exponent, 24-bit significand

(9) $H_1 = (H_0 + e \cdot G_0)_{rn} \approx$
$1/(2\sqrt{a}) \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \cdot \delta^2)$
82-bit floating-point register format

(10) $d_1 = (a - S_1 \cdot S_1)_{rn} \approx$
$a \cdot (5 \cdot \delta^3 + 15/4 \cdot \delta^4 + 9/2 \cdot \delta^5)$
82-bit floating-point register format

(11) $S = (S_1 + d_1 \cdot H_1)_{rnd} \approx$
$\sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \cdot \delta^2 + 5/2 \cdot \delta^3 +$
$35/8 \cdot \delta^4 + 63/8 \cdot \delta^5 + 81/16 \cdot \delta^6 + 27/8 \cdot \delta^7)$
$= \sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 - \varepsilon_0 + \varepsilon_0^2 - \varepsilon_0^3 + \varepsilon_0^4 - \varepsilon_0^5 +$
$O(\varepsilon_0^6)) = \sqrt{a} \cdot (1 + O(\varepsilon_0^6))$
single precision

This shows that approximations S_0, S_1, and S are getting increasingly closer to $\sqrt{a}$. As steps (2) and (3), then (5), (6) and (7), and also (8) and (9) can be executed in parallel, the total latency on the Itanium 2 processor will be 7 x 4 = 28 clock cycles. In software-pipelined form, this algorithm could generate a result every 11/2=5.5 clock cycles. However, an algorithm can be found that has better throughput characteristics.

Single precision square root, optimized for throughput

The following algorithm for the calculation of the single precision square root has the least number of instructions possible, and therefore is best suited for software-pipelined loops. It calculates $S = (\sqrt{a})_{rnd}$ in single precision, where a is a single precision number:

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

(2) $H_0 = (0.5 \cdot y_0)_{rn} = 1/(2 \cdot \sqrt{a}) \cdot (1+\varepsilon_0)$
82-bit floating-point register format

(3) $S_0 = (a \cdot y_0)_{rn} = \sqrt{a} \cdot (1+\varepsilon_0)$
82-bit floating-point register format

(4) $d = (0.5 - S_0 \cdot H_0)_{rn} = -\varepsilon_0 + 1/2 \ \varepsilon_0^2 = \delta$
82-bit floating-point register format

(5) $d' = (d + 0.5 * d)_{rn} \approx 3/2 \cdot \delta$
82-bit floating-point register format

(6) $e = (d + d * d')_{rn} \approx \delta + 3/2 \cdot \delta^2$
82-bit floating-point register format

(7) $S_1 = (S_0 + e * S_0)_{rn} \approx$
$\sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \ \delta^2)$
17-bit exponent, 24-bit significand

(8) $H_1 = (H_0 + e * H_0)_{rn} \approx$
$1/(2\sqrt{a}) \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \cdot \delta^2)$
82-bit floating-point register format

(9) $d_1 = (a - S_1 \cdot S_1)_{rn} \approx$
$a \cdot (5 \cdot \delta^3 + 15/4 \cdot \delta^4 + 9/2 \cdot \delta^5)$
82-bit floating-point register format

(10) $S = (S_1 + d_1 \cdot H_1)_{rnd} \approx$
$\sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \cdot \delta^2 + 5/2 \cdot \delta^3 +$
$35/8 \cdot \delta^4 + 63/8 \cdot \delta^5 + 81/16 \cdot \delta^6 + 27/8 \cdot \delta^7)$
$= \sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 - \varepsilon_0 + \varepsilon_0^2 - \varepsilon_0^3 + \varepsilon_0^4 - \varepsilon_0^5 +$
$O(\varepsilon_0^6)) = \sqrt{a} \cdot (1 + O(\varepsilon_0^6))$
single precision

Only steps (2) and (3), and then (7) and (8) can be executed in parallel, so the latency of 8 x 4 = 32 clock cycles is worse than that of the previous algorithm. However, its throughput of 10/2 = 5 clock cycles/result is better. It can be noticed that even though the throughput-optimized sequence differs slightly from the latency-optimized one, they both calculate practically the same result. The rounding errors might accumulate differently, but the end result was shown to be IEEE-correct in both cases.

