Floating-Point Division

As noted in Chapter 3, there are no instructions for floating-point (or integer) division in the Itanium architecture, but only a reciprocal approximation instruction intended to support these operations. The \texttt{frcpa} instruction, when applied to floating-point numbers $a$ and $b$, 1. Except when using IA-32 emulation mode. Even then, the implementation in all current Itanium processors is by microcode using algorithms similar to the ones presented here, not by dedicated hardware.
Floating-Point Division

normally provides merely an initial approximation to the reciprocal \( \frac{1}{b} \), and software must refine this to the correct quotient. For present purposes, it will be assumed that a correct quotient is one correctly rounded according to the stipulations of the IEEE Standard 754-1985 for binary floating-point arithmetic, including generation of appropriate flags or exceptions [2]. It may be that in certain special situations, for example in graphics applications, such a rigorous approach is not necessary, and it would suffice to have a quotient correct to only the last few bits, or even less. It is relatively easy to adapt the present algorithms for such purposes: they often work by successive improvement of an approximation, and one can more quickly obtain a less accurate result simply by omitting later stages. The algorithms will first be presented in a more mathematically abstract way, ignoring rounding errors except where otherwise indicated. Later the detailed effect of rounding errors and the actual implementation will be considered. Suppose then that \( \text{frcpa} \) gives a reciprocal approximation

\[
y_0 = \frac{1}{b}(1 + \epsilon_0)
\]

where according to the architecture definition, \( |\epsilon_0| \leq 2^{-8.886} \) and \( y_0 \) has at most 11 significant bits. Simply multiplying the initial \( y_0 \) by \( a \)

\[
q_0 = ay_0
\]

gives an approximation to \( a/b \) with a relative error \( \eta_0 = \epsilon_0 \) slightly larger in the worst case owing to the rounding error committed in calculating \( q_0 \):

\[
q_0 = \frac{a}{b}(1 + \eta_0)
\]

**Refining Reciprocals**

In order to improve a reciprocal approximation \( y \) such as the initial estimate \( y_0 \), it is useful to obtain its relative error \( \epsilon \) as a floating-point number. Thanks to the \( \text{fma} \), this can be done rather accurately using a single \( \text{fnma} \) operation:

\[
d = 1 - by
\]

Ignoring rounding errors as usual:

\[
d = (1 - by) = 1 - b\frac{1}{b}(1 + \epsilon) = 1 - (1 + \epsilon) = -\epsilon
\]
In fact, if $d$ is calculated in double-extended precision and the input is in single or double precision, no rounding error is committed at all and $d = -\varepsilon$ exactly anyway for the initial $y_0$, since the initial approximations returned by $\text{frcpa}$ always have at most 11 significant bits. In any case, the accuracy of the initial $d$ is not usually significant for these algorithms since later refinements use a more precise estimate of the current error. However, the use of the $\text{fma}$ is crucial here; if instead of the $\text{fma}$ the calculation of $d$ were done with a separate multiplication and addition, the final $d$ could be very inaccurate in relative terms.

Now $d$ can be used to ‘correct’ the approximation $y$. Ideally $y$ would be divided by $1 + \varepsilon$ to obtain $\frac{1}{b}$ exactly. Of course, division cannot be used because that is what the algorithms are intended to implement, but since

$$\frac{1}{1 + \varepsilon} = 1 - \varepsilon + \varepsilon^2 - \varepsilon^3 + \ldots$$

$$= 1 + d + d^2 + d^3 + \ldots$$

a fairly good correction to $y$ can be obtained using multiplication by a suitable truncation of this infinite series. If the series is cut off at some point $d^n$ then, ignoring rounding errors as usual:

$$y(1 + d + d^2 + \ldots + d^n)$$

$$= \frac{1}{b}(1 - d)(1 + d + d^2 + \ldots + d^n)$$

$$= \frac{1}{b}(1 - d^{n+1})$$

Since $|d| \leq 2^{-8.886}$, this approximation can be made as accurate as need be by choosing a sufficiently large $n$. On the other hand, once the mathematical approximation error $d^{n+1}$ becomes close to the machine precision, say when $n = 5$ for double precision ($6 \times 8.886 \approx 53$), rounding errors will become a significant component of the final error no matter how the computation is organized, so it may be preferable to use a smaller $n$ and then apply the same correction repeatedly. The simplest case using $n = 1$, a linear polynomial in $d$, can be done with just one more $\text{fma}$ operation. Given an approximation $y$ and its corresponding error estimate $d$ calculated as shown, from the computation

