Formal verification of floating-point algorithms

John Harrison

Intel Corporation

- Floating point algorithm verification
- HOL Light
- Floating point numbers and formats
- HOL floating point theory
- Division algorithms
- Square root algorithms
- Conclusions

Floating-point algorithm verification

Functions for computing common mathematical functions are fairly mathematically subtle. This applies even to relatively simple operations such as division.

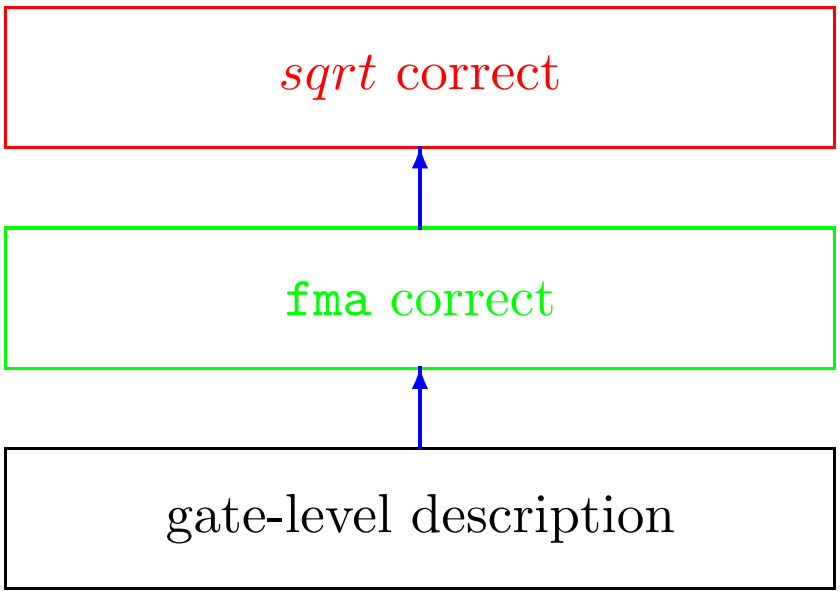
There have been some high-profile errors such as the FDIV bug in some early Intel® Pentium® processors.

Intel therefore uses formal verification to improve the reliability and quality of the underlying algorithms.

The work reported here is at the *algorithmic* level, and is not concerned with gate-level circuit descriptions.

Levels of verification

We are verifying higher-level floating-point algorithms based on assumed correct behavior of hardware primitives.



We will assume that all the operations used obey the underlying specifications as given in the Architecture Manual and the IEEE Standard for Binary Floating-Point Arithmetic.

This is a typical *specification* for lower-level verification.

Context of this work

- The algorithms considered here are implemented in software (as part of math libraries) and microcode in the CPU.
- Whatever the underlying implementation, the basic algorithms and the mathematical details involved are the same, and it makes sense to consider them at the algorithmic level.
- We will focus on the algebraic operations of division and square root for the Intel® Itanium® processor family.
- Similar work is being undertaken for transcendental functions, both for the Itanium® and Pentium® 4 processor families.

Quick introduction to HOL Light

HOL Light is one of the family of theorem provers based on Mike Gordon's original HOL system.

- An LCF-style programmable proof checker written in CAML Light, which also serves as the interaction language.
- Supports classical higher order logic based on polymorphic simply typed lambda-calculus.
- Extremely simple logical core: 10 basic logical inference rules plus 2 definition mechanisms.
- More powerful proof procedures programmed on top, inheriting their reliability from the logical core. Fully programmable by the user.
- Well-developed mathematical theories including basic real analysis.

HOL Light is available for download from:

`http://www.cl.cam.ac.uk/users/jrh/hol-light`

Floating point numbers

There are various different schemes for floating point numbers. Usually, the floating point numbers are those representable in some number n of significant binary digits, within a certain exponent range, i.e.

$$(-1)^s \times d_0.d_1 d_2 \cdots d_n \times 2^e$$

where

- The field $s \in \{0, 1\}$ is the *sign*
- The field $d_0.d_1 d_2 \cdots d_n$ is the *significand* and $d_1 d_2 \cdots d_n$ is the *fraction*. These are not always used consistently; sometimes 'mantissa' is used for one or the other
- The field e is the exponent.