Similar algorithms were designed for double and double-extended precision square root operations. In each instance, the optimal sequence was selected that would still afford sufficient accuracy in the final step to allow for correct IEEE rounding in all cases. Similar to the single precision square root, an optimal sequence was determined in each case, that would lead to a result in the form $S \approx \sqrt{a} \cdot (1-(-\varepsilon_0)^k)$. Of all possible sequences, the one that minimizes the number of clock cycles was chosen for latency-optimized algorithms, and the one with the lowest number of floating-point instructions for throughput-optimized algorithms. The complete set of IEEE-correct algorithms for the square root operation can be found in [6], where source code for all the IPF square root algorithms can also be obtained.

Correctness Proofs

Proofs were developed to show that the results of the square root algorithms proposed for single, double, and double-extended computations are IEEE-correct for any combination of operands and for any of the four IEEE rounding modes. This included showing that the floating-point exception status flags are always set correctly, and that unmasked exceptions trap as specified in the IEEE Standard (just as for division, using the user status field sf0 only in the first and last computation steps and the reserved status field sf1 in the intermediate steps helps ensure correct IEEE exception behavior). To prove that the results are always numerically correct, two properties were used [7]:

Theorem 4. Let $a \in \mathbf{F}_N$ and ulp ($\sqrt{a}$) one ulp of $\sqrt{a}$ in $\mathbf{F}_N$. If $\sqrt{a} \notin \mathbf{F}_N$, then for any $f \in \mathbf{F}_N$, the distance between $\sqrt{a}$ and f satisfies

$$| \sqrt{a} - f | > 2^{-N-1} \cdot \text{ulp} (\sqrt{a})$$

Theorem 5. Let $a \in \mathbf{F}_N$ and ulp ($\sqrt{a}$) one ulp of $\sqrt{a}$ in $\mathbf{F}_N$. For any $m \in \mathbf{F}_{N+1} - \mathbf{F}_N$ (midpoint between two consecutive floating-point numbers in $\mathbf{F}_N$), the distance between $\sqrt{a}$ and m satisfies

$$| \sqrt{a} - m | > 2^{-N-3} \cdot \text{ulp} (\sqrt{a})$$

These two properties show that if $\sqrt{a}$ cannot be represented as a floating-point number with an N-bit significand (which is the non-trivial case to check),

then there are exclusion zones of known minimal width around any floating-point number, as well as around any midpoint between two consecutive floating-point numbers, within which $\sqrt{a}$ cannot exist. The minimum distance between $\sqrt{a}$ and f or $\sqrt{a}$ and m can be determined analytically, as well as values of the argument a for which $\sqrt{a}$ is close to points f or m [7] (few points are 'really close' and they can be determined relatively easily). Excluding a number of these points a has the effect of increasing the widths of the exclusion zones. This can be done until the exclusion zones are more than twice wider than the maximum error of the result that approximates $\sqrt{a}$. It means that the exact result and the approximation computed by the algorithm are on the same side of any floating-point number or any midpoint, and therefore they will both round to the same floating-point value. For the relatively few cases of arguments inside the increased exclusion zones, verification was carried out directly.

Similar to the case of the division algorithms, the mathematical proofs of correctness were checked further using an automatic proof checker written in HOL [10].

