Formal verification of Floating Point Algorithms

John Harrison

Intel Corporation

- The cost of bugs
- Formal verification
- Machine-checked proof
- Automatic and interactive approaches
- HOL Light
- Formalized real analysis
- Formalized floating point arithmetic
- Algebraic example: square root
- Transcendental example: tangent
- Conclusions

The human cost of bugs

Computers are often used in safety-critical systems where a failure could cause loss of life.

- Heart pacemakers
- Aircraft
- Nuclear reactor controllers
- Car engine management systems
- Radiation therapy machines
- Telephone exchanges (!)
- ...

Financial cost of bugs

Even when not a matter of life and death, bugs can be financially serious if a faulty product has to be recalled or replaced.

- 1994 FDIV bug in the Intel®Pentium® processor: US $500 million.
- Today, new products are ramped much faster...

So Intel is especially interested in all techniques to reduce errors.

Complexity of designs

At the same time, market pressures are leading to more and more complex designs where bugs are more likely.

- A 4-fold increase in bugs in Intel processor designs per generation.
- Approximately 8000 bugs designed into the Pentium 4 ('Willamette') processor.

Fortunately, pre-silicon detection rates are now at least 99.7%.

Just enough to tread water...

Limits of testing

Bugs are usually detected by extensive testing, including pre-silicon simulation.

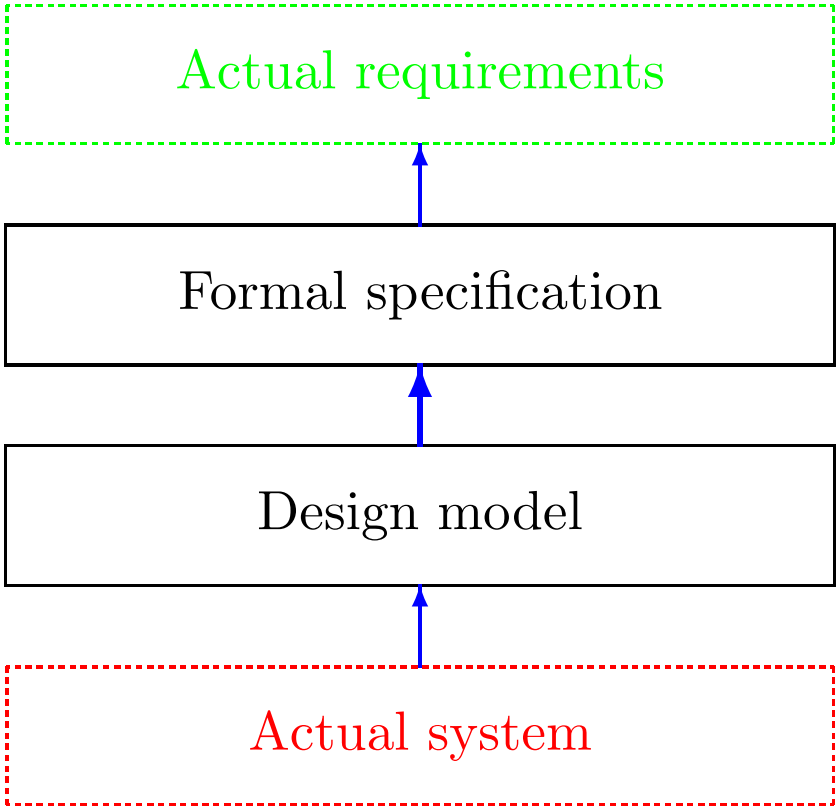
- Slow — especially pre-silicon
- Too many possibilities to test them all

For example:

- 2^{160} possible pairs of floating point numbers (possible inputs to an adder).
- Vastly higher number of possible states of a complex microarchitecture.

Formal verification

Formal verification: mathematically prove the correctness of a *design* with respect to a mathematical *formal specification*.



Verification vs. testing

Verification has some advantages over testing:

- Exhaustive.
- Improves our intellectual grasp of the system.

However:

- Difficult and time-consuming.
- Only as reliable as the formal models used.
- How can we be sure the proof is right?

Faulty hand proofs

"Synchronizing clocks in the presence of faults" (Lamport & Melliar-Smith, JACM 1985)

This introduced the Interactive Convergence Algorithm for clock synchronization, and presented a 'proof' of it.

- Presented five supporting lemmas and one main correctness theorem.
- Lemmas 1, 2, and 3 were all false.
- 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.

- It can reduce the risk of mistakes.
- The computer can automate some parts of the proofs.

There are limits on the power of automation, so detailed human guidance is usually necessary.

The spectrum of theorem provers

From interactive proof checkers to fully automatic theorem provers.

AUTOMATH (de Bruijn)

Stanford LCF (Milner)

Mizar (Trybulec)

. . .

. . .

PVS (Owre, Rushby, Shankar)

. . .

. . .

ACL2 (Boyer, Kaufmann, Moore)

Otter (McCune)

HOL Light

HOL Light is a member of the family of HOL theorem provers.

- 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 and 3 axioms.
- 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 verification

We've used HOL Light to verify the accuracy of floating point algorithms (used in hardware and software) for:

- Division and square root
- Transcendental function such as sin, exp, $atan$.

This involves background work in formalizing:

- Real analysis
- Basic floating point arithmetic

Context

Specific work reported here is for the Intel® Itanium™ processor.

Similar work is underway on software libraries for the Intel Pentium® 4 processor.

Floating point algorithms for transcendental functions are used for:

- Software libraries (C `libm` etc.)
- Implementing x86 hardware intrinsics

The level at which the algorithms are modelled is similar in each case.

