Formal verification of IA-64 division algorithms

John Harrison

Intel Corporation, EY2-03
5200 NE Elam Young Parkway
Hillsboro, OR 97124, USA

Abstract. The IA-64 architecture defers floating point and integer division to software. To ensure correctness and maximum efficiency, Intel provides a number of recommended algorithms which can be called as subroutines or inlined by compilers and assembly language programmers. All these algorithms have been subjected to formal verification using the HOL Light theorem prover. As well as improving our level of confidence in the algorithms, the formal verification process has led to a better understanding of the underlying theory, allowing some significant efficiency improvements.

1 Introduction

IA-64 is a new 64-bit computer architecture jointly developed by Hewlett-Packard and Intel, and the Intel Itanium™ processor is its first silicon implementation. We will summarize below the details of the IA-64 instruction set architecture (ISA) necessary for the present paper. A more complete description may be found in the IA-64 Application Developer's Architecture Guide, available from Intel in printed form and online.[1]

To avoid some of the limitations of traditional architectures, IA-64 incorporates a unique combination of features, including an instruction format encoding parallelism explicitly, instruction predication, and speculative/advanced loads [4]. Nevertheless, it also offers full upwards-compatibility with IA-32 (x86) code.

1.1 The IA-64 floating point architecture

The IA-64 floating point architecture has been carefully designed to allow high performance. Features include multiple floating-point status fields and special instructions for transferring data between integer and floating point registers. The centerpiece of the architecture is the `fma` (floating point multiply-add or fused multiply-accumulate) instruction. This computes $xy+z$ from inputs x, y and z with a single rounding error. Except for subtleties over signed zeros, floating point addition and multiplication are just degenerate cases of `fma`, $1y+z$ and $xy+0$, so do not need separate instructions. Variants of the `fma` switch signs of operands: `fms` computes $xy-z$ while `fnma` computes $z-xy$.

The IA-64 architecture supports several different floating point formats compatible with the IEEE 754 Standard for Binary Floating-Point Arithmetic [10]. For the four most important formats, we give the conventional name, the precision, and the minimum

---

[1] See `http://developer.intel.com/design/ia64/downloads/adag.htm`.

and maximum exponents. Thus, numbers in a format with precision p and minimum and maximum exponent E_{min} and E_{max} are those representable as:

$$\pm d_0.d_1d_2d_3\cdots d_{p-1} \times 2^e$$

with the $d_i \in \{0,1\}$ and $E_{min} \leq e \leq E_{max}$.

Format name	p	E_{min}	E_{max}
Single	24	-126	127
Double	53	-1022	1023
Double-extended	64	-16382	16383
Register	64	-65534	65535

The single and double formats are mandated and completely specified in the Standard. The double-extended format (we will often just call it 'extended') is recommended and only partially specified by the Standard. The register format has the same precision as extended, but allows greater exponent range, helping to avoid overflows and underflows in intermediate calculations. As well as these "scalar" formats, IA-64 features a SIMD format where two single-precision numbers are packed in a floating point register and the pair operated on in parallel. Numerically, this amounts to just two parallel copies of the single-precision format, but pragmatically it places different demands on the programmer since one can no longer use higher intermediate precision or range while maintaining the additional level of parallelism.

Most operations, including the `fma`, take arguments and return results in some of the 128 floating point registers provided for by IA-64, in which floating point numbers from all formats map onto a standard bit encoding. By a combination of settings in the multiple status fields and completers on instructions, the results of operations can be rounded in any of the four IEEE rounding modes (to nearest, towards positive or negative infinity, and towards zero) and into any of the supported floating point formats, whatever format the operands come from.

1.2 Division in software

In most current computer architectures, in particular the Intel IA-32 (x86) architecture currently represented by the Pentium® III processor, instructions are specified for the floating point and integer division operations. In IA-64, the only instruction specifically intended to support division is the *floating point reciprocal approximation* instruction, `frcpa`. This merely provides an approximate reciprocal which software can use to generate a correctly rounded quotient. There are several reasons for relegating division to software.

- By implementing division in software it immediately inherits the high degree of pipelining in the basic `fma` operations. Even though these operations take several clock cycles, new ones can be started each cycle while others are in progress. Hence, many division operations can proceed in parallel, leading to much higher throughput than is the case with typical hardware implementations.

- Greater flexibility is afforded because alternative algorithms can be substituted where it is advantageous. It is often the case that in a particular context a faster algorithm suffices, e.g. because the ambient IEEE rounding mode is known at compile-time, or even because only a moderately accurate result is required (e.g. in some graphics applications).
- In typical applications, division is not an extremely frequent operation, and so it may be that die area on the chip would be better devoted to something else. However it is not so infrequent that a grossly inefficient software solution is acceptable, so the rest of the architecture needs to be designed to allow reasonably fast software implementations.