Non-IEEE Floating-Point Square Root

Just as for division, non-IEEE floating-point square root algorithms that are less accurate but more efficient were derived from the IEEE-correct versions, with relative errors not exceeding 1 ulp (the IEEE-correct operations have relative errors of at most 0.5 ulp). Non-IEEE algorithms were designed also for reciprocal square root operations, which are not defined by the IEEE Standard. For example, the non-IEEE single precision square root algorithm is:

(1) $y_0 = 1/\sqrt{a} \cdot (1+\varepsilon_0),\ |\varepsilon_0| < 2^{-8.831}$
table lookup
(2) $S_0 = (a \cdot y_0)_{rn} = \sqrt{a} \cdot (1+\varepsilon_0)$
82-bit floating-point register format
(3) $d = (1 - S_0 \cdot y_0)_{rn} = -2 \cdot \varepsilon_0 + \varepsilon_0^2 = 2 \cdot \delta$
82-bit floating-point register format
(4) $e = (0.5 + 0.375 \cdot d)_{rn} \approx 1/2 + 3/4 \cdot \delta$
82-bit floating-point register format
(5) $T_0 = (d \cdot S_0)_{rn} \approx 2 \cdot \delta \cdot \sqrt{a} \cdot (1+\varepsilon_0)$
82-bit floating-point register format
(6) $S = (S_0 + e \cdot T_0)_{rnd} \approx$
$\sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \cdot \delta^2) =$
$\sqrt{a} \cdot (1 + 5/2 \cdot \varepsilon_0^3 + 15/8 \cdot \varepsilon_0^4 + 3/8 \cdot \varepsilon_0^5) =$
$\sqrt{a} \cdot (1 + O(\varepsilon_0^3))$
single precision

The same algorithm can be used both in latency-optimized as well as throughput-optimized code.

Only a limited correctness proof is required. The maximum relative error of the result has to be determined, but the exception status flag for precision is not checked in this case.

The algorithm for calculating the non-IEEE single precision reciprocal square root is:

(1) $y_0 = 1/\sqrt{a} \cdot (1+\varepsilon_0),\ |\varepsilon_0| < 2^{-8.831}$
table lookup
(2) $S_0 = (a \cdot y_0)_{rn} = \sqrt{a} \cdot (1+\varepsilon_0)$
82-bit floating-point register format
(3) $d = (1 - S_0 \cdot y_0)_{rn} = -2 \cdot \varepsilon_0 + \varepsilon_0^2 = 2 \cdot \delta$
82-bit floating-point register format
(4) $e = (0.5 + 0.375 \cdot d)_{rn} \approx 1/2 + 3/4 \cdot \delta$
82-bit floating-point register format
(5) $T_0 = (d \cdot y_0)_{rn} \approx 2 \cdot \delta \cdot 1/\sqrt{a} \cdot (1+\varepsilon_0)$
82-bit floating-point register format
(6) $S = (y_0 + e \cdot T_0)_{rnd} \approx$
$1/\sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \cdot \delta^2) =$
$1/\sqrt{a} \cdot (1 + 5/2 \cdot \varepsilon_0^3 + 15/8 \cdot \varepsilon_0^4 + 3/8 \cdot \varepsilon_0^5) =$
$1/\sqrt{a} \cdot (1 + O(\varepsilon_0^3))$
single precision

Similar algorithms for non-IEEE double precision square root and reciprocal square root are given in [9], together with source code.

Latency, Throughput, and Accuracy for Square Root and Reciprocal Square Root Operations

Latency and throughput numbers for the single, double, and double-extended IEEE-correct and non-IEEE square root and reciprocal square root operations on the Itanium 2 processor are given in Tables 3a and 3b.

Table 3a. Latency, Throughput, and Accuracy for IPF IEEE Square Root and Reciprocal Square Root Operations

Operation	Latency (clock cycles)	Throughput (clock cycles/ result)	Accuracy (ulps)
Single Precision Square Root	28	5.0	0.50
Double Prec. Square Root	36	6.5	0.50
Double-Ext. Precision Square Root	40	7.5	0.50
Single Precision Reciprocal Square Root	52 (sqrt+div)	8.5 (sqrt+div)	1.0 (sqrt+div)
Double Prec. Rec. Sq. Root	64 (sqrt+div)	11.5 (sqrt+div)	1.0 (sqrt+div)

Theoretical error bounds and maximum errors observed in testing are both included in Table 3b.

