Formal Verification of Floating-Point Arithmetic

John Harrison

Intel Corporation

- Formal verification
- Machine-checked proof
- Automatic and interactive approaches
- HOL Light
- Floating point verification
- Specification example
- Lemma examples
- Verification example

Formal Verification

Traditionally, errors in hardware and software have been discovered empirically, by testing them in many possible situations.

However, the number of possible situations is usually so large that we can only exercise a tiny proportion of them.

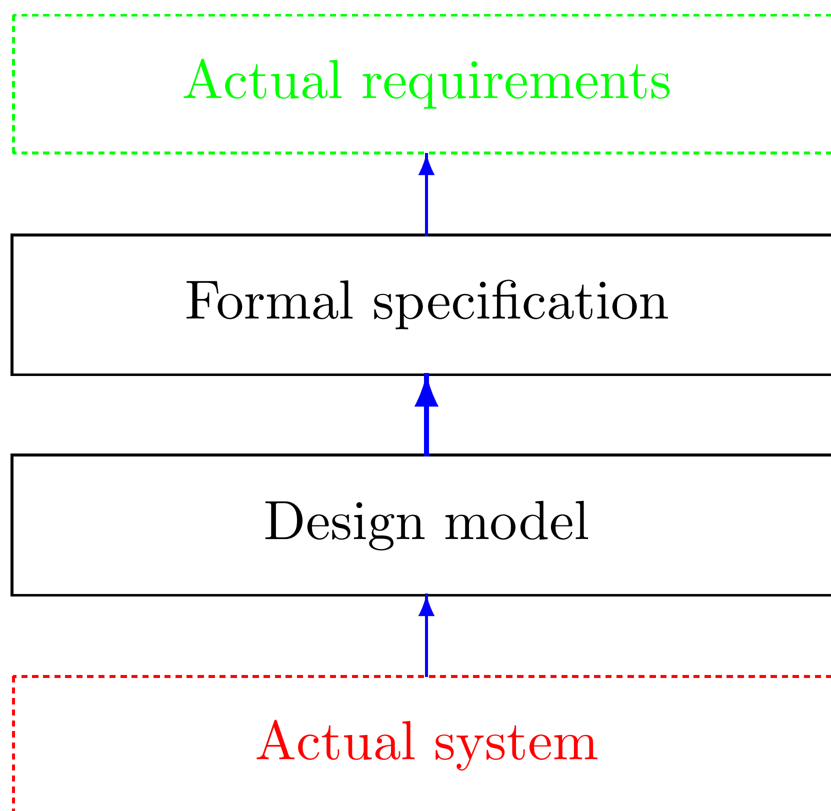
For example, there are about 2^{80} double extended precision floating point numbers. Testing an operation on all of them will probably never be feasible, even if it's only unary.

Pre-silicon testing of microprocessor designs is especially limited, since everything is run on simulators orders of magnitude slower than real hardware.

Formal verification is an alternative that involves trying to *prove* mathematically that a computer system will function as intended.

Formal models

Formal verification aims to prove the correctness of a *design* with respect to a mathematical *formal specification*. This still leaves two gaps:



Note that the same criticisms can be levelled at certain kinds of testing. A simulator is not the same as a real chip. Checking against a 'reference implementation' doesn't prove that the reference is correct.

Formal verification is hard

Writing out a completely formal proof of correctness for real-world hardware and software is difficult.

One needs to make explicit lots of assumptions and special cases that one often forgets about informally. Moreover, one has to avoid making any mistakes or oversights. This is a major undertaking, even for a small system.

It's not easy to get such long and detailed proofs right, nor for others to read them and be assured of their correctness.

The state of the art, at least in the software world, is quite limited. Software verification has been around since the 60s, but there have been few major successes.

Faulty hand proofs

The paper “Synchronizing clocks in the presence of faults” (Lamport & Melliar-Smith, JACM 1985) introduced the Interactive Convergence Algorithm for clock synchronization, and presented a ‘proof’ of it.

A later attempt to reproduce this by Rushby and von Henke in a mechanical theorem prover (EHDM) discovered serious flaws.