1.3 Formal floating point theory

The formal verifications are conducted using the freely available[2] HOL Light prover [7]. HOL Light is a version of HOL [5], itself a descendent of Edinburgh LCF [6] which first defined the 'LCF approach' that these systems take to formal proof. LCF provers explicitly generate proofs in terms of extremely low-level primitive inferences, in order to provide a high level of assurance that the proofs are valid. In HOL Light, as in most other LCF-style provers, the proofs (which can be very large) are not usually stored permanently, but the strict reduction to primitive inferences in maintained by the abstract type system of the interaction and implementation language, which for HOL Light is CAML Light [16, 3]. This language serves as a programming medium allowing higher-level derived rules (e.g. to automate linear arithmetic, first order logic or reasoning in other special domains) to be programmed as reductions to primitive inferences, so that proofs can be partially automated. In general, however, the user must describe the proof at a moderate level of detail.

The verifications described here draw extensively on a formalized theory of real analysis [8] and floating point arithmetic [9]. These sources should be consulted for more details, but we now summarize some of the main formal concepts used in the present paper.

HOL notation is generally close to traditional logical and mathematical notation. However, the type system distinguishes natural numbers and real numbers, and maps between them by `&`; hence `&2` is the real number 2. The multiplicative inverse x^{-1} is written `inv(x)`, the absolute value $|x|$ as `abs(x)` and the power x^n as `x pow n`.

Much of the theory of floating point numbers is generic. Formats are identified by triples of natural numbers `fmt` and the corresponding set of representable real numbers, ignoring the upper limit on the exponent range, is `iformat fmt`. The second field of the triple, extracted by the function `precision`, is the precision, i.e. the number of significand bits. The third field, extracted by the `ulpscale` function, is N where 2^{-N} is the smallest nonzero floating point number of the format.

Floating-point rounding is performed by `round fmt rc x` which denotes the result of rounding the real number `x` into a floating point format `fmt` under rounding mode `rc`, neglecting the upper limit on exponent range. The predicate `normalizes` determines

---

[2] See `http://www.cl.cam.ac.uk/users/jrh/hol-light/index.html`.

whether a real number is within the range of normal floating point numbers in a particular format, i.e. those representable with a leading 1 in the significand, while `losing` determines whether a real number will lose precision, i.e. underflow, when rounded to a given format.

An important concept in floating point arithmetic is a unit in the last place or *ulp*. Though widely used by floating point experts, there are a number of divergent definitions and care is needed in the formalization [9]. To understand the present paper, the following is adequate: if `x` is any real number and `fmt` identifies a floating point format, then `ulp fmt x` ('an ulp in x with respect to floating point format fmt') is the distance between the two closest floating point numbers straddling `x`.

The canonical sign, exponent and significand fields for a representable real number are extracted by functions `decode_sign`, `decode_exponent` and `decode_fraction`. Actual floating-point register bitstrings are distinguished from the real numbers they represent, and the mapping from bitstrings to reals is performed by a function `Val`. Whether a floating point number is normal is determined by a predicate `normal`.

2 Perfect rounding

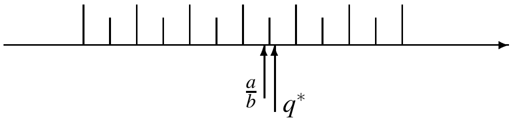
The IEEE Standard for Binary Floating-Point arithmetic [10] specifies that the result of division (as with other basic algebraic operations such as addition, multiplication and square root) should be as if the ideal mathematical result were calculated exactly then rounded in the appropriate rounding mode. Later we examine in detail how to make sure of this for division, but first some general discussion of perfect rounding and the related HOL proofs seems appropriate. Suppose x is the exact result of the operation, e.g. a/b in the case of division, and the calculated answer is z. Whatever the implementation, z will result from rounding an ideal mathematical answer, say y, to some operation. Anticipating later examples, suppose the final step of a division algorithm computes the final quotient from three arguments q, r and y by means of the `fma` operation:

$$\texttt{fma}.pc.sf\ \ q = r_3,\ y_3,\ q_3$$

Because the `fma` itself conforms to (the obvious extrapolation of) the IEEE Standard, the result q arises from rounding the exact mathematical value $q^* = r_3 y_3 + q_3$ in the intended rounding mode. We need to ensure that whatever the rounding mode, q^* and the exact quotient a/b round to the same floating point number.

2.1 Sufficient conditions for perfect rounding

In the following diagram the longer markings denote floating point numbers and the shorter ones the midpoints between floating point numbers. Assuming we are in round-to-nearest mode, $\frac{a}{b}$ will round to the number below it, but q^* to the number above it.



