Formal verification of floating-point arithmetic at Intel

John Harrison

johnh@ichips.intel.com

JNAO

2nd June 2006

Overview

- Computer arithmetic bugs
- Formal verification and theorem proving
- Floating-point arithmetic
- Square root function
- Transcendental functions

Naive use of floating-point arithmetic

In 1982 the Vancouver stock exchange index was established at a level of 1000.

A couple of years later the index was hitting lows of around 520.

The cause was repeated truncation of the index to 3 decimal digits on each recalculation, several thousand times a day.

On correction, the stock index leapt immediately from 574.081 to 1098.882.

Faulty implementation of floating-point arithmetic

Early Hewlett-Packard HP-35 calculators (1972) had several floating-point bugs:

- Exponential function, e.g. $e^{ln(2.02)} = 2.00$
- sin of some small angles completely wrong

At this time HP had already sold 25,000 units, but they advised users of the problem and offered a replacement.

A floating-point bug closer to home

Intel has also had at least one major floating-point issue:

- Error in the floating-point division (FDIV) instruction on some early Intel®Pentium® processors
- Very rarely encountered, but was hit by a mathematician doing research in number theory.
- Intel eventually set aside US $\$475$ million to cover the costs of replacements.

Things are not getting easier

The environment is becoming even less benign:

- The overall market is much larger, so the potential cost of recall/replacement is far higher.
- New products are ramped faster and reach high unit sales very quickly.
- Competitive pressures are leading to more design complexity.

Some complexity metrics

Recent Intel processor generations (Pentium, P6 and Pentium 4) indicate:

- A 4-fold increase in overall complexity (lines of RTL . . .) per generation
- A 4-fold increase in design bugs per generation.
- Approximately 8000 bugs introduced during design of the Pentium 4.

Fortunately, pre-silicon detection rates are now very close to 100%.

Just enough to keep our heads above 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.

So Intel is very active in formal verification.

A spectrum of formal techniques

There are various possible levels of rigor in correctness proofs:

- Programming language typechecking
- Lint-like static checks (uninitialized variables . . .)
- Checking of loop invariants and other annotations
- Complete functional verification

FV in the software industry

Some recent success with partial verification in the software world:

- Analysis of Microsoft Windows device drivers using SLAM
- Non-overflow proof for Airbus A380 flight control software

Much less use of full functional verification. Very rare except in highly safety-critical or security-critical niches.

FV in the hardware industry

In the hardware industry, full functional correctness proofs are increasingly becoming common practice.

- Hardware is designed in a more modular way than most software.
- There is more scope for complete automation
- The potential consequences of a hardware error are greater

Formal verification methods

Many different methods are used in formal verification, mostly trading efficiency and automation against generality.

- Propositional tautology checking
- Symbolic simulation
- Symbolic trajectory evaluation
- Temporal logic model checking
- Decidable subsets of first order logic
- First order automated theorem proving
- Interactive theorem proving

Intel's formal verification work

Intel uses formal verification quite extensively, e.g.

- Verification of Intel® Pentium® 4 floating-point unit with a mixture of STE and theorem proving
- Verification of bus protocols using pure temporal logic model checking
- Verification of microcode and software for many Intel® Itanium® floating-point operations, using pure theorem proving