$$y' = y + dy$$

the result is, as expected:
Floating-Point Division

The magnitude of the relative error has thus been squared, or looked at another way, the number of significant bits has been approximately doubled. This, in fact, is exactly a step of the traditional Newton-Raphson iteration for reciprocals, \( y_{n+1} = y_n \cdot (2 - y_n) \). In order to get a still better approximation, one can either use a longer polynomial in \( d \), or repeat the Newton-Raphson linear correction several times. Mathematically speaking, repeating the Newton-Raphson iteration \( n \) times is equivalent to using a polynomial \( 1 + d + d^2 + d^3 + \ldots + d^{2^{n-1}} \). For example, since \( d' = \varepsilon^2 = d^2 \), two iterations yield:

\[
y'' = y(1 + d)(1 + d^2) = y(1 + d + d^2 + d^3)
\]

Whether a repeated Newton iteration or a more direct power series evaluation is better depends on a careful analysis of efficiency and the impact of rounding error. The algorithms discussed here use both, as appropriate.

**Refining Quotients**

The ultimate goal is to accurately approximate the quotient \( a/b \) rather than merely the reciprocal \( 1/b \). Given a reciprocal approximation \( y \), one can always simply multiply it by \( a \). But although satisfactory for obtaining an approximate quotient, this kind of approach can never guarantee getting the last bit right, so it will turn out that for IEEE-correct results, it is necessary to consider how to refine a quotient \( q \) directly. Suppose a quotient has relative error \( \eta \):

\[
q = \frac{a}{b}(1 + \eta)
\]

A remainder term can be computed by a single _fnma_ operation:

\[
r = a - bq
\]

Neglecting the rounding error
Chapter 8: Division, Remainder, and Square Root

In order to use this remainder term to improve $q$, a reciprocal approximation $y = \frac{1}{b}(1+\varepsilon)$ is also needed. Given these components, the $\text{fma}$ operation

$$q' = q + ry$$

results in:

$$q' = q + ry$$

$$= \frac{a}{b}(1 + \eta) - a\eta\frac{1}{b}(1 + \varepsilon)$$

$$= \frac{a}{b}(1 + \eta - \eta(1 + \varepsilon))$$

$$= \frac{a}{b}(1 - \varepsilon\eta)$$

If both $\varepsilon$ and $\eta$ are of the same order, this again yields a quadratic improvement from one refinement stage to the next. Thus, by refining both quotient and reciprocal approximations together, extremely accurate approximations can be obtained.

**IEEE-754 Correctness**

While rounding errors have been neglected up to this point, it’s fairly straightforward to place a sensible bound on their effect. While the relative errors in the approximations are significantly above $2^{-p}$ (where $p$ is the precision of the floating-point format), the effects of rounding error on the overall error are minor. But once one gets close to having a perfectly rounded result, rounding error becomes highly significant. How the algorithm is designed and verified depends radically on whether higher precision is available.

Quite generally, whatever the structure of the algorithm one can consider it as rounding a final approximation $q^*$ of the exact quotient $a/b$ to yield an answer $q$. This final approximation $q^*$ is the mathematically exact answer from the final floating-point operation, ignoring its rounding error. For IEEE correctness, it must be ensured that $q^*$ rounds to the same floating-point number as the exact quotient $a/b$. Consider the
round-to-nearest mode for the present, since it is much easier to get cor-
rect behavior in other rounding modes. According to the IEEE Standard
754, whether a real number $z$ rounds up to the next or down to the pre-
vious floating-point number depends on which is closer to $z$. The mid-
point between the two adjacent floating-point numbers is the threshold
for this decision. In the following diagram, longer lines indicate floating-
point numbers, and shorter ones the midpoints that mark the threshold
of rounding decisions. In the case of the particular $q^*$ and $a/b$ shown,
the rounding will be incorrect because $q^*$ will round to the number
above it, while $a/b$ will round to the number below it.