We often refer to $p = n + 1$ as the *precision*.

Intel floating point formats

A floating point format is a particular allowable precision and exponent range.

The Intel architectures support a multitude of possible formats, e.g.

- IEEE single: $p = 24$ and $-126 \leq e \leq 127$
- IEEE double: $p = 53$ and $-1022 \leq e \leq 1023$
- IEEE double-extended: $p = 64$ and $-16382 \leq e \leq 16383$
- Register format: $p = 64$ and $-65534 \leq e \leq 65535$

There are various other hybrid formats, and a separate type of parallel FP numbers, which is SIMD single precision.

The highest precision, 'register', is normally used for intermediate calculations in algorithms.

HOL floating point theory (1)

We have formalized a generic floating point theory in HOL, which can be applied to all the Intel formats, and others supported in software such as quad precision.

A floating point format is identified by a triple of natural numbers `fmt`.

The corresponding set of real numbers is `format(fmt)`, or ignoring the upper limit on the exponent, `iformat(fmt)`.

Floating point rounding returns a floating point approximation to a real number, ignoring upper exponent limits. More precisely

```
round fmt rc x
```

returns the appropriate member of `iformat(fmt)` for an exact value `x`, depending on the rounding mode `rc`, which may be one of `Nearest`, `Down`, `Up` and `Zero`.

HOL floating point theory (2)

For example, the definition of rounding down is:

```
|- (round fmt Down x = closest
        {a | a IN iformat fmt ∧ a <= x} x)
```

We prove a large number of results about rounding, e.g. that a real number rounds to itself if it is in the floating point format:

```
|- ¬(precision fmt = 0) ∧ x IN iformat fmt
   ⟹ (round fmt rc x = x)
```

that rounding is monotonic:

```
|- ¬(precision fmt = 0) ∧ x <= y
   ⟹ round fmt rc x <= round fmt rc y
```

and that subtraction of nearby floating point numbers is exact:

```
|- a IN iformat fmt ∧ b IN iformat fmt ∧
   a / &2 <= b ∧ b <= &2 * a
   ⟹ (b - a) IN iformat fmt
```

Division and square root on Itanium

There are no Itanium instructions for division and square root. Instead, approximation instructions are provided, e.g. the floating point reciprocal approximation instruction.

$$\texttt{frcpa}.sf\ f_1, p_2 = f_3$$

In normal cases, this returns in f_1 an approximation to $\frac{1}{f_3}$. The approximation has a worst-case relative error of about $2^{-8.86}$. The particular approximation is specified in the architecture manual. Similarly, `frsqrta` returns an approximation to $\frac{1}{\sqrt{f_3}}$.

Software is intended to start from this approximation and refine it to an accurate quotient, using for example Newton-Raphson iteration, power series expansions or any other technique that seems effective.

Correctness issues

The IEEE standard states that all the algebraic operations should give the closest floating point number to the true answer, or the closest number up, down, or towards zero in other rounding modes.

It is easy to get within a bit or so of the right answer, but meeting the IEEE spec is significantly more challenging.

In addition, all the flags need to be set correctly, e.g. inexact, underflow, ...

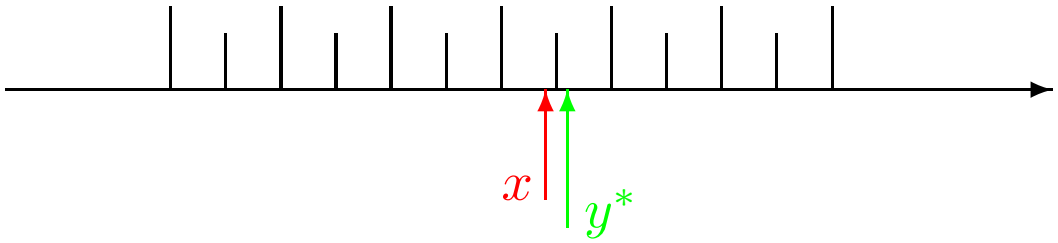
Whatever the overall structure of the algorithm, we can consider its last operation as yielding a result y by rounding an exact value y^*. What is the required property for perfect rounding?