A little reflection shows that in order to ensure perfect rounding in the round-to-nearest mode, a sufficient condition is that q^* and a/b are never separated by a midpoint, for which in turn it suffices that for any midpoint m we have $|a/b - q^*| < |a/b - m|$. Quite generally, we can prove in HOL the following theorem:

```
⊢ ¬(precision fmt = 0) ∧
  (∀m. m ∈ midpoints fmt ⟹ abs(x - y) < abs(x - m))
  ⟹ (round fmt Nearest x = round fmt Nearest y)
```

Obviously this precondition cannot be satisfied if a/b is exactly a midpoint. However it is easy to prove that this cannot occur *provided the quotient is in the normal range*:

```
⊢ a ∈ iformat fmt ∧ b ∈ iformat fmt ∧
  ¬(b = &0) ∧ normalizes fmt (a / b)
  ⟹ ¬(a / b ∈ midpoints fmt)
```

For other rounding modes, an analogous property is required for floating point numbers rather than midpoints. To ensure correctness for all rounding modes, the following suffices.

```
⊢ ¬(precision fmt = 0) ∧
  (∀a. a ∈ iformat(exprange fmt,precision fmt + 1,ulpscale fmt + 1)
       ⟹ abs(x - y) < abs(x - a))
  ⟹ (round fmt rc x = round fmt rc y)
```

Note that we state the theorem in terms of a floating point format with one extra bit of precision, which is exactly the floating point numbers plus midpoints:

```
⊢ ¬(p = 0)
  ⟹ (midpoints(E,p,N) ∪ iformat(E,p,N) = iformat(E,p+1,N+1))
```

Since it is possible for the quotient to be exactly a floating point number, or the midpoint between denormal numbers (e.g. $1.11\cdots11 \times 2^{E_{min}}/2$), we need to deal with these special cases separately. As we shall see, these work automatically for the algorithms as they are structured here.

2.2 Flag settings

We must ensure not only correct results in all rounding modes, but that the flags are set correctly. However, this essentially follows in general from the correctness of the result in all rounding modes (strictly, in the case of underflow, we need to verify this for a format with slightly larger exponent range). For the correct setting of the inexact flag, we need only prove the following HOL theorem:

```
⊢  ¬(precision fmt = 0) ∧
   (∀rc. round fmt rc x = round fmt rc y)
   ⟹ ∀rc. (round fmt rc x = x) = (round fmt rc y = y)
```

The proof is simple: if `x` rounds to itself, then it must be representable. But by hypothesis, `y` rounds to the same thing, that is `x`, in *all rounding modes*. In particular the roundings up and down imply `x <= y` and `x >= y`, so `y = x`. The other way round is similar.

2.3 Exclusion zones

The theorems above show that provided q^* and a/b are closer to each other than a/b is to a floating point number or midpoint, correct rounding is assured. One approach to proving this for a given algorithm is to ask: how close can a/b be to a floating point number or midpoint? A little work allows us to provide an answer to that question [2], which we can formalize as the following HOL theorem:

```
⊢  a ∈ iformat(E,p,N) ∧
   b ∈ iformat(E,p,N) ∧
   c ∈ iformat(E,p+1,N+1) ∧
   &2 pow (p - 1) / &2 pow N <= abs(a) ∧
   ¬(b = &0)
   ⟹ (a / b = c) ∨
       abs(a / b - c) >= abs(a / b) / &2 pow (2 * p + 2)
```

It can be read as saying that every floating point number or midpoint c is surrounded by an ‘exclusion zone’ of size approximately $\frac{|c|}{2^{2p+2}}$ within which no floating point quotient can lie. This implies that if a/b is not exactly a floating point number, then having:

$$|q^* - a/b| < \frac{|a/b|}{2^{2p+2}}$$

would suffice for perfect rounding. By using higher intermediate precision together with the benefit of the `fma`, this kind of relative error can be achieved without trouble, and some of the Intel division algorithms can be verified using the above property. However, in the case of extended precision or SIMD operation, we have no higher intermediate precision available. Then even the `fma` does not quite allow us to guarantee getting q^* that close to a/b in a straightforward way, and we must prove more precise theorems, which we discuss below.

A refinement of the ‘exclusion zone’ approach is not only to identify the width of the exclusion zone but to isolate the inputs a and b where the quotients lie closest to floating point numbers or midpoints. Then one can get away with a worse error bound provided those special cases also work correctly, which one can verify by explicitly running through the algorithm. For the square root, this approach works well [2], and one can feasibly isolate a moderate number of 'difficult cases', allowing a uniform and effective way of verifying square root algorithms (which we have used in analogous verifications for square root). For division, there are too many solutions for a and b for this to be a feasible approach for verification. However, once either a or b is fixed — for example in the special case of finding reciprocals — the number of solutions is typically quite moderate.

3 Implementing division on IA-64

The general form of the IA-64 assembly language `frcpa` instruction is:

```
frcpa.sf q, p = a, b
```

where q, a and b are floating point registers, p is a predicate register, and sf is a floating-point status field. Essentially, a and b are the dividend and divisor respectively, and q is the destination register for the result. The status field sf controls the behavior in exceptional cases, e.g. division by zero, and the predicate register p is set to `false` if the inputs were exceptional, e.g. if a or b was zero. In the exceptional cases, q is set to the IEEE-correct quotient, either directly by the hardware or via a SWA (software assistance) trap, and no further action is necessary. Otherwise p is set to `true` and q is set to an approximation of $1/b$ with a guaranteed relative error:

$$|q - 1/b| \leq 2^{-8.86}|1/b|$$

(In fact, the ISA specifies the details of the approximation more precisely, so that the particular value, which by the way has at most 11 significant bits, is predictable on all IA-64 processors.) Software is then expected to use this to arrive at the IEEE-correct quotient, i.e. the result that would be obtained if the quotient were calculated exactly then rounded using the ambient IEEE rounding mode. Moreover, the six IEEE flags must be set correctly, e.g. the inexact flag is set if and only if the quotient is not exactly a floating point number.