The paper presented five supporting lemmas and one main correctness theorem.

Lemmas 1, 2, and 3 were all false. Lemma 4 was false too, but only because of a minor typographical error. The proof of the main induction in the final theorem was wrong. The main result, however, was correct!

Machine-checked proof

A more promising approach is to have the proof checked (or even generated) by a computer program. This offers two potential advantages over doing proofs by hand:

- It can reduce the risk of mistakes. The computer can check that the user only proves results in ways known to be sound.
- The computer can make (some parts of) the proof easier than they would be by hand, even automating large parts of it.

In the hardware world the latter has proven to be especially important, and has led to a recent upsurge of interest in formal hardware verification.

Decidable systems

There are well-known fields of logic and mathematics where validity is decidable, e.g:

- Propositional logic, e.g. $\neg(p \vee q) \Rightarrow \neg p \wedge \neg q$.
- AE fragment of first order logic, e.g. $\forall x.\ \exists y.\ P[x] \Rightarrow P[y]$.
- Linear arithmetic over $\mathbb{N}$, e.g. $x < y \Rightarrow 2x + 1 < 2y$.
- Nonlinear arithmetic over $\mathbb{R}$, e.g. $\exists x.\ x^2 - 3x + 1 = 0$.

This only covers small fragments of mathematics. However, it is often enough to solve significant real-world verification problems.

For example, checking that an optimized combinational circuit has the same behavior as an unoptimized one amounts to proving a formula in propositional logic. Symbolic trajectory evaluation and temporal logical model checking can even verify properties of *sequential* systems.

Theoretical limits

Full automation has strong theoretical limits, by virtue of the following (related) theorems:

- Tarski's theorem on the undefinability of truth
- Gödel's first incompleteness theorem.
- The Church-Turing theorem.

Even if a theory is decidable in principle, the time or space usage of the decision procedure may make it ineffective in practice.

Combinational comparison, STE and temporal logic model checking, for example, are widely used for hardware verification, but rapidly run into practical capacity limits when verifying many real-world systems.

Besides, they oblige us to express both the system model *and* (which is worse) the specification using only a limited fragment of mathematics.

General theorem proving

One can meet these objections by using a general theorem prover.

This can deal with all the high-level mathematics required, and the specification can therefore be written in a more natural way. In fact, the verification can be modularized and structured into layers, with increasingly general levels of specification.

Verification can also be performed generically, e.g. proving n-bit adders correct for arbitrary n rather than some particular value.

However, validity is no longer decidable in theory and certainly not feasible in practice. Instead, theorem proving programs require a skilled user to communicate in a formal way the outline of a mathematical proof, though they can usually fill in simple gaps for themselves.

HOL Light

The prover we use, HOL Light, is based on the approach to theorem proving pioneered in Edinburgh LCF in the 70s. The key ideas are:

- All theorems created by low-level primitive rules.
- Guaranteed by using an abstract type of theorems; no need to store proofs.
- ML available for implementing derived rules by arbitrary programming.

This gives advantages of reliability and extensibility. The system's source code can be completely open. The user controls the means of production (of theorems).

HOL Light includes, built on top of this logical core, a variety of automated proof tools and formalized mathematical theories that can be applied in proofs. Real analysis is particularly useful for floating point verification.

Floating point verification

Nowadays, most floating point implementations are intended to conform to the IEEE Standard for binary floating point arithmetic. This gives rules that fix most aspects of floating point behavior.

A key initial part of our work is to formalize in HOL what IEEE-correct behavior is, as well as IA-64 specific choices where the IEEE standard does not fix behavior. In software verification:

- We are entitled to assume that the basic operations we use (mainly the fused-multiply-add) behave according to specification.
- Based on that assumption, we need to show that certain higher-level algorithms also obey such a specification or an appropriate variant.

If we were verifying the low-level hardware, our starting assumptions would be, e.g. logical or electrical properties of gates.