We will concentrate on round-to-nearest mode, since the other modes are either similar (in the case of square root) or much easier (in the case of division).

Condition for perfect rounding

A sufficient condition for perfect rounding is that the closest floating point number to the exact answer x is also the closest to y^*, the approximate result before the last rounding. That is, the two real numbers x and y^* never fall on opposite sides of a midpoint between two floating point numbers.

In the following diagram this is not true; x would round to the number below it, but y^* to the number above it.



How can we prove this?

Proving perfect rounding

There are two distinct approaches to justifying perfect rounding:

- Specialized theorems that analyze the precise way in which the approximation y^* rounds and how this relates to the mathematical function required.
- More direct theorems that are based on general properties of the function being approximated.

We will demonstrate how both approaches have been formalized in HOL.

- Verification of division algorithms based on a special technique due to Peter Markstein.
- Verification of square root algorithms based on an 'exclusion zone' method due to Marius Cornea

Markstein's main theorem

Markstein (IBM Journal of Research and Development, vol. 34, 1990) proves the following general theorem. Suppose we have a quotient approximation $q_0 \approx \frac{a}{b}$ and a reciprocal approximation $y_0 \approx \frac{1}{b}$. Provided:

- The approximation q_0 is within 1 ulp of $\frac{a}{b}$.
- The reciprocal approximation y_0 is $\frac{1}{b}$ rounded to the nearest floating point number

then if we execute the following two `fma` (fused multiply add) operations:

$$\begin{aligned} r &= a - bq_0 \\ q &= q_0 + ry_0 \end{aligned}$$

the value r is calculated exactly and q is the correctly rounded quotient, whatever the current rounding mode.

Markstein's reciprocal theorem

The problem is that we need a perfectly rounded y_0 first, for which Markstein proves the following variant theorem.

If y_0 is within $1ulp$ of the exact $\frac{1}{b}$, then if we execute the following `fma` operations in round-to-nearest mode:

$$\begin{aligned} e &= 1 - by_0 \\ y &= y_0 + ey_0 \end{aligned}$$

then e is calculated exactly and y is the correctly rounded reciprocal, *except possibly when the mantissa of b is all 1s*.

Using the theorems

Using these two theorems together, we can obtain an IEEE-correct division algorithm as follows:

- Calculate approximations y_0 and q_0 accurate to 1 ulp (straightforward). [N `fma` latencies]
- Refine y_0 to a perfectly rounded y_1 by two `fma` operations, and in parallel calculate the remainder $r = a - bq_0$. [2 `fma` latencies]
- Obtain the final quotient by $q = q_0 + ry_0$. [1 `fma` latency].

There remains the task of ensuring that the algorithm works correctly in the special case where b has a mantissa consisting of all 1s.

One can prove this simply by testing whether the final quotient is in fact perfectly rounded. If it isn't, one needs a slightly more complicated proof. Markstein shows that things will still work provided q_0 *overestimates* the true quotient.

Initial algorithm example

Our example is an algorithm for quotients using only single precision computations (hence suitable for SIMD). It is built using the `frcpa` instruction and the (negated) `fma` (fused-multiply-add):

1. $y_0 = \frac{1}{b}(1+\epsilon)$ [`frcpa`]
2. $e_0 = 1 - by_0$
3. $y_1 = y_0 + e_0 y_0$
4. $e_1 = 1 - by_1$ $q_0 = ay_0$
5. $y_2 = y_1 + e_1 y_1$ $r_0 = a - bq_0$
6. $e_2 = 1 - by_2$ $q_1 = q_0 + r_0 y_2$
7. $y_3 = y_2 + e_2 y_2$ $r_1 = a - bq_1$
8. $q = q_1 + r_1 y_3$

This algorithm needs 8 times the basic `fma` latency, i.e. $8 \times 5 = 40$ cycles.