3.1 Intel-provided algorithms

It is not immediately obvious that without tricky and time-consuming bit-twiddling, it is possible to produce an IEEE-correct result and set all the IEEE flags correctly via ordinary software. Remarkably, however, fairly short straight-line sequences of `fma` operations (or negated variants), suffice to do so. This approach to division was pioneered by Markstein [11] on the IBM RS/6000[3] family. It seems that the ability to perform both a multiply and an add or subtract without an intermediate rounding is essential to this, but besides its utility here, the `fma` has many other benefits in improving floating point performance and accuracy.

Intel provides a number of recommended division and square root algorithms, in the form of short sequences of straight-line code written in IA-64 assembly language.

---

[3] All other trademarks are the property of their respective owners.

The intention is that these can be inlined by compilers, used as the core of mathematical libraries, or called on as macros by assembly language programmers. The algorithms are available for download from:

`http://developer.intel.com/software/opensource/numerics.htm`

All the Intel-provided algorithms have been carefully designed to provide IEEE-correct results and trigger IEEE flags and exceptions appropriately. Subject to this correctness constraint, they have been written to maximize performance on the Itanium™ processor. However, they are also likely to be the most appropriate algorithms for future IA-64 processors, even those with significantly different hardware characteristics.

Separate algorithms are provided for the main IA-64 floating point formats (single, double, extended and SIMD), since faster algorithms are usually possible when the required precision is lower. As well as the multiplicity of formats, most algorithms have two separate variants, one of which is designed to minimize latency (i.e. the number of clock cycles between starting the operation and having the result available), and the other to maximize throughput (the number of operations executed per cycle, averaged over a large number of independent instances). Which variant is best to use depends on the kind of program within which it is being invoked.

3.2 Refining approximations

First we will describe in general terms how we can use `fma` operations to refine an initial reciprocal approximation towards a better reciprocal or quotient approximation. For clarity of exposition, we will ignore rounding errors at this stage, and later show how they are taken account of in the formal proof. In the next subsection we cover the subtler issue of guaranteeing correct rounding.

Consider determining the reciprocal of some floating point value b. Starting with a reciprocal approximation y with a relative error ε:

$$y = \frac{1}{b}(1+\varepsilon)$$

we can perform just one `fnma` operation:

$$e = 1 - by$$

and get:

$$\begin{aligned} e &= 1 - by \\ &= 1 - b\frac{1}{b}(1+\varepsilon) \\ &= 1 - (1+\varepsilon) \\ &= -\varepsilon \end{aligned}$$

Now observe that:

$$\begin{aligned}\frac{1}{b} &= \frac{y}{(1+\varepsilon)} \\ &= y(1-\varepsilon+\varepsilon^2-\varepsilon^3+\cdots) \\ &= y(1+e+e^2+e^3+\cdots)\end{aligned}$$

This suggests that we might improve our reciprocal approximation by multiplying y by some truncation of the series $1+e+e^2+e^3+\cdots$. The simplest case using a linear polynomial in e can be done with just one more `fma` operation:

$$y' = y + ey$$

Now we have

$$\begin{aligned}y' &= y(1+e) \\ &= \frac{1}{b}(1+\varepsilon)(1+e) \\ &= \frac{1}{b}(1+\varepsilon)(1-\varepsilon) \\ &= \frac{1}{b}(1-\varepsilon^2)\end{aligned}$$

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. In order to get a still better approximation, one can either use a longer polynomial in e, or repeat the Newton-Raphson linear correction several times. Mathematically speaking, repeating Newton-Raphson iteration n times is equivalent to using a polynomial $1+e+\cdots+e^{2^n-1}$, e.g. since $e' = \varepsilon^2 = e^2$, two iterations yield:

$$y'' = y(1+e)(1+e^2) = y(1+e+e^2+e^3)$$

However, whether 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 Intel algorithms use both, as appropriate.

Now consider refining an approximation to the quotient with relative error ε; we can get such an approximation in the first case by simply multiplying a reciprocal approximation $y \approx \frac{1}{b}$ by a. One approach is simply to refine y as much as possible and then multiply. However, this kind of approach can never guarantee getting the last bit right; instead we also need to consider how to refine q directly. Suppose

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

We can similarly arrive at a remainder term by an `fnma`:

$$r = a - bq$$

when we have:

$$\begin{aligned} r &= a - bq \\ &= a - b\frac{a}{b}(1+\varepsilon) \\ &= a - a(1+\varepsilon) \\ &= -a\varepsilon \end{aligned}$$