Key aspects of the IEEE Standard

- Defines a variety of floating point formats such as 'single' and 'double'. Numbers in a given format obey restrictions on precision (p) and exponent range ($E_{min} \leq e \leq E_{max}$):

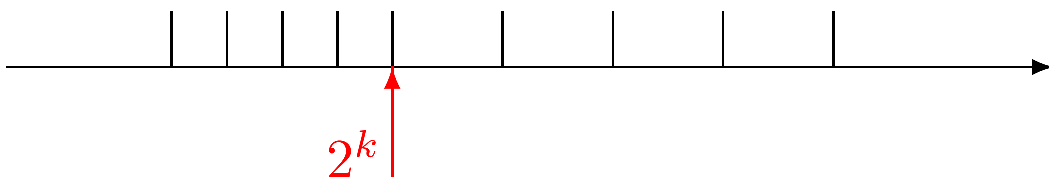
$$x = (-1)^s \times d_1.d_2 d_3 \cdots d_p \times 2^e$$

- The *rounding* operation is taken as basic, with the default mode being round-to-nearest, where a number is mapped to the closest floating point number to it.
- All arithmetic operations normally proceed as if they generated an exact mathematical result and then rounded it.
- Appropriate flags are set or exceptions generated in special situations like overflow, underflow or invalid operations.

Specification example: ulps

The IEEE Standard does not specify that the transcendental function like `exp` and `sin` give correctly rounded results, since no algorithm with guaranteed good performance is known. Instead, it's customary to measure their error in terms of 'units in the last place' (ulps).

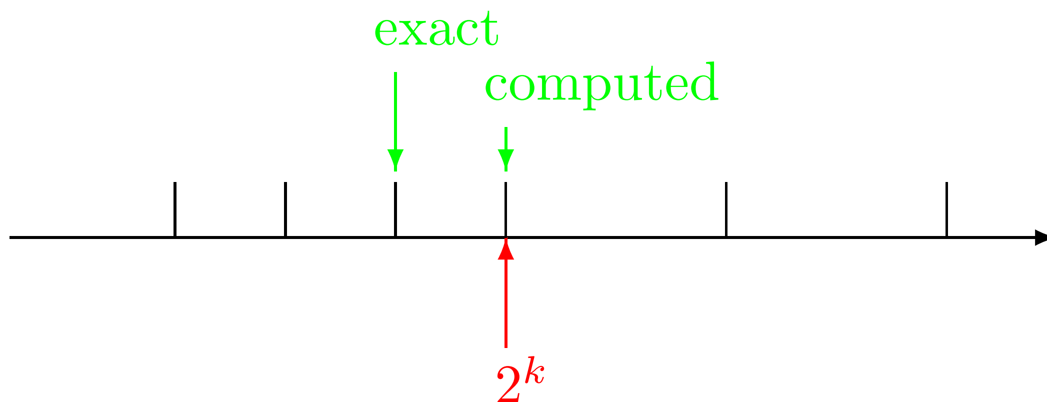
While ulps are a standard way of measuring error, there's a remarkable lack of unanimity in published definitions of the term. One of the merits of a formal treatment is to clear up such ambiguities.



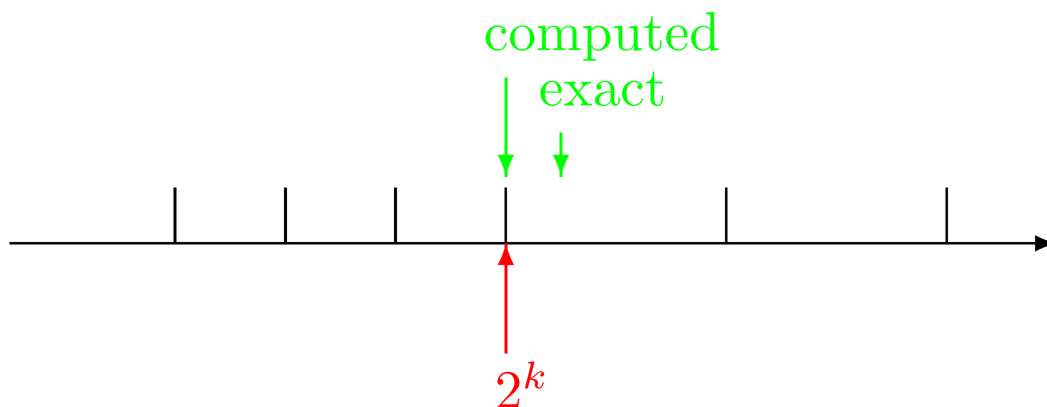
Roughly, a unit in the last place is the gap between adjacent floating point numbers. But at the boundary 2^k between 'binades', this distance changes.