Formalized real analysis

- Definitional construction of real numbers
- Basic topology
- General limit operations
- Sequences and series
- Limits of real functions
- Differentiation
- Power series and Taylor expansions
- Transcendental functions
- Gauge integration

Examples of useful theorems

```
|- sin(x + y) =
   sin(x) * cos(y) + cos(x) * sin(y)

|- tan(&n * pi) = &0

|- &0 < x /\ &0 < y
   ==> (ln(x / y) = ln(x) - ln(y))

|- f contl x /\ g contl (f x)
   ==> (g o f) contl x

|- (!x. a <= x /\ x <= b
        ==> (f diffl (f' x)) x) /\
   f(a) <= K /\ f(b) <= K /\
   (!x. a <= x /\ x <= b /\ (f'(x) = &0)
        ==> f(x) <= K)
   ==> !x. a <= x /\ x <= b ==> f(x) <= K
```

HOL floating point theory

Generic theory, applicable to all required formats (hardware-supported or not).

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`.

The $(1 + \epsilon)$ property

Most routine floating point proofs just use results like 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))
```

Rounded result is true result perturbed by relative error.

Derived rules apply this result to computations in a floating point algorithm automatically, discharging the conditions as they go.

Cancellation theorems

Many algorithms also rely on a number of low-level tricks.

Rounding is trivial when the value being rounded is already representable exactly:

```
|- a IN iformat fmt ==> (round fmt rc a = a)
```

Some special situations where this happens are as follows:

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

|- 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
```

Algebraic example: square root

Division and square root operations on the Itanium processor are performed in software.

This offers a number of advantages, particularly increased throughput.

Square root algorithms are constructed using two kinds of instruction:

- The floating point reciprocal square root approximation instruction, which given x returns an 8-bit approximation to $\frac{1}{\sqrt{x}}$.
- The fused multiply-add instruction, which computes $xy + z$ with a single rounding error.

Typical algorithms refine the initial approximation to an accurate square root using Newton-Raphson iteration or power series expansions.

A 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

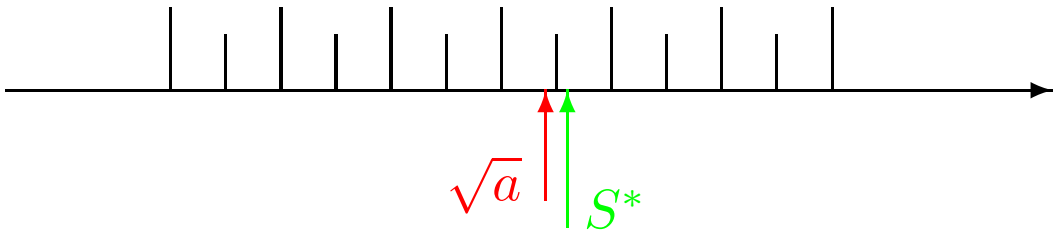
Condition for perfect rounding

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

```
http://developer.intel.com/technology/itj/q21998/articles/art_3.htm
```

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 this?

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

To ensure correct rounding, we need to make the final approximation before the last rounding accurate to *more than twice* the final accuracy.

In fact, even using the fused multiply-add it is not quite that accurate.

However, only a fairly small number of possible inputs a can come closer than say $2^{-(2p-1)}$.

We can then use number-theoretic reasoning to isolate the additional cases we need to consider, then simply try them!

The critical cases can be found by solving diophantine equations of the form:

$$2^q m = k^2 + d$$

to give the mantissa m. In HOL, the solutions can be generated and checked automatically, thanks to HOL's programmability.

Transcendental example: tangent

Works essentially as follows.

- The input number X is first reduced to r with approximately $|r| \leq \pi/4$ such that $X = r + N\pi/2$ for some integer N. We now need to calculate $\pm tan(r)$ or $\pm cot(r)$ depending on N modulo 4.
- If the reduced argument r is still not small enough, it is separated into its leading few bits B and the trailing part $x = r - B$, and the overall result computed from $tan(x)$ and pre-stored functions of B, e.g.

$$tan(B + x) = tan(B) + \frac{\frac{1}{sin(B)cos(B)}tan(x)}{cot(B) - tan(x)}$$

- Now a power series approximation is used for $tan(r)$, $cot(r)$ or $tan(x)$ as appropriate.

Overview of the verification

To verify this algorithm, we need to prove:

- The range reduction to obtain r is done accurately.
- The mathematical facts used to reconstruct the result from components are applicable.
- The pre-stored constants such as $tan(B)$ are sufficiently accurate.
- The power series approximation does not introduce too much error in approximation.
- The rounding errors involved in computing with floating point arithmetic are within bounds.

Most of these parts are non-trivial. Moreover, some of them require more pure mathematics than might be expected.

Why mathematics?

Controlling the error in range reduction becomes difficult when the reduced argument $X - N\pi/2$ is small.

To check that the computation is accurate enough, we need to know:

> How close can a floating point number be to an integer multiple of $\pi/2$?

Even deriving the power series (for $x \neq 0$):

$$cot(x) = 1/x - \frac{1}{3}x - \frac{1}{45}x^3 - \frac{2}{945}x^5 - \ldots$$

is much harder than you might expect.

Conclusions

- Formal verification of mathematical software is industrially important, and can be attacked with current theorem proving technology.
- A large part of the work involves building up general theories about both pure mathematics and special properties of floating point numbers.
- Programmability is critical to automate special procedures like solving diophantine equations, bounding copound errors etc.
- Using HOL Light, we can confidently integrate all the different aspects of the proof from pure mathematics to testing particular arguments.