# Formal Verification of Mathematical Algorithms

**John Harrison**

**Intel Corporation**

- The cost of bugs
- Formal verification
- Levels of verification
- HOL Light
- Formalizing mathematics
- Applications
- 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 well over 99.5%.

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

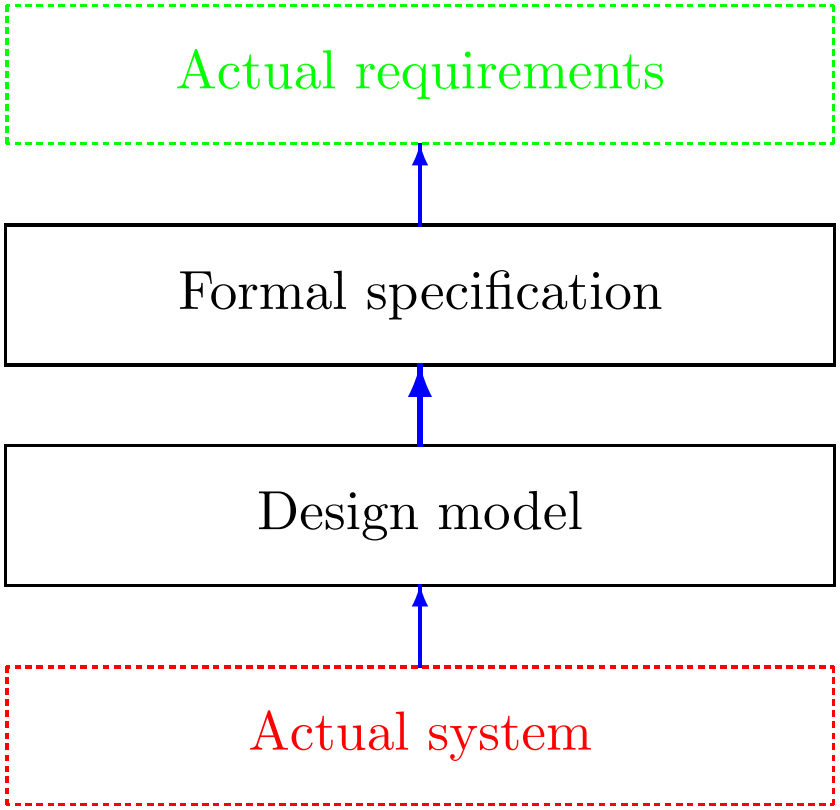
For example:

- $2^{160}$ 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*.



# Formal verification is hard

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

- Must specify intended behavior formally
- Need to make many hidden assumptions explicit
- Requires long detailed proofs, difficult to review

The state of the art is quite limited.

Software verification has been around since the 60s, but there have been few major successes.

# 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.

# Approaches to formal verification

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

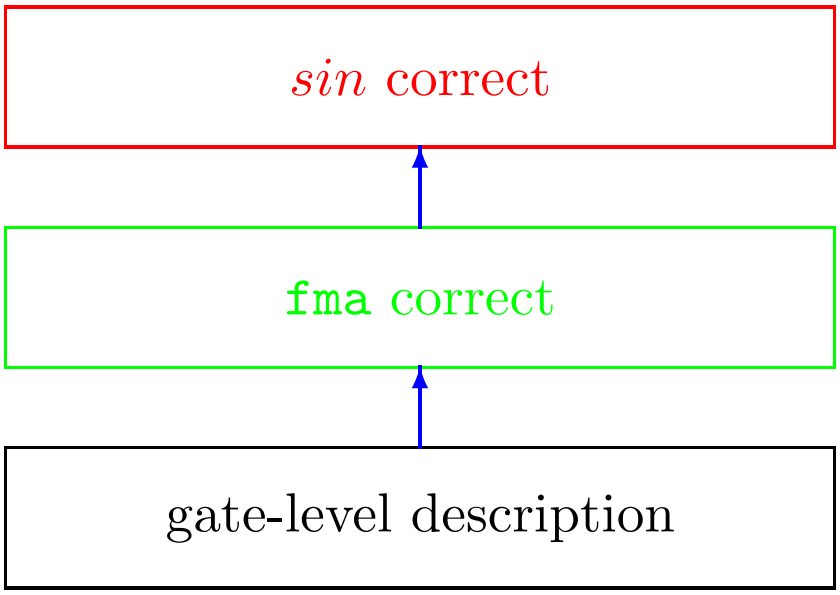
- Symbolic simulation
- Temporal logic model checking
- General theorem proving

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 large family of HOL theorem provers.