The first point to note is that while a quotient of two floating-point
numbers may itself be exactly a floating-point number (e.g. $6/3 = 2$) it
cannot be exactly a midpoint between two normal floating-point num-
bers. For if $e^{a/b} = m$ where $m$ is a midpoint, then consequently $a = bm$.
Since $m$ is a midpoint, it has $p + 1$ significant digits, yet $a = bm$ is a float-
ing-point number and so has at most $p$, which is impossible (unless
$b = 0$, which is not normally considered when dividing). Note however
that this property breaks down if the quotient is in the denormal range.
For example, $2^{-\infty} \times 1.1111111/2$ is exactly the midpoint between two
denormal floating-point numbers, since there is no floating-point num-
ber $2^{E_{\infty} - 1} \times 1.111111$. Sometimes extra care is needed in this case.

Since there are only finitely many pairs of floating-point values (albeit
a huge number), this means that there is a nonzero minimal distance $w$
between floating-point quotients and midpoints such that $0 < w \leq \frac{|a - m|}{b}$
for any floating-point numbers $a$ and $b$ and any midpoint $m$. One way
of guaranteeing perfect rounding, then, is to ensure that $q^* - \frac{a}{b} < w$. If
this is the case, then clearly $q^*$ and $a/b$ cannot be separated by a mid-
point, for if $m$ lay between $q^*$ and $a/b$ then also $|a - m| < w$, a contradic-
tion. But in order to assess whether this property holds, an explicit
bound on $w$ is needed. As might be expected, one can make $a/b$ closest
to a midpoint by making $a/b$ itself small, where the floating-point num-
bers around it are closer together. In fact, the following theorem charac-
terizes the distance relative to the size of the quotient $a/b$:

**Theorem 6.1** (Exclusion zone theorem for quotients.) If $a$ and $b$ are
floating-point numbers, with $a$ in the normal range and $b \neq 0$, then if $c$
is either a floating-point number or a midpoint between two floating-
point numbers, then either \( a/b = c \) (which cannot happen if \( c \) is a midpoint) or

\[
|a/b - c| \geq |a/b|/2^{2p+2}
\]

where \( p \) is the floating-point precision.

**Proof** See “Proving the IEEE Correctness of Iterative Floating-Point Square Root, Divide and Remainder Algorithms” by Cornea [20], or “IA-64 and Elementary Functions: Speed and Precision”, by Markstein [21].

This means that by making the approximation \( q^* \) have a relative error of slightly better than \( 2^{-(2p+2)} \), perfect rounding is guaranteed. This is a fairly stringent requirement, for it means that the value used in the final rounding must be more than twice as accurate as the required precision in the answer. Nevertheless, if higher precision is used for the internal calculations and only the final rounding performed in the required precision, which is easily done in the Itanium architecture, this is entirely possible. For example, when implementing single precision division \((p = 24)\) a relative error of \( 2^{-(2p+2)} = 2^{-50} \) is adequate. This can be achieved using double-precision \((p = 53)\) or double-extended precision \((p = 64)\) for the internal computations. For example, one can use the following algorithm:

1. \( y_0 = \frac{1}{b}(1 + e_0) \) Table lookup (\texttt{frcpa})
2. \( e_0 = 1 - by_0 \) Register, round to nearest \( q_0 = ay_0 \) Register, round to nearest
3. \( q_1 = q_0 + e_0q_0 \) Register, round to nearest \( e_1 = e_0e_0 \) Register, round to nearest
4. \( q_2 = q_1 + e_1q_1 \) Register, round to nearest \( e_2 = e_1e_1 \) Register, round to nearest
5. \( q_3 = q_2 + e_2q_2 \) Register double, round to nearest
6. \( q = q_3 \) Single, \( rc \)

