Formal Verification Methods
5: Floating-point verification

John Harrison

Intel Corporation

Marktoberdorf 2003

Mon 4th August 2003 (10:35 − 11:20)

Summary

- Itanium overview

- Floating point numbers and Itanium formats

- HOL floating point theory

- Square root algorithm

- Correctness proof in HOL

Itanium overview

The Intel® Itanium® architecture is a new 64-bit computer architecture jointly developed by Hewlett-Packard and Intel, implemented in the Itanium Processor Family (IPF).

- An instruction format encoding parallelism explicitly

- Instruction predication

- Speculative and advanced loads

- Upward compatibility with IA-32 (x86).

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

- Field $s \in \{0, 1\}$ is the *sign*

- 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

- Field e is the exponent.

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

Itanium floating point formats

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

Itanium supports 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 $-16382 \leq e \leq 16383$

- Itanium register format: $p = 64$ and $-65534 \leq e \leq 65535$

There are various other hybrid formats.

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 Itanium 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.

```
|- ¬(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
```

The $(1 + \epsilon)$ property

Most of the routine parts of floating point proofs rely on either an absolute or relative bound on the effect of floating point rounding. The key theorem underlying relative error analysis is the following:

```
|- normalizes fmt x ∧
   ¬(precision fmt = 0)
   ⇒ ∃e. abs(e) <= mu rc / &2 pow (precision fmt − 1) ∧
         (round fmt rc x = x * (&1 + e))
```

This says that given that the value being rounded is in the range of normalized floating point numbers, then rounding perturbs the exact result by at most a relative error bound depending only on the floating point precision and rounding control.
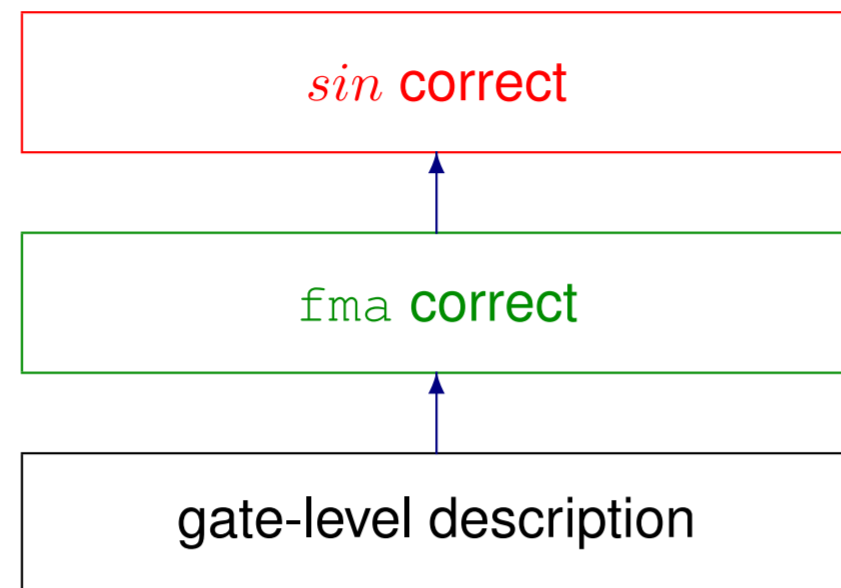
Derived rules apply this result to computations in a floating point algorithm automatically, discharging the conditions as they go.

Levels of verification

Verifying higher-level floating-point algorithms based on assumed correct behavior of hardware primitives.

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

Division and square root on Itanium

There are no hardware instructions (in Itanium mode) for division and square root. Instead, approximation instructions are provided, e.g.

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

In normal cases, this returns in f_1 an approximation to $\frac{1}{\sqrt{f_3}}$ with worst-case relative error of about $2^{-8.85}$.

The particular approximation is specified in the Itanium architecture.

Software is intended to start from this approximation and refine it to an accurate square root, 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, ...

There are various methods for designing IEEE-correct software algorithms, and we will show one such algorithm for square root and show how it was formally verified.

Related techniques can be used for division.

Our algorithm example

Our example is an algorithm for square roots using only single precision computations (hence suitable for SIMD). It is built using two basic Itanium operations:

- The reciprocal square root approximation `frsqrta` described above, which given an input a returns an approximation to $1/\sqrt{a}$ with relative error at most about $2^{-8.85}$.

- The fused multiply add and its negated variant, which calculates $xy + z$ or $z - xy$ with just a single rounding error.

Because it only uses single precision calculations, readers can 'try it at home'.

11

The square root algorithm

1. $y_0 = \frac{1}{\sqrt{a}}(1 + \epsilon)$ `f(p)rsqrta`

   $b = \frac{1}{2}a$ 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

Proving IEEE correctness

Provided the input number is in a certain range, this algorithm returns the correctly rounded square root *and* sets the IEEE flags correctly.

How do we prove that the result is correctly rounded? We will concentrate on round-to-nearest mode, which is the most interesting case. What the algorithm actually returns is the result of rounding the value:

$$S^* = S_1 + d_1 H_1$$

The algorithm is correct if this is always the same as the result of rounding the exact square root $\sqrt{a}$.

Moreover, properties of this value S^*, e.g. whether it is already exactly a floating point number, determine the final flag settings (intermediate steps do not set flags). We also want to make sure these properties are the same as for the exact square root.

Condition for perfect rounding

We prove perfect rounding using a formalization of a technique described here:

A sufficient condition for perfect rounding is that the closest floating point number to $\sqrt{a}$ is also the closest to S^*. That is, the two real numbers $\sqrt{a}$ and S^* never fall on opposite sides of a midpoint between two floating point numbers.

In the following diagram this is not true; $\sqrt{a}$ would round to the number below it, but S^* to the number above it.
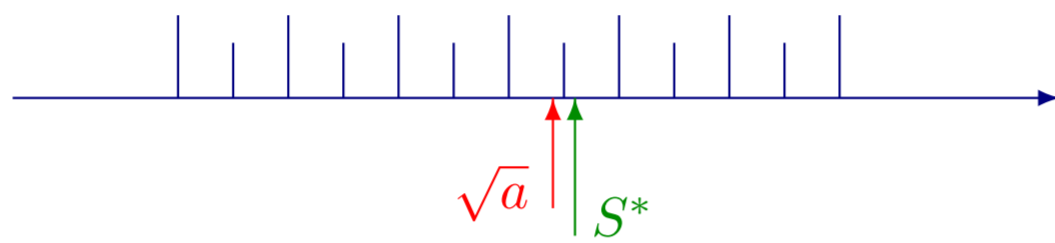
Exclusion zones

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:

```
|- ¬(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 (largely automated in HOL) 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 they will all be correct.

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.

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

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 this has a solution:

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

If so, the possible m values 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:

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

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:

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

In general, we recurse down to an equation that is trivially unsatisfiable, as here, or immediately solvable.

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 that the condition tested ensures perfect rounding

- Routine relative error computation for 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.