In order to use this remainder term to improve q, we also need a reciprocal approximation $y = \frac{1}{b}(1+\eta)$. Now the `fma` operation:

$$q' = q + ry$$

results in, ignoring the final rounding:

$$\begin{aligned} q' &= q + ry \\ &= \frac{a}{b}(1+\varepsilon) - a\varepsilon\frac{1}{b}(1+\eta) \\ &= \frac{a}{b}(1+\varepsilon-\varepsilon(1+\eta)) \\ &= \frac{a}{b}(1-\varepsilon\eta) \end{aligned}$$

3.3 Obtaining the final result

While we have neglected rounding errors hitherto, it is fairly straightforward to place a sensible bound on their effect. To be precise, the error from rounding is at most half an ulp in round-to-nearest mode and a full ulp in the other modes.

```
⊢ ~(precision fmt = 0)
  ⟹ (abs(error fmt Nearest x) <= ulp fmt x / &2) ∧
     (abs(error fmt Down x) < ulp fmt x) ∧
     (abs(error fmt Up x) < ulp fmt x) ∧
     (abs(error fmt Zero x) < ulp fmt x)
```

where

```
⊢ error fmt rc x = round fmt rc x - x
```

It turn, we can easily get fairly tight lower and upper bounds on an ulp in x in terms of the magnitude of x, the upper bound assuming normalization:

```
⊢ abs(x) / &2 pow (precision fmt) <= ulp fmt x
```

and

```
⊢ normalizes fmt x ∧ ~(precision fmt = 0) ∧ ~(x = &0)
   ⟹ ulp fmt x <= abs(x) / &2 pow (precision fmt - 1)
```

Putting these together, we can easily prove simple relative error bounds on all the basic operations, which can be propagated through multiple calculations by simple algebra. It is easy to see that 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. However, once we get close to having a perfectly rounded result, rounding error becomes highly significant. How the algorithm is designed and verified now depends radically on whether we have higher precision available. If we do, then we can usually rely on a simple 'exclusion zone' proof. Otherwise, we need more precise theorems, the central one being the following due to Markstein [11]:

Theorem 1. *If q is a floating point number within 1 ulp of the true quotient a/b of two floating point numbers, and y is the correctly rounded-to-nearest approximation of the exact reciprocal $\frac{1}{b}$, then the following two floating point operations:*

$$
\begin{aligned}
r &= a - bq \\
q' &= q + ry
\end{aligned}
$$

using round-to-nearest in each case, yield the correctly rounded-to-nearest quotient q'.

This is not too difficult to prove in HOL. First we observe that because the initial q is a good approximation, the computation of r cancels so much that no rounding error is committed. (This is intuitively plausible and stated by Markstein without proof, but the formal proof was surprisingly messy.)

```
⊢ 2 <= precision fmt ∧
   a ∈ iformat fmt ∧ b ∈ iformat fmt ∧ q ∈ iformat fmt ∧
   normalizes fmt q ∧ abs(a / b - q) <= ulp fmt (a / b) ∧
   &2 pow (2 * precision fmt - 1) / &2 pow (ulpscale fmt) <= abs(a)
   ⟹ (a - b * q) ∈ iformat fmt
```

Now the overall proof given by Markstein is quite easily formalized. However, we observed that the property actually used in the proof is in general somewhat weaker than requiring y to be a perfectly rounded reciprocal. The theorem actually proved in HOL is:

Theorem 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 $\frac{1}{b}$ to a relative error $< \frac{1}{2^p}$, where p is the precision of the floating point format concerned, then the following two floating point operations:*

$$
\begin{aligned}
r &= a - bq \\
q' &= q + ry
\end{aligned}
$$

using round-to-nearest in each case, yield the correctly rounded-to-nearest quotient q'.

The formal HOL statement is as follows:

```
⊢ 2 <= precision fmt ∧
   a ∈ iformat fmt ∧ b ∈ iformat fmt ∧
   q ∈ iformat fmt ∧ r ∈ iformat fmt ∧
   ¬(b = &0) ∧
   ¬(a / b ∈ iformat fmt) ∧
   normalizes fmt (a / b) ∧
   abs(a / b - q) <= ulp fmt (a / b) ∧
   abs(inv(b) - y) < abs(inv b) / &2 pow (precision fmt) ∧
   (r = a - b * q) ∧
   (q' = q + r * y)
   ⟹ (round fmt Nearest q' = round fmt Nearest (a / b))
```

Although in the worst case, the preconditions of the original and modified theorem hardly differ (recall that $|x|/2^p \leq ulp(x) \leq |x|/2^{p-1}$), it turns out that in many situations the relative error condition is much easier to satisfy. In Markstein's original methodology, one needs first to obtain a perfectly rounded reciprocal, which he proves can be done as follows:

Theorem 3. *If y is a floating point number within 1 ulp of the true reciprocal $\frac{1}{b}$, then one iteration of:*

$$e = 1 - by$$
$$y' = y + ey$$

using round-to-nearest in both cases, yields the correctly rounded reciprocal, except possibly when the mantissa of b consists entirely of 1s.