After the initial \texttt{frcpa}, the subsequent instructions are all just \texttt{fma} or \texttt{fnma} instructions, rounding into the format and with the rounding direction indicated at the right, for example register format (64 bits of precision and a 17-bit exponent field) or register double (53 bits of precision and a 17-bit exponent field). The rounding mode \( rc \) indicates the desired rounding for the overall division operation. This pattern will be maintained when presenting more algorithms next.
At first sight, rounding a result first in double precision and then in single precision looks very peculiar, but this is necessary in the special cases when the answer is exactly a floating-point number, as will be explained now. A quite simple relative-error analysis allows one to conclude that the error in $q_3$ before rounding; i.e., $q_3 = q_2 + e_2 q_2$, is at most about $2^{-62.4}$. That is, one can write $q_3 = \frac{a}{b}(1 + \varepsilon)$ for some $\varepsilon$ with $|\varepsilon| \leq 2^{-62.4}$. Now distinguish two cases according to whether $a/b$ is actually representable in the register single format. (The use of register single rather than single simplifies the later argument, which is otherwise complicated by the possibility that $a/b$ could be exactly the midpoint between two denormal numbers.)

If $a/b$ is representable in the register single format, then it is also representable in the larger register double format. Since $q_3 = \frac{a}{b}(1 + \varepsilon)$ with $|\varepsilon| \leq 2^{-62.4}$, it is clear that $q_3 = a/b$ exactly since the error is much less than $\frac{1}{2}ulp(a/b)$, and so $q$ is certainly the IEEE correct answer since it literally results from rounding $a/b$ to single precision.

If $a/b$ is not representable in the register single format, then the relative error for $q_3$ after rounding is still satisfactory, because rounding was into a format with more than twice the number of significand bits as single precision. In fact, $q_3 = \frac{a}{b}(1 + \varepsilon)$ with $|\varepsilon| \leq 2^{-52}$, and examining the exclusion zone theorem, only $|\varepsilon| \leq 2^{-(2^{24}+2)}$ is needed. Consequently, correctness is proved.

Even without using additional precision for internal calculations like this, the `fma` family of instructions allows reaching almost twice the required precision, a relative error of about $2^{-2p-1}$, not far short of the $2^{-2p+2}$ required. So when implementing double precision division, there is also little difficulty reaching the required error bound on $q$ thanks to the availability of double-extended precision.

**Native Precision**

Designing and verifying a division algorithm for the maximum available precision is more difficult. Even with the benefit of the `fma`, it is difficult to realize the kind of accuracy in the result before the last rounding needed to justify correctness using the ‘exclusion zone’ theorem. However, one can often design an algorithm based on the following theorem. Although not given with exactly the same hypothesis, the proof of a similar theorem from “Computation of Elementary Functions on the IBM RISC System/6000 Processor” by Markstein [22] carries over unchanged.
Chapter 8: Division, Remainder, and Square Root

Theorem 6.2 If $q$ is a floating-point number within $1\, ulp$ of the true quotient $a/b$ of two floating-point numbers, and $y$ approximates the exact reciprocal $1/b$ to a relative error $e$ where $e<1/2^p$ ($p$ being as usual the precision of the floating-point format concerned), then the following iteration:

1. $r = a - (b \cdot q)$ Format with precision $p$, round to nearest
2. $q' = q + r \cdot y$ Format with precision $p$, round to nearest

yields the correctly rounded-to-nearest quotient $q'$. Moreover, the first computation $r = a - bq$ in the sequence is exact, incurring no rounding error, so the rounding direction of the first computation is not important.

A quotient $q$ accurate to $1\, ulp$ can be obtained using the refinement methods considered earlier. The previous sequence is also guaranteed to work in other rounding modes, if $q'$ is evaluated in the intended rounding mode. For in this case the difficult cases for rounding are precisely those where an exact quotient is close to a floating-point number. It is easy to see that provided $q'^*$ (that is, $q$ before the final rounding) is within $\frac{1}{2} ulp$ of $a/b$, the difficult cases must be when the quotient is very close to $q$. But in these cases $r$ will have the correct sign and roughly the correct magnitude (including $r = 0$ if the answer is exactly $a/b = q$) and so the correction will always be in the right direction.

In order to use this fact, a value $y$ that satisfies the relative-error hypothesis is needed. For this purpose, the following theorem is useful:

Theorem 6.3 If $y$ is a floating-point number that results from rounding a value $y_0$, and the relative error $e$ in $y_0$ with respect to $\frac{1}{b}$ satisfies $e \leq d/2^{2p}$ for some positive integer $d$, assuming $d \leq 2^{p-2}$, then $y$ will have a relative error less than $\frac{1}{2^p}$ with respect to $\frac{1}{b}$, except possibly if the mantissa of $b$ is one of the $2d$ largest. (That is, when scaled up to an integer $2^{p-1} \leq m_b < 2^p$, then in fact $(2^p - d) \leq m_b < 2^p$.)