For extreme inputs, underflow and overflow can occur, and the formal proof needs to take account of this.

Improved theorems

In proving Markstein's theorems formally in HOL, we noticed a way to strengthen them. For the main theorem, instead of requiring y_0 to be perfectly rounded, we can require only a relative error:

$$|y_0 - \frac{1}{b}| < |\frac{1}{b}|/2^p$$

where p is the floating point precision. Actually Markstein's original proof only relied on this property, but merely used it as an intermediate consequence of perfect rounding.

The altered precondition looks only trivially different, and in the worst case it is. However it is in general much easier to achieve.

Achieving the relative error bound

Suppose y_0 results from rounding a value y_0^*.

The rounding can contribute as much as $\frac{1}{2}\ ulp(y_0^*)$, which in all significant cases is the same as $\frac{1}{2}\ ulp(\frac{1}{b})$.

Thus the relative error condition after rounding is achieved provided y_0^* is in error by no more than

$$|\frac{1}{b}|/2^p - \frac{1}{2}\ ulp(\frac{1}{b})$$

In the worst case, when b's mantissa is all $1s$, these two terms are almost identical so extremely high accuracy is needed. However at the other end of the scale, when b's mantissa is all 0s, they differ by a factor of two.

Thus we can generalize the way Markstein's reciprocal theorem isolates a single special case.

Stronger reciprocal theorem

We have the following generalization: if y_0 results from rounding a value y_0^* with relative error better than $\frac{d}{2^{2p}}$:

$$|y_0^* - \frac{1}{b}| \leq \frac{d}{2^{2p}}|\frac{1}{b}|$$

then y_0 meets the relative error condition for the main theorem, *except possibly when the mantissa of b is one of the d largest, i.e. when considered as an integer is $2^p - d \leq m \leq 2^p - 1$.*

Hence, we can compute y_0 more 'sloppily', and hence perhaps more efficiently, at the cost of explicitly checking more special cases.

An improved algorithm

The following algorithm can be justified by applying the theorem with $d = 165$, explicitly checking 165 special cases.

1. $y_0 = \frac{1}{b}(1 + \epsilon)$ [`frcpa`]
2. $d = 1 - by_0$ $q_0 = ay_0$
3. $y_1 = y_0 + dy_0$ $r_0 = a - bq_0$
4. $e = 1 - by_1$ $y_2 = y_0 + dy_1$ $q_1 = q_0 + r_0 y_1$
5. $y_3 = y_1 + ey_2$ $r_1 = a - bq_1$
6. $q = q_1 + r_1 y_3$

On a machine capable of issuing three FP operations per cycle, this can be run in 6 FP latencies.

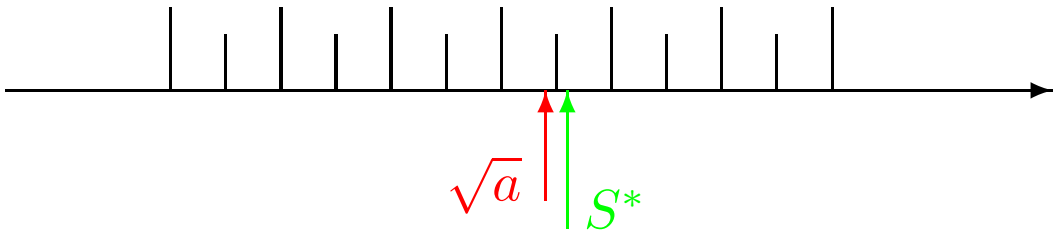
The current Itanium can only issue two FP instructions per cycle, but since it is fully pipelined, this only increases the overall latency by one cycle, not a full FP latency. Thus the whole algorithm runs in 31 cycles.