FV found many high-quality bugs in P4 and verified "20%" of design

FV is now standard practice in the floating-point domain

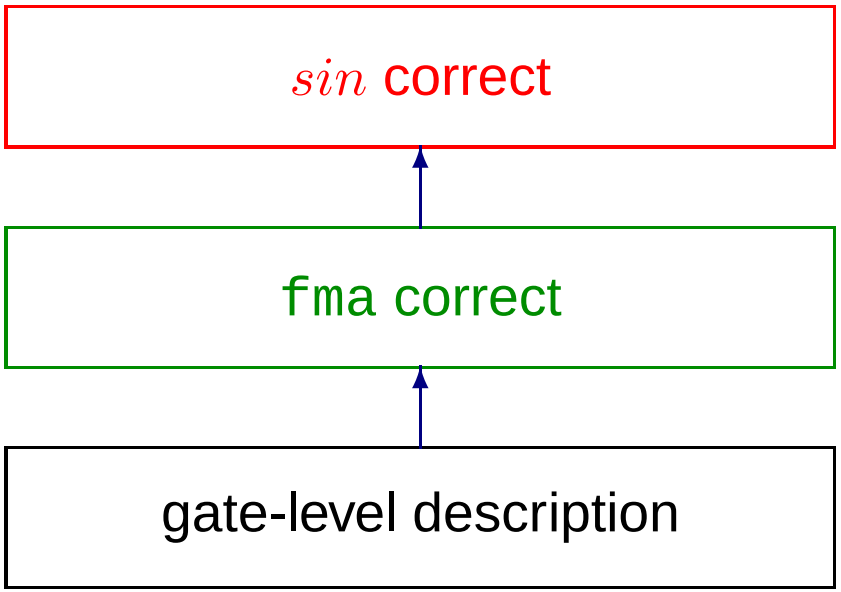
Our work

We will focus on our own formal verification activities:

- Formal verification of floating-point operations
- Targeted at the Intel® Itanium® processor family.
- Conducted using the interactive theorem prover HOL Light.

Levels of verification

High-level algorithms assume correct behavior of some hardware primitives.



Proving my assumptions is someone else's job . . .

Why floating-point?

There are obvious reasons for focusing on floating-point:

- Known to be difficult to get right, with several issues in the past.
  We don't want another FDIV!
- Quite clear specification of how most operations *should* behave.
  We have the IEEE Standard 754.

However, Intel is also applying FV in many other areas, e.g. control logic, cache coherence, bus protocols . . .

Why interactive theorem proving?

Limited scope for highly automated finite-state techniques like model checking.

It’s difficult even to specify the intended behaviour of complex mathematical functions in bit-level terms.

We need a general framework to reason about mathematics in general while checking against errors.

Theorem provers for floating-point

There are several theorem provers that have been used for floating-point verification, some of it in industry:

- ACL2 (used at AMD)
- Coq
- HOL Light (used at Intel)
- PVS

All these are powerful systems with somewhat different strengths and weaknesses.

HOL Light overview

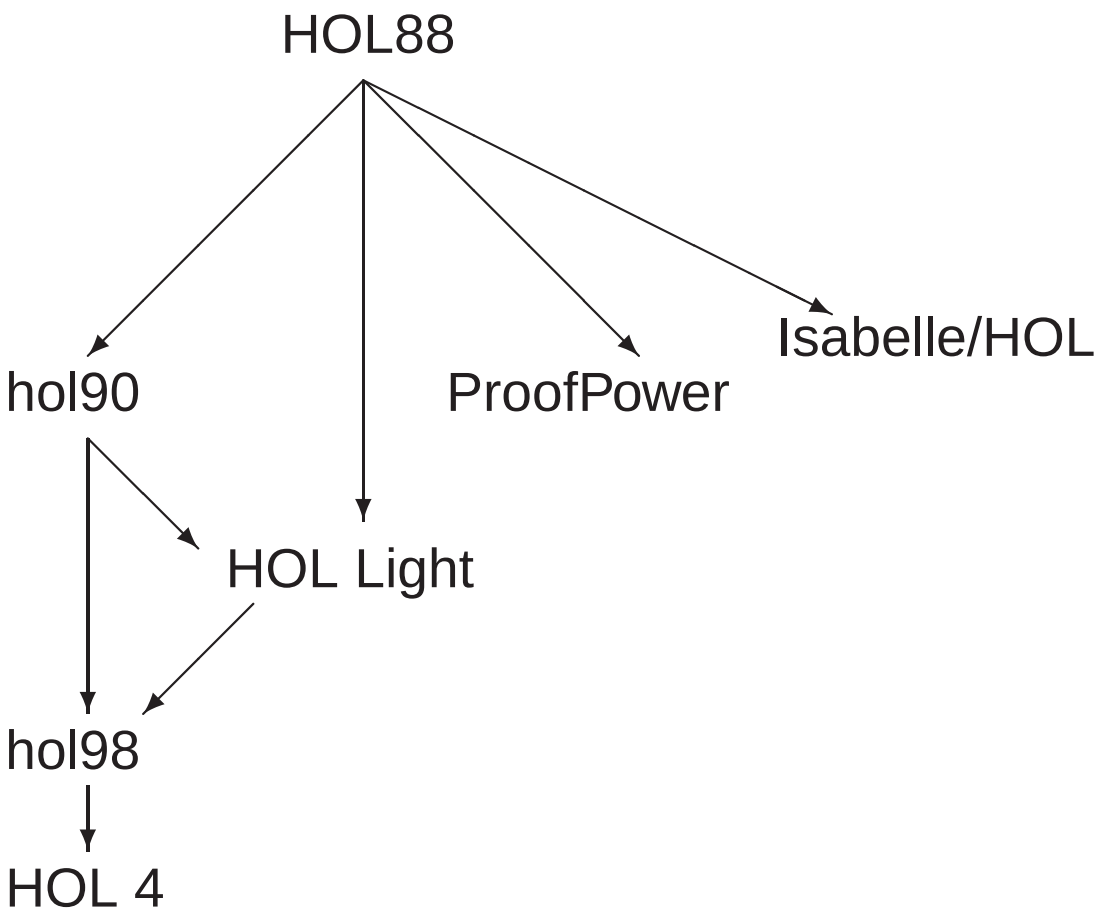
HOL Light is a member of the HOL family of provers, descended from Mike Gordon's original HOL system developed in the 80s.