Proof For simplicity assume $b>0$, since the general case can be deduced from this. One can therefore write $b = 2^q m_b$ for some integer $m_b$ with $2^{p-1} \leq m_b < 2^p$. In fact, it is convenient to assume that $2^{p-1} < m_b$, since when $b$ is an exact power of 2 the main result follows easily from $d \leq 2^{p-2}$. So:
\[
\frac{1}{b} = 2^{-e} \frac{1}{m_b} \\
= 2^{-(e + 2p - 1)} \left( \frac{2^{2p} - 1}{m_b} \right)
\]

and \( \text{ulp} \left( \frac{1}{b} \right) = 2^{-(e + 2p - 1)} \). In order to ensure that \( \left| y - \frac{1}{b} \right| < \| y \| / 2^p \) it suffices, since \( |y - y_0| \leq \text{ulp} \left( \frac{1}{b} \right) / 2 \), to have:

\[
\left| y_0 - \frac{1}{b} \right| < \left( \frac{1}{b} \right) / 2^p - \text{ulp} \left( \frac{1}{b} \right) / 2
\]

\[
= \left( \frac{1}{b} \right) / 2^p - 2^{-(e + 2p - 1) / 2}
\]

\[
= \left( \frac{1}{b} \right) / 2^p - \left( \frac{1}{b} \right) m_b / 2^{2p}
\]

By hypothesis, \( |y_0 - \frac{1}{b}| \leq \left( \frac{1}{b} \right) d / 2^{2p} \). So it is sufficient if:

\[
\left( \frac{1}{b} \right) d / 2^{2p} < \left( \frac{1}{b} \right) / 2^p - \left( \frac{1}{b} \right) m_b / 2^{2p}
\]

Canceling \( \left( \frac{1}{b} \right) / 2^{2p} \) from both sides, this is equivalent to:

\[
d < 2^p - m_b
\]

Consequently, the required relative error is guaranteed except possibly when \( d \geq 2^p - m_b \), or equivalently \( m_b \geq 2^p - d \), as claimed.

These theorems are both strengthenings of the original theorems given by Markstein [22]. The original version of Theorem 6.2 required \( y \) to be accurate to \( 1/\text{ulp} \), almost perfectly rounded, creating a correspondingly more rigorous requirement on \( y_0 \). The intuition behind this improvement is that the overall error in \( y \) is, in the worst case, the error in \( y_0 \) plus \( 1/\text{ulp} \left( \frac{1}{b} \right) \) rounding error. For this to be always within \( 1/\text{ulp} \left( \frac{1}{b} \right) \) is clearly a rather rigorous condition. But except when the mantissa of \( b \) is very large, \( 1/\text{ulp} \left( \frac{1}{b} \right) \) is significantly less than \( \left( \frac{1}{b} \right) / 2^p \), so to achieve only the relative error condition, a greater error in \( y_0 \) is acceptable. Consequently, one can often rely on an iteration

1. \( e = 1 - b \cdot y \) Format with precision \( p \), round to nearest

2. \( y' = y + e \cdot y \) Format with precision \( p \), round to nearest

even when the initial \( y \) is a fairly poor approximation. The theorem may require checking more than the one exceptional value in Markstein’s theorem, and so in some sense generalizes the original style of argument.
The following algorithm for single precision quotients is justified by these theorems. In contrast to the earlier algorithm, this requires only single precision operations throughout, and therefore cannot be justified based on a simple 'exclusion zone' proof. It turns out that on the basis of Theorem 6.3, 165 special cases for the significand of $b$ need to be examined. For each of these, one can confirm explicitly that $y_3$ still satisfies the relative error criterion.

