# Theorem Proving in Real Applications

**John Harrison**

**Intel Corporation**

- The cost of bugs
- Formal verification
- Levels of verification
- HOL Light
- Real analysis
- Formalizing floating-point arithmetic
- Examples illustrating claims
- Square root example
- Conclusions

# The cost of bugs

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

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 pre-silicon bugs in Intel processor designs per generation.
- Approximately 8000 bugs introduced during design of the Pentium 4.

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

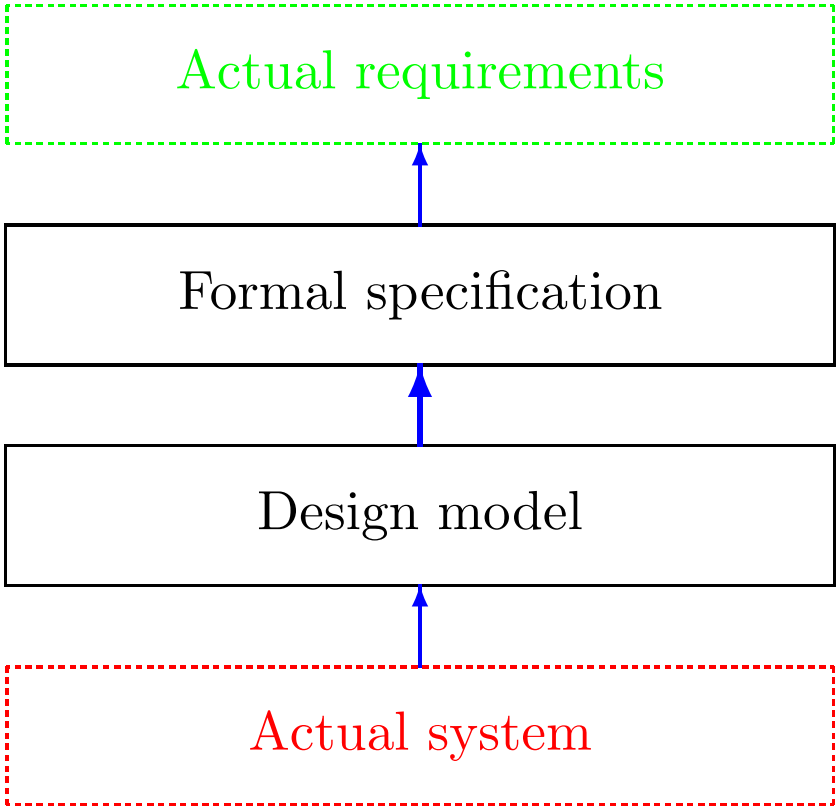
For example:

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

Formal verification offers a possible solution to the non-exhaustiveness problem.

# Formal verification

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



# Approaches to formal verification

There are three major approaches to formal verification, and Intel uses all of them, often in combination:

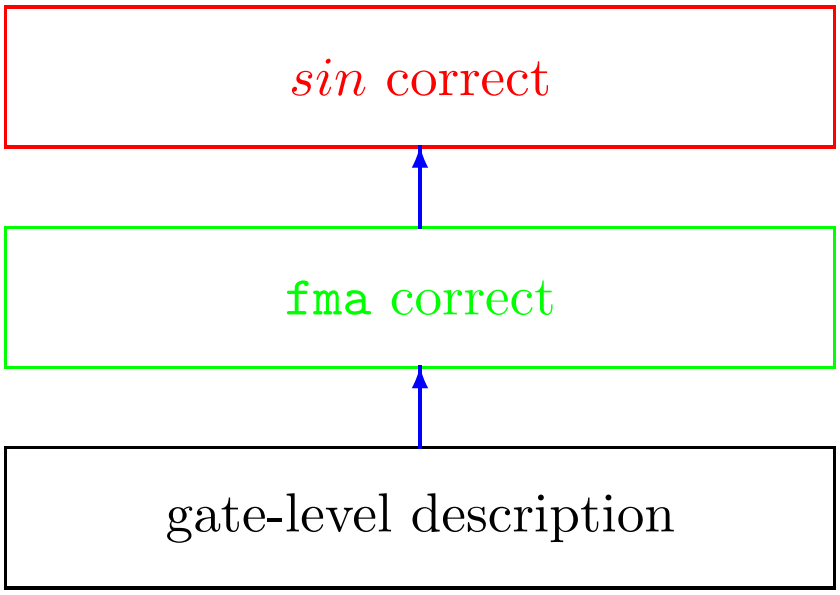
- Symbolic simulation
- Temporal logic model checking
- General theorem proving

One of the major tools used for hardware verification at Intel is a combined system.