An LCF-style proof checker for classical higher-order logic built on top of (polymorphic) simply-typed λ-calculus.

HOL Light is designed to have a simple and clean logical foundation.

Written in Objective CAML (OCaml).

The HOL family DAG



Real analysis details

Real analysis is especially important in our applications

- 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

Formal real analysis 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) ⇒ (λx. g(f x)) 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 (1)

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
```

Division and square root

There are several different algorithms for division and square root, and which one is better is a fine choice.

- Digit-by-digit: analogous to pencil-and-paper algorithms but usually with quotient estimation and redundant digits (SRT, Ercegovac-Lang etc.)
- Multiplicative: get faster (e.g. quadratic) convergence by using multiplication, e.g. Newton-Raphson, Goldschmidt, power series.

The Intel® Itanium® architecture uses some interesting multiplicative algorithms relying *purely* on conventional floating-point operations.

Basic ideas due to Peter Markstein, first used in IBM Power series.

Correctness issues

Easy to get within a bit or so of the right answer.

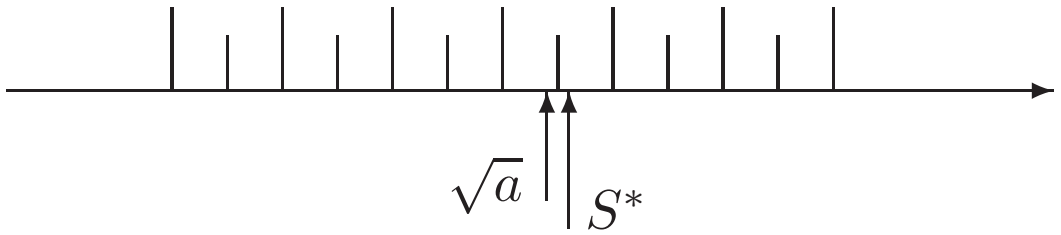
Meeting the correct rounding in the IEEE spec is significantly more challenging.

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^*.

Condition for perfect rounding (to nearest)

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:



Still seems complex to establish such properties.

Reduction to inequality reasoning

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
```

Square root exclusion zones

Around every midpoint m there *is* an 'exclusion zone' of width $\delta_m > 0$ within which the square root of a floating point number cannot occur.

However, can be as small as $\delta_m \approx |m|/2^{2p+2}$ where p is the floating-point precision.

Example: square root of significand that's all 1s.

Difficult cases

To ensure correct rounding, relative error before last rounding better 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.

We can bridge the gap by a technique due to Marius Cornea.

Mixed analytic-combinatorial proofs

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 check them explicitly.

Just like certain proofs in pure mathematics.

Isolating difficult cases

Difficult cases have significands m (considered as p-bit integers) such that for some k and small d:

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

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:

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

The procedure simply uses even-odd reasoning and recursion on the power of two 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. Always recurse down to an equation that is unsatisfiable or immediately solvable.

Transcendental functions: tangent algorithm

- 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.
- 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 $0 < |x| < \pi$):

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

is much harder than you might expect.

Conclusions

- Computer arithmetic is a difficult and subtle area where mistakes do occur.
- For verification, we need a general mathematical framework; finite-state techniques are of little help.
- For many applications, we need a proof system that is programmable.
- HOL Light and other well-developed LCF provers (Coq, HOL4, Isabelle, . . .) seem a good fit.