The square root algorithm

1.	$y_0 = \frac{1}{\sqrt{a}}(1+\epsilon)$	`frsqrta`
	$b = \frac{1}{2}a$	Single
2.	$z_0 = y_0^2$	Single
	$S_0 = ay_0$	Single
3.	$d = \frac{1}{2} - bz_0$	Single
	$k = ay_0 - S_0$	Single
	$H_0 = \frac{1}{2}y_0$	Single
4.	$e = 1 + \frac{3}{2}d$	Single
	$T_0 = dS_0 + k$	Single
5.	$S_1 = S_0 + eT_0$	Single
	$c = 1 + de$	Single
6.	$d_1 = a - S_1 S_1$	Single
	$H_1 = cH_0$	Single
7.	$S = S_1 + d_1 H_1$	Single

Condition for perfect rounding

Recall the general condition for perfect rounding. We want to ensure that the two real numbers $\sqrt{a}$ and S^* never fall on opposite sides of a midpoint between two floating point numbers, as here:



Rather than analyzing the rounding of the final approximation explicitly, we can just appeal to general properties of the square root function.

Exclusion zones

It would suffice if we knew for any midpoint m that:

$$|\sqrt{a} - S^*| < |\sqrt{a} - m|$$

In that case $\sqrt{a}$ and S^* cannot lie on opposite sides of m. Here is the formal theorem in HOL:

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

And this is possible to prove, because in fact every midpoint m is surrounded by an 'exclusion zone' of width $\delta_m > 0$ within which the square root of a floating point number cannot occur.

However, this δ can be quite small, considered as a relative error. If the floating point format has precision p, then we can have $\delta_m \approx |m|/2^{2p+2}$.

Difficult cases

So to ensure the equal rounding property, we need to make the final approximation before the last rounding accurate to *more than twice* the final accuracy.

The fused multiply-add can help us to achieve *just under twice* the accuracy, but to do better is slow and complicated. How can we bridge the gap?

Only a fairly small number of possible inputs a can come closer than say $2^{-(2p-1)}$. For all the other inputs, a straightforward relative error calculation (which in HOL we have largely automated) yields the result.

We can then use number-theoretic reasoning to isolate the additional cases we need to consider, then simply *try them and see*! More than likely we will be lucky, since all the error bounds are worst cases and even if the error is exceeded, it might be in the right direction to ensure perfect rounding anyway.

Isolating difficult cases

By some straightforward mathematics, formalizable in HOL without difficulty, one can show that the difficult cases have mantissas m, considered as p-bit integers, such that one of the following diophantine equations has a solution k for d a small integer. (Typically ≤ 10, depending on the exact accuracy of the final approximation before rounding.)

$$2^{p+2}m = k^2 + d$$

or

$$2^{p+1}m = k^2 + d$$

We consider the equations separately for each chosen d. For example, we might be interested in whether:

$$2^{p+1}m = k^2 - 7$$

has a solution. If so, the possible value(s) of m are added to the set of difficult cases.

Solving the equations

It's quite easy to program HOL to enumerate all the solutions of such diophantine equations, returning a disjunctive theorem of the form:

$$(2^{p+1}m = k^2 + d) \Longrightarrow (m = n_1) \vee \ldots \vee (m = n_i)$$

The procedure simply uses even-odd reasoning and recursion on the power of two (effectively so-called 'Hensel lifting'). For example, if

$$2^{25}m = k^2 - 7$$

then we know k must be odd; we can write $k = 2k' + 1$ and get the derived equation:

$$2^{24}m = 2k'^2 + 2k' - 3$$

By more even/odd reasoning, this has no solutions. In general, we recurse down to an equation that is trivially unsatisfiable, as here, or immediately solvable. One equation can split into two, but never more.

Conclusions

Because of HOL's mathematical generality, all the reasoning needed can be done in a unified way with the customary HOL guarantee of soundness:

- Underlying pure mathematics
- Formalization of floating point operations
- Proof of precise Markstein-type theorems
- Proof of basic exclusion zone properties
- Routine relative error computation for the final result before rounding
- Number-theoretic isolation of difficult cases
- Explicit computation with those cases

Moreover, because HOL is programmable, many of these parts can be, and have been, automated.

The detailed examination of the proofs that formal verification requires threw up significant improvements that have led to some faster algorithms.