If we rely on this theorem, we need a very good approximation to $\frac{1}{b}$ before these two further serial operations and one more to get the final quotient using the new y'. However, with the weaker requirement on y', we can get away with a correspondingly weaker y. In fact, we prove:

Theorem 4. *If y is a floating point number that results from rounding a value y_0, and the relative error in y_0 w.r.t. $\frac{1}{b}$ is $\leq \frac{d}{2^{2p}}$ for some natural number d (assumed $\leq 2^{p-2}$), then y will have relative error $< \frac{1}{2^p}$ w.r.t. $\frac{1}{b}$, except possibly if the mantissa of b is one of the d largest. (That is, when scaled up to an integer $2^{p-1} \leq m_b < 2^p$, we have in fact $2^p - d \leq m_b < 2^p$.)*

Proof. *For simplicity we assume $b > 0$, since the general case can be deduced by symmetry from this. We can therefore write $b = 2^e m_b$ 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 $d \leq 2^{p-2}$. Now we have:*

$$\begin{aligned}\frac{1}{b} &= 2^{-e}\frac{1}{m_b}\\ &= 2^{-(e+2p-1)}(\frac{2^{2p-1}}{m_b})\end{aligned}$$

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

$$\begin{aligned}|y_0-\frac{1}{b}| &< (\frac{1}{b})/2^p - ulp(\frac{1}{b})/2\\ &= (\frac{1}{b})/2^p - 2^{-(e+2p-1)}/2\\ &= (\frac{1}{b})/2^p - (\frac{1}{b})m_b/2^{2p}\end{aligned}$$

By hypothesis, we have $|y_0-\frac{1}{b}| \le (\frac{1}{b})\frac{d}{2^{2p}}$. So it is sufficient if:

$$(\frac{1}{b})d/2^{2p} < (\frac{1}{b})/2^p - (\frac{1}{b})m_b/2^{2p}$$

Canceling $(\frac{1}{b})/2^{2p}$ from both sides, we find that this is equivalent to:

$$d < 2^p - m_b$$

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

The HOL statement is as follows. Note that it uses $e = d/2^{2p}$ as compared with the statement we gave above, but this is inconsequential.

```
⊢ 2 <= precision fmt ∧
  b ∈ iformat fmt ∧
  y ∈ iformat fmt ∧
  ¬(b = &0) ∧
  normalizes fmt b ∧
  normalizes fmt (inv(b)) ∧
  (y = round fmt Nearest y0) ∧
  abs(y0 - inv(b)) <= e * abs(inv(b)) ∧
  e <= inv(&2 pow (precision fmt + 2)) ∧
  &(decode_fraction fmt b) <
  &2 pow (precision fmt) - &2 pow (2 * precision fmt) * e
  ⟹ abs(inv(b) - y) < abs(inv(b)) / &2 pow (precision fmt)
```

4 HOL algorithm verifications

We will now give two examples of actual IA-64 division algorithms and describe their HOL verification. Both algorithms are for single precision arithmetic, but one is a scalar algorithm that uses higher precision internally, and the other is a SIMD algorithm that uses only single precision operations. The two verifications thus present interesting contrasts.

4.1 Scalar single precision algorithm

The following algorithm is for single precision computation, but makes clever use of the availability of higher intermediate precision. The steps of the algorithm are grouped into six stages which may be executed in parallel if the particular IA-64 machine allows this, as the Itanium™ processor does. The last column indicates the floating point format into which the result of that operation is rounded. Note that in all the algorithms we consider, all steps but the last are done in round-to-nearest mode, and the last in the ambient rounding mode.

1. $y_0 = \frac{1}{b}(1+\varepsilon)$ `frcpa`

2. $e_0 = 1 - by_0$ Register
   $q_0 = ay_0$ Register

3. $q_1 = q_0 + e_0q_0$ Register
   $e_1 = e_0e_0$ Register

4. $q_2 = q_1 + e_1q_1$ Register
   $e_2 = e_1e_1$ Register

5. $q_3 = q_2 + e_2q_2$ Registerdouble

6. $q = q_3$ Single

The algorithm forms an initial reciprocal approximation y_0 and a quotient approximation q_0, then refines them both by two stages of Newton-Raphson iteration. The subtlety is in the last two lines, where q_3 is rounded to 'register double' (double precision but with a wider exponent range) and subsequently rounded again to single precision, in order to obtain a perfectly rounded result. We now turn to the formal verification.

As detailed in [9], we have written derived rules that can automatically propagate forward known upper and lower ranges on the size of arguments to the result of `fma`-type operations, automatically verifying that the result neither overflows nor loses precision and hence that we can express the result as a relative perturbation of the exact result. HOL's programmability is vital here; these proofs would be extraordinarily tedious to orchestrate by hand.