Table 3b. Latency, Throughput, and Accuracy for IPF Non-IEEE Square Root and Reciprocal Square Root Operations

Operation	Latency (clock cycles)	Throughput (clock cycles/ result)	Theoretical Accuracy (ulps)	Observed Accuracy (ulps)
Single Precision Square Root	20	3.0	0.9449	0.8194
Double Precision Square Root	32	5.5	0.5001	0.5000
Double-Extended Precision Square Root	NA	NA	NA	NA
Single Precision Reciprocal Square Root	20	3.5	0.9449	0.8860
Double Prec. Rec. Square Root	32	6.5	0.5031	0.5007

Conclusion

Several factors determined the implementation in software of the division and square root operations for Itanium processors. A first consideration was flexibility, as alternative algorithms can be easily substituted for the original ones, should this be needed. One example is using non-IEEE algorithms instead of IEEE-correct ones when accuracy can be relaxed for the benefit of better performance. Second, the software implementations of these operations inherit the high degree of pipelining in the basic floating-point multiply-add operations, leading to high-throughput algorithms. Third, as in typical applications division and square root are not extremely frequent, it may be that the die area on the chip that would be dedicated to hardware implementations of these operations could be better used for some other purpose.

The Itanium floating-point architecture was designed so that its high performance, accuracy, and flexibility characteristics make it ideal for technical and scientific computing. The present paper showed how software implementations of division and square root operations based on the fused floating-point multiply-add instruction support this goal. The principles used in designing algorithms for these operations were presented together with examples. Correctness proofs were outlined and underlying properties were stated. Non-IEEE algorithms were described in contrast with those that implement the division and square root operations mandated by the IEEE Standard 754-1985. Finally, performance numbers were presented.

Acknowledgements

Many people contributed to the development and verification of the algorithms described in this paper. Peter Markstein developed the original division and square root algorithms for software implementation. The author helped improve some algorithms, derived non-IEEE versions, performed mathematical correctness proofs, and determined special cases where the software algorithms are limited in their capabilities (not encountered in compiled code). John Harrison improved several algorithms (especially square root), and carried out automated proofs of correctness based on the existing mathematical proofs. Roger Golliver, Bob Norin, Cristina Iordache, and Shane Story reviewed the mathematical proofs of correctness or various related documents. Special thanks are due to Bob Norin for reviewing this paper.

References

[1] ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, NY, 1985

[2] Markstein P., Computation of Elementary functions on the IBM RISC System/6000 Processor, *IBM Journal*, 1990

[3] Intel(R) Itanium(TM) Architecture Software Developer's Manual, Vol 1-4, Intel Corp., 2003

[4] Cornea, M., Harrison, J., Tang, P, Scientific and Engineering Computation on Itanium™ Processors, Intel Press, 2002

[5] Cornea, M., Harrison, J., Tang, P, Intel Itanium™ Floating-Point Architecture, WCAE 2003, San Diego

[6] Divide, Square Root, and Remainder Algorithms for the Itanium Architecture, Intel Corporation, Dec. 2003,http://www.intel.com/software/products/opensource/libraries/numnote2.htm

[7] Cornea-Hasegan, M. and Golliver, R., Markstein, P. Correctness Proofs Outline for Newton-Raphson Based Floating-Point Divide and Square Root Algorithms, Proceedings of the 14th IEEE Symposium on Computer Arithmetic, Adelaide, 1999

[8] Harrison, J. Formal Verification of IA-64 Division Algorithms, Proceedings of the 13h International Conference TPHOLs 2000, Springer-Verlag, pp 234-251

[9] Non-IEEE Division, Square Root, Reciprocal, and Reciprocal Square Root Algorithms for the Intel Itanium Architecture, Intel Corporation, Dec. 2003, http://www.intel.com/software/products/opensource/libraries/numnote3.htm

[10] Harrison, J. Formal Verification of Square Root Algorithms, Formal Methods in System Design, Vol. 22, 2003, pp 143-153