As well as general theorem proving and traditional CTL and LTL model checking it supports *symbolic trajectory evaluation* (STE).

# Levels of verification

My job involves 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 (someone else's job).

# Context

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

Some similar work has been done for software libraries for the Intel Pentium® 4 processor.

Floating point algorithms for division, square root and transcendental functions are used for:

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

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

# Theorem proving infrastructure

What do we need to formally verify such mathematical software?

- Theorems about basic real analysis and properties of the transcendental functions, and even bits of number theory.
- Theorems about special properties of floating point numbers, floating point rounding etc.
- Automation of as much tedious reasoning as possible.
- Programmability of special-purpose inference routines.
- A flexible framework in which these components can be developed and applied in a reliable way.

We use the HOL Light theorem prover. Other possibilities would include PVS and maybe ACL2.

# Quick introduction to HOL Light

HOL Light is a member of the family of HOL theorem provers, demonstrated at FMCAD'96.

- 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`

# HOL real analysis theory

- 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
```

# Using real analysis

Just to say what mathematical functions like *sin*, *log* etc. *are*, and prove their basic properties, requires a fair amount of real analysis. For example, we use simple identities like:

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

At their core many algorithms use power series, and to justify these, we need the typical Taylor or Laurent series for the basic functions. For example:

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

This one is harder to prove than you might think (see *Proofs from the Book*).

# Using number theory

Trigonometric range reduction requires an analysis of how close a nonzero floating-point number can be to a multiple of $\frac{\pi}{2}$.

We formalize the proof that *convergents* to a real number $x$, i.e. rationals $p_1/q_1 < x < p_2/q_2$ with $p_2 q_1 = p_1 q_2 + 1$, are the best possible approximation without having a larger denominator.

```
|- (p2 * q1 = p1 * q2 + 1) /\
   (&p1 / &q1 < x /\ x < &p2 / &q2)
   ==> !b. ~(b = 0) /\ b < q1 /\ b < q2
           ==> abs(&a / &b - x)
                 > &1 / &(q1 * q2)
```

We can use such convergents to find the minimal distance between a nonzero floating-point number and a multiple of $\frac{\pi}{2}$.

# 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 of our 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
```

# The need for automation

Linear arithmetic, e.g. in analyzing properties of floating-point rounding.

```
REAL_ARITH
  `a <= x /\ b <= y /\
   abs(x - y) < abs(x - a) /\
   abs(x - y) < abs(x - b) /\
   (b <= x ==> abs(x - a) <= abs(x - b)) /\
   (a <= y ==> abs(y - b) <= abs(y - a))
   ==> (a = b)`;;
```

First order logic, e.g. proving basic logical lemmas or filling in boring details.

```
let sym_lemma = prove
 (`(!m n. P m n ==> P n m)
   ==> ((!m n. P m n) =
        (!m n. m <= n ==> P m n))`,
  MESON_TAC[LE_CASES]);;
```

# The need for programmability

There's really no chance that a theorem prover designer will build in all the special proof procedures needed, so the user must be able to program the system (without compromising soundness). For example:

- Automatically evaluate value of specific floating-point encoding into its real value
- Automatically prove that a particular power series approximation to a transcendental function is accurate to a given $\epsilon$ over a given $[a, b]$.

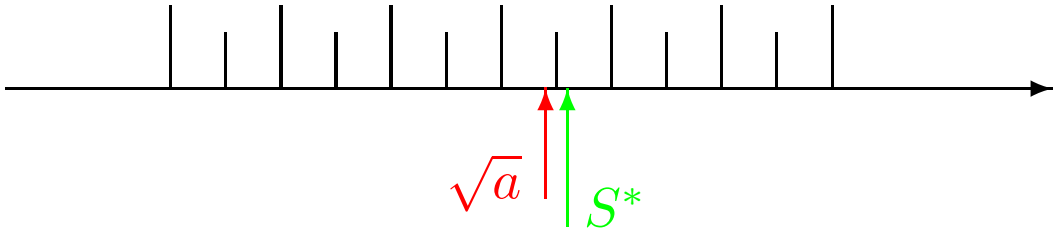
The latter is much harder, but is important since most transcendentals are approximated by Remez minimax approximations, not the Taylor series. Besides, the coefficients are floating-point numbers.

# Square root example

| | | |
|---|---|---|
| 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 |

# Square root perfect rounding

Several square root algorithms work by a final rounding of a more accurate intermediate result $S^*$. For perfect rounding, we should 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 $\delta$ 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 (`fma`) 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.

To obtain the complete result, we isolate all special cases and explicitly "run" the algorithm on them inside the logic.

# 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 $\leq 10$, 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:

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

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 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 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
- Etc.

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.