We do this for all steps of the algorithm, though we then have to reexamine some of them more precisely to make the proof work. Results of later lines have accumulated many errors from previous ones, and again we use an automatic HOL rule to bound these. The bounds derived automatically in this way are naive. For example, if we know $|\varepsilon| < 2^{-24}$, the automatic rule can deduce that $y_0(1+\varepsilon)(1-\varepsilon) = y_0(1+\varepsilon')$ with $|\varepsilon'| \leq 2^{-24} + 2^{-24} + 2^{-24}2^{-24}$. Of course, with a little intelligence, a human can derive $|\varepsilon'| \leq 2^{-48}$. This kind of intelligence has to be injected sometimes, but generally, the automated process is enough to do the donkey work of keeping track of the dozens (hundreds in some other verifications) of ultimately negligible error terms. The first important relative error is in q_3 before rounding, i.e. $q_3^* = q_2 + e_2 q_2$. We find that $q_3^* = \frac{a}{b}(1+e)$ with $|e| \leq 197509/2^{80}$.

Now we 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 in the register single format, then it is *a fortiori* in the register double format. Since $q_3^* = \frac{a}{b}(1+e)$ with $|e| \leq 2^{-62}$, it is clear that $q_3 = a/b$ exactly, 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 in the register single format, then we still have a respectable relative error for q_3 after rounding because rounding was into a format with more than twice single precision. In fact, we have $q_3 = \frac{a}{b}(1+e)$ with $|e| \leq 2^{-52}$, and examining the exclusion zone theorem, we need only $|e| < 2^{-(2\times 24+2)}$. Consequently, correctness is proved.

4.2 SIMD single precision algorithm

The following algorithm is for SIMD single precision computation. It can also be grouped in 6 parallel stages, though on a machine capable of issuing fewer than 3 floating point operations per cycle, some instructions may need to be offset by a cycle.

```
1. $y_0 = \frac{1}{b}(1+\varepsilon)$   frcpa

2. $d = 1 - by_0$         Single
   $q_0 = ay_0$           Single

3. $y_1 = y_0 + dy_0$     Single
   $r_0 = a - bq_0$       Single

4. $e = 1 - by_1$         Single
   $y_2 = y_0 + dy_1$     Single
   $q_1 = q_0 + r_0 y_1$  Single

5. $y_3 = y_1 + ey_2$     Single
   $r_1 = a - bq_1$       Single

6. $q = q_1 + r_1 y_3$    Single
```

Once again we can use the automated tools to produce simple relative error bounds for the intermediate stages. In this case, however, more human intervention in the proofs is necessary, since for extreme inputs the intermediate steps, which have no additional exponent range, *can* overflow or underflow. However, the parallel version of `frcpa` indicates this possibility by clearing a predicate register, triggering the use of a different algorithm. We simply need to verify that the condition tested ensures that no overflow or underflow occurs here, which is easily done.

First, suppose that a/b is exactly, or is very close to, a single precision floating point number c. In this case, the semi-automatic error analysis indicates that $q_1^* = q_0 + r_0 y_1 = \frac{a}{b}(1+\varepsilon)$ with $|\varepsilon| \leq 2^{-25.9}$, close enough to ensure that $q_1 = c$. As before, this ensures that the exact cases work correctly, and allows us to dispose also of the directed rounding mode cases, since these are the only problematic ones for a simple exclusion zone proof. For the more difficult case of round-to-nearest and where the quotient is not close to a floating point number, the critical relative error result is for y_3 before rounding, which is indicated in the HOL goal by the following derived assumptions:

```
[`Val e * Val y_2 + Val y_1 = inv(Val b) * (&1 + e16)`]
[`abs e16 <= &657 / &2 pow 50`]
```

In other words, $y_3^* = \frac{1}{b}(1+e_{16})$ with $|e_{16}| \leq 657/2^{50}$. Since $657/2^{50} \leq 165/2^{2\times 24}$, we can now apply Theorem 4 to show that y_3 will satisfy the relative error criterion needed for Theorem 2, except possibly when the mantissa of b is one of the 165 largest. For these cases, HOL is programmed to evaluate the result of the y_3 computation on them explicitly (dealing with the arbitrary exponent scaling is the only slight difficulty), and it automatically confirms that the criterion is always attained in these cases too. (Note that if this fails, we may still be able to show the overall quotient result will be correct, but it needs somewhat more work and has never arisen in practice so far.) Consequently, we can now apply Theorem 2 and deduce that the final result is correctly rounded and all flags set (subject to the criterion identified for the intermediate results not to overflow or underflow, which matches the cases indicated by the parallel `frcpa`).

A more complicated analysis (which has not been formalized in HOL) suggests that while y_3 always satisfies the relative error criterion, it fails to be perfectly rounded for precisely 12 of the possible 2^{24} input b significands. Consequently, this algorithm could not be justified based only on Markstein's theorems in their original form.