1. $y_0 = \frac{1}{b}(1 + \epsilon_0)$ Table lookup (frCPA)
2. $d = 1 - by_0$ Single, round to nearest $q_0 = ay_0$ Single, round to nearest
3. $y_1 = y_0 + dy_0$ Single, round to nearest $r_0 = a - bq_0$ Single, round to nearest
4. $e = 1 - by_1$ Single, round to nearest $y_2 = y_0 + dy_1$ Single, round to nearest $q_1 = q_0 + r_0y_1$ Single, round to nearest
5. $y_3 = y_1 + ey_2$ Single, round to nearest $r_1 = a - bq_1$ Single, round to nearest
6. $q = q_1 + r_1y_3$ Single, rc

Note that one may need to use the stack single or register single format for the intermediate calculations in order to avoid overflow or underflow in extreme cases. Similarly, the following algorithm, justified by this theorem with the analysis of 65 special cases, can be used for double-extended precision division, but for some register format numbers well beyond the double-extended range, overflow or underflow may be a concern.

1. $y_0 = \frac{1}{b}(1 + \epsilon_0)$ Table lookup (frCPA)
2. $d = 1 - by_0$ Register, round to nearest $q_0 = ay_0$ Register, round to nearest
3. $d_2 = dd$ Register, round to nearest $d_0 = dd + d$ Register, round to nearest
4. $d_3 = d_2d_2 + d$ Register, round to nearest $y_1 = y_0 + y_0d_3$ Register, round to nearest
5. \[ y_2 = y_0 + y_1 d_5 \]  
   \[ r_0 = a - b q_0 \]  
   Register, round to nearest  
   Register, round to nearest

6. \[ e = 1 - b y_2 \]  
   \[ q_1 = q_0 + r_0 y_2 \]  
   Register, round to nearest  
   Register, round to nearest

7. \[ y_3 = y_2 + e y_2 \]  
   \[ r = a - b q_1 \]  
   Register, round to nearest  
   Register, round to nearest

8. \[ q = q_1 + r y_3 \]  
   Extended, \( rc \)

This latter algorithm uses a degree-6 polynomial correction, exploiting the following factorization:
\[ 1 + d + d^2 + d^3 + d^4 + d^5 + d^6 = 1 + (1 + d + d^2) \cdot ((d^2)^2 + d) \]

**Implementation**

So far, division algorithms have been presented as abstract sequences of \( \text{fma} \)-type operations following an initial \( \text{frcpa} \). This analysis neglected the special cases involving zeros, infinities or NaNs. In fact, in special cases like these, the initial \( \text{frcpa} \) instruction (possibly with assistance from system software) simply delivers the correct result for the quotient \( a/b \) (which may involve producing an infinity or NaN and properly triggering exceptions). The instruction
\[
(qp) \ \text{frcpa} . sf \quad f_1 , \quad p_2 = f_2 , \quad f_3
\]

indicates that it has already delivered the full quotient by clearing the associated predicate register \( p_2 \), and that it has merely delivered a reciprocal approximation by setting \( p_2 \). Thus, when \( p_2 \) is cleared, the subsequent operations in the sequences do not need to be executed—indeed they should not be since often they will not work correctly in such cases.

Therefore the user should organize the remainder of the division algorithm as follows. All the instructions should be predicated by predicate register \( p_2 \), and the final result should be returned in floating-point register \( f_1 \), the same one used in the original \( \text{frcpa} \) instruction. If this is done, everything will work correctly, as can be seen by considering the various possible behaviors of the \( \text{frcpa} \) instruction given in the Itanium architecture documentation.
Chapter 8: Division, Remainder, and Square Root

First, if the qualifying predicate \( q_p \) is 0 (false), then \( p_2 \) is cleared, but no other action is taken. This will have the effect of also predicating off the subsequent operations in the division and hence effectively making the entire operation predicated on \( q_p \). Clearly this is the appropriate action; note that \texttt{frcpa} is exceptional in that it does perform some action even when its predicate is cleared.

Otherwise, if the qualifying predicate is 1 (true), the architecture specifies that \( p_2 \) is set to 1 if and only if \( f_1 \) is set to an approximation to \( 1/f_1 \). In all other cases, it is set to 0 and the correct IEEE result is delivered in \( f_1 \), either directly by the hardware, or via a software assistance fault. (The software assistance mechanism was explained in detail in Chapter 7.) Thus, the main part of the division sequence should be executed precisely if \( p_2 \) is true, and this is exactly what predicating all the instructions after \texttt{frcpa} by \( p_2 \) will achieve. Moreover, the \texttt{frcpa} instruction will not enter this path if there is a danger of overflow or underflow in the kinds of intermediate calculations used in division sequences, such as \( r = a - b q \), but will again deliver the full quotient and clear its predicate register.