- An LCF-style programmable proof checker written in CAML Light / OCaml, 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 Light primitive rules (1)

$$\frac{}{\vdash t = t}\ \texttt{REFL}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\ \texttt{TRANS}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash s(u) = t(v)}\ \texttt{MK\_COMB}$$

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash (\lambda x.\ s) = (\lambda x.\ t)}\ \texttt{ABS}$$

$$\frac{}{\vdash (\lambda x.\ t)x = t}\ \texttt{BETA}$$

# HOL Light primitive rules (2)

$$\frac{}{\{p\} \vdash p}\ \texttt{ASSUME}$$

$$\frac{\Gamma \vdash p = q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}\ \texttt{EQ\_MP}$$

$$\frac{\Gamma \vdash p \quad \Delta \vdash q}{(\Gamma - \{q\}) \cup (\Delta - \{p\}) \vdash p = q}\ \texttt{DEDUCT\_ANTISYM\_RULE}$$

$$\frac{\Gamma[x_1, \ldots, x_n] \vdash p[x_1, \ldots, x_n]}{\Gamma[t_1, \ldots, t_n] \vdash p[t_1, \ldots, t_n]}\ \texttt{INST}$$

$$\frac{\Gamma[\alpha_1, \ldots, \alpha_n] \vdash p[\alpha_1, \ldots, \alpha_n]}{\Gamma[\gamma_1, \ldots, \gamma_n] \vdash p[\gamma_1, \ldots, \gamma_n]}\ \texttt{INST\_TYPE}$$

# Formalized mathematics

Our work involves the *actual* formalization of mathematics in a simple logical proof system. (Not just formalization-in-principle.)

In the same spirit as the work of many logical pioneers (Frege, Peano, Russell and Whitehead).

The aim is the same: precision in assertions and reliability of proofs.

Arguably, formal proofs written out by people would *not* be more reliable than informal proofs — probably quite the reverse.

In fact, the proofs we do sometimes involve $\approx 10^8$ primitive inferences — very difficult for people to do at all!

But computers are very good at applying formal rules efficiently and without error, so we really do get a dramatic improvement in reliability.

# Applying formal real analysis

We've formalized a definitional construction of the real numbers, and the development on top of it of basic real analysis (limits, series, differentiation, power series, ...). Used:

- to prove basic identities used in computation

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

- to verify Taylor or Laurent expansions for functions with convergence criteria:

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

- and to prove that particular minimax polynomials really are approximations to a given precision.

# Applying formal number theory

Sometimes we need to employ a little number theory too:

- Initial trigonometric range reduction $r = x - N \cdot \pi/2$ needs to be done with a relatively involved algorithm. To justify it, we need to analyze how close a floating-point number can be to an integral multiple of $\pi/2$, a classic problem in Diophantine approximation solvable using convergents.
- Analytical proof of correctness of some square root algorithms excludes special cases that can be characterized as the solution of Diophantine equations of the form $2^p m = k^2 + d$. We need to enumerate a provably exhaustive set of $k$ and $m$ for given $p$ and $d$.

# Programmability

Note that for some applications, programmability of the theorem prover is practically essential.

The structure of the proof is essentially predictable, and we can program up a general procedure, whereas chreographing various instances by hand would be almost unbearably tedious.

- Can bound polynomial approximation error using a more accurate Taylor series and a recursive procedure based on recursive root isolation and bounding of all the derivatives of the difference polynomial.
- Can solve Diophantine equations using even-odd case analysis to reduce $p$, and proceed by recursion.

This is something HOL supports well, given its implementation within a general purpose programming language.

# 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.

Could be said to realize, and extend, the work of the logical pioneers in actual formalization.