Another situation where the new theorems allow us to justify faster algorithms is extended precision division. Using Markstein's original theorems, it seems the best that can be achieved is the following:

1. $y_0 = \frac{1}{b}(1+\varepsilon)$ [`frcpa`]
2. $e_0 = 1 - b y_0$ $\quad q_0 = a y_0$
3. $y_1 = y_0 + e_0 y_0$ $\quad e_1 = e_0^2$
4. $y_2 = y_1 + e_1 y_1$ $\quad r_0 = a - b q_0$
5. $e_2 = 1 - b y_2$
6. $y_3 = y_2 + e_2 y_2$
7. $e_3 = 1 - b y_3$ $\quad q_1 = q_0 + r_0 y_3$
8. $y_4 = y_3 + e_3 y_3$ $\quad r_1 = a - b q_1$
9. $q_2 = q_1 + r_1 y_4$

However, using the new theorems, we can justify the following, which is faster by one `fma` latency.

```
1. $y_0 = \frac{1}{b}(1+\varepsilon)$  [frcpa]
2. $d = 1 - by_0$          $q_0 = ay_0$
3. $d_2 = dd$              $d_3 = dd + d$
4. $d_5 = d_2 d_2 + d$     $y_1 = y_0 + y_0 d_3$
5. $y_2 = y_0 + y_1 d_5$   $r_0 = a - bq_0$
6. $e = 1 - by_2$          $q_1 = q_0 + r_0 y_2$
7. $y_3 = y_2 + ey_2$      $r = a - bq_1$
8. $q = q_1 + ry_3$
```

5 Conclusions and related work

We have outlined an approach to the formal verification of classes of division algorithms which is a formalization and improvement of standard theoretical approaches [11,2]. The approach has been successfully applied to a large number of division algorithms that Intel is distributing to help IA-64 programmers, helping to give greater confidence in the correctness of these subtle algorithms. Moreover, the verification effort has led to some stronger theorems on which to base algorithms of this type, and so directly to some efficiency improvements.

The verification is conducted on a detailed abstract model of the application programmer's view of the IA-64 ISA, and naturally relies on the IA-64 processor on which the code is run accurately implementing the ISA. Moreover, formal verification cannot completely guard against simple transcription errors in utilized versions of the code, a danger particularly significant since they may be inlined by various compilers and software development tools. For the purpose of isolating such errors as well as providing additional levels of assurance, Intel has also developed extensive validation suites. Formal verification can never completely eliminate the need for such precautions, but it can allow us to focus testing on more productive areas. (Indeed, a particularly attractive feature of the 'exclusion zone' approach [2] is that the difficult cases are not only used in a formal proof but are also good test cases to exercise the algorithm and its practical realization.)

As well as the floating point division work reported here, we have verified various analogous square root algorithms using a formalization of the refined exclusion zone approach [2]. In addition, we have formally verified several integer divide algorithms, which use a specialized floating-point division algorithm as a core. For an overview of the implementation of integer division on IA-64 and proofs of correctness, see [1]. Much more detail about the IA-64 implementation of division, square root and other mathematical functions are given in [12].

The closest related work to that described here is the formal verification of division algorithms reported in [13] and [15]. Although these are respectively for microcode and hardware RTL, and the present work is for software, this difference is not as significant as it may seem, since all these implementations seem to be modeled at a similar level. The major difference is that our work covers algorithms written using the standard resources available to the application programmer, based on a high-level specification that the underlying operations are IEEE-correct. Other work on formal verification of division hardware using a combined theorem prover and model checker [14] is also closely related, but in this work the verification is taken down to a lower level (the implementation in terms of logic gates), and closely integrated with the overall design flow, helping to reduce the chance of transcription errors.

References

1. Marius Cornea, Cristina Iordache, Peter Markstein, and John Harrison. Integer divide and remainder operations in the Intel IA-64 architecture. In Jean-Claude Bajard, Christiane Frougny, Peter Kornerup, and Jean-Michel Muller, editors, *RNC4, the fourth international conference on Real Numbers and Computers*, pages 161–184, 2000.
2. Marius Cornea-Hasegan. Proving the IEEE correctness of iterative floating-point square root, divide and remainder algorithms. *Intel Technology Journal*, 1998-Q2:1–11, 1998. See `http://developer.intel.com/technology/itj/q21998/articles/art_3.htm`.
3. Guy Cousineau and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.
4. Carole Dulong. The IA-64 architecture at work. *IEEE Computer*, 64(7):24–32, July 1998.
5. Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
6. Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
7. John Harrison. HOL Light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *FMCAD'96*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.
8. John Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998. Revised version of author's PhD thesis.
9. John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, 1999. Springer-Verlag.
10. IEEE. Standard for binary floating point arithmetic. ANSI/IEEE Standard 754-1985, The Institute of Electrical and Electronic Engineers, Inc.
11. Peter Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34:111–119, 1990.
12. Peter Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Prentice-Hall, 2000.
13. J Strother Moore, Tom Lynch, and Matt Kaufmann. A mechanically checked proof of the correctness of the kernel of the $AMD5_K86$ floating-point division program. *IEEE Transactions on Computers*, 47:913–926, 1998.
14. John O'Leary, Xudong Zhao, Rob Gerth, and Carl-Johan H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, 1999-Q1:1–14, 1999. `http://developer.intel.com/technology/itj/q11999/articles/art_5.htm`.
15. David Rusinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998. Available on the Web via `http://www.onr.com/user/russ/david/k7-div-sqrt.html`.
16. Pierre Weis and Xavier Leroy. *Le langage Caml*. InterEditions, 1993. See also the CAML Web page: `http://pauillac.inria.fr/caml/`.