For example, the division algorithm for double-extended precision given above may be implemented as follows. Assume that the inputs \( a \) and \( b \) are provided in registers \( f8 \) and \( f9 \) respectively, and the quotient is to be delivered in \( f10 \).

```plaintext
frcpa f10, p6 = f8, f9;; // y_0 = 1/b (1 + eps)

(p6) fnma.s1 f11 = f9, f10, f1 // d = 1 - b y_0
(p6) fmpy.s1 f12 = f8, f10;; // q_0 = a y_0
(p6) fmpy.s1 f13 = f11, f11 // d_2 = d d
(p6) fma.s1 f14 = f11, f11, f11;; // d_3 = d d + d
(p6) fma.s1 f13 = f13, f13, f11 // y_1 = y_0 d + y_0
(p6) fma.s1 f14 = f14, f10, f14, f10;; // y_2 = y_1 d_5 + y_0
(p6) fnma.s1 f11 = f9, f12, f8;; // r_0 = a - b q_0
(p6) fnma.s1 f14 = f9, f13, f1 // e = 1 - b y_2
(p6) fma.s1 f12 = f11, f13, f12;; // q_1 = r_0 y_2 + q_0
(p6) fma.s1 f13 = f14, f13, f13 // y_3 = e y_2 + y_2
(p6) fnma.s1 f11 = f9, f12, f8;; // r = a - b q_1
(p6) fma.s0 f10 = f11, f13, f12;; // q = r y_3 + q_1
```
Actual source code for a number of division, square root and remainder algorithms, based on the principles discussed in this chapter, is provided free of charge by Intel Corporation and may be downloaded from the Web [14]. A detailed explanatory document accompanying the source code is also available [14].

Special Division Algorithms

The previous discussion was focused on obtaining correctly rounded results for arbitrary values of \(a\) and \(b\) according to the IEEE Standard. If only a good approximation to \(a/b\) is required, one can often produce simpler algorithms. For example, the native-precision algorithms, and some others too, are based on a final correction of a penultimate approximation \(q\) via more \(\text{fma}\) operations. This penultimate approximation is already accurate to \(1\text{ulp}\), and often quite close to \(\frac{1}{2}\text{ulp}\), so one can often simply omit the final two correction stages.

Another situation where division algorithms can be optimized, even if an IEEE-correct result is required, is when \(b\) is known at compile time, or at least known earlier than \(a\). A common optimization in such cases is simply to multiply by \(\frac{1}{b}\). When \(\frac{1}{b}\) is exactly representable—that is, when \(b\) is a power of 2—this will already give an IEEE-correct result and no more needs to be done. Indeed, most numerical programmers would use \(0.5 \cdot x\) in place of \(x/2.0\), or would hope that the compiler would do it for them. On the Itanium architecture, the splitting of the division into a few simpler operations allows much improved performance for any compile-time constant \(b\), while still guaranteeing an IEEE-correct result. Assuming \(y\) is a correctly rounded \(1/b\), computed at compile-time, then even in native precision one can—assuming no overflow or underflow—do simply:

1. \(q = a \cdot y\) Format with precision \(p\), round to nearest
2. \(r = a - bq\) Format with precision \(p\), round to nearest
3. \(q' = q + r \cdot y\) Format with precision \(p\), \(rc\)

For since

\[
\left| y - \frac{1}{b} \right| \leq \frac{1}{2}\text{ulp}(1/b) \leq |1/b|/2^p
\]

where \(p\) is the floating-point precision, it is also the case that:

\[
\left| ay - \frac{a}{b} \right| \leq \frac{|a|}{b} \cdot 2^p \leq \text{ulp}\left(\frac{a}{b}\right),
\]
and therefore \( ay \) after rounding will also be within one \( u/ p \) of \( a/b \). This and correct rounding of \( y \) is all that is needed for the main theorem 6.2. This sequence has a latency of 3 floating-point instruction latencies, whereas a standard division algorithm might take as many as 8 (for double-extended precision).