Two definitions

Definitions in two common reference works disagree. An error of 0.5 ulp according to Goldberg, but intuitively 1 ulp.



An error of 0.4 ulp according to Muller, but intuitively 0.2 ulp. Rounding up is worse...



Our definition: $ulp(x)$ is the distance between the closest pair of floating point numbers a and b with $a \leq x \leq b$.

Lemma examples

There are many results used informally by floating point algorithm designers that need to be formally proved in HOL as lemmas. A simple example is that rounding is monotonic:

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

Particularly interesting, and tricky to prove, are results that guarantee certain quantities can be calculated exactly. For example this is a classic result:

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

while the following says that we can always get a sum of floating point numbers exactly as a ‘large’ and ‘small’ part by adding them as usual and then getting a correction by subtracting the addends from the result, the larger one first:

```
|- x IN iformat fmt ∧ y IN iformat fmt ∧ abs(x) <= abs(y)
   ⇒ (round fmt Nearest (x + y) - y) IN iformat fmt ∧
      (round fmt Nearest (x + y) - (x + y)) IN iformat fmt
```

Verification example

We take as given the correct behavior of the IA-64 `frsqrta` (reciprocal square root approximation) instruction and the `fma` operations that calculate `x y + z` with one rounding error. We put these together into the following algorithm:

1. $y_0 = \frac{1}{\sqrt{a}}(1+\epsilon)$ $\quad b = \frac{1}{2}a$
2. $z_0 = y_0^2$ $\quad S_0 = ay_0$
3. $d = \frac{1}{2} - bz_0$ $\quad k = ay_0 - S_0$ $\quad H_0 = \frac{1}{2}y_0$
4. $e = 1 + \frac{3}{2}d$ $\quad T_0 = dS_0 + k$
5. $S_1 = S_0 + eT_0$ $\quad c = 1 + de$
6. $d_1 = a - S_1S_1$ $\quad H_1 = cH_0$
7. $S = S_1 + d_1H_1$

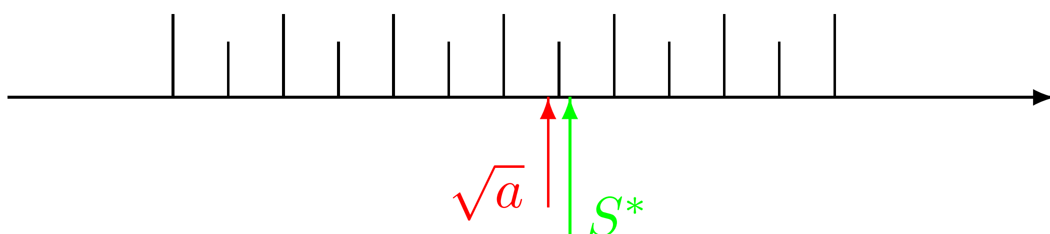
The claim is that this (subject to some range assumptions) calculates the correctly rounded square root in IEEE single precision, and sets all the flags and exceptions correctly.

Condition for perfect rounding

Let S^* be the exact value $S_1 + d_1 H_1$ before the final rounding.

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.



How can we prove that this situation never arises for our algorithm?

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

Because the `fma` does two operations before rounding, we can come close to the required error in S^*, but not quite attain it. We therefore adopt a 2-part proof. This methodology is due to Marius Cornea at Intel:

- Use number theory to find the inputs that might have square roots dangerously close to midpoints between floating point numbers.
- Show by a straightforward relative error analysis that all other inputs are guaranteed to work, by the above exclusion zone reasoning.
- Prove casewise that the exceptional values all give the correct result anyway.

All of these parts can be automated so that HOL performs much of the tedious reasoning behind the scenes. We then just need to plug together the results afterwards.