Formal Verification at Intel

John Harrison

Intel Corporation

- The cost of bugs
- Testing and formal verification
- Automated and general methods
- HOL Light
- Floating point verification
- Tangent example
- 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 bugs in Intel processor designs per generation.
- Approximately 8000 bugs designed into the Pentium 4 (‘Willamette’).

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.

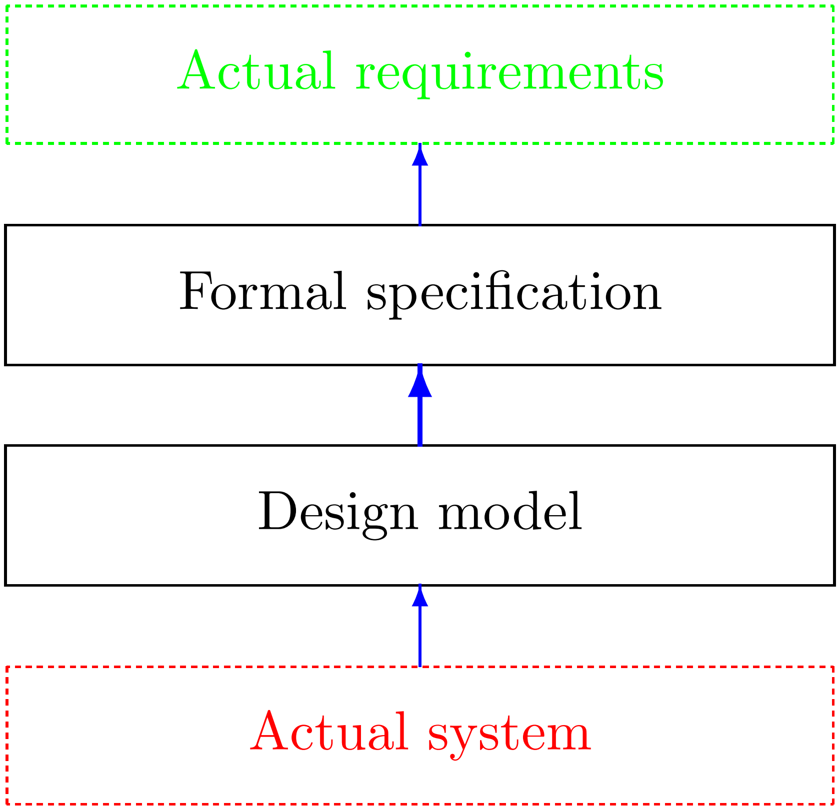
- Slow — especially pre-silicon
- Too many possibilities to test them all

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

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?

Analogy with mathematics

Sometimes even a huge weight of empirical evidence can be misleading.

- $\pi(n)$ = number of primes $\leq n$
- $li(n) = \int_0^n du/ln(u)$

Littlewood proved in 1914 that $\pi(n) - li(n)$ changes sign infinitely often.

No change of sign at all had ever been found despite testing up to $n = 10^{10}$ (in the days before computers).

Similarly, extensive testing of hardware or software may still miss errors that would be revealed by a formal proof.

Formal verification is hard

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

- Must specify intended behaviour 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.

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.

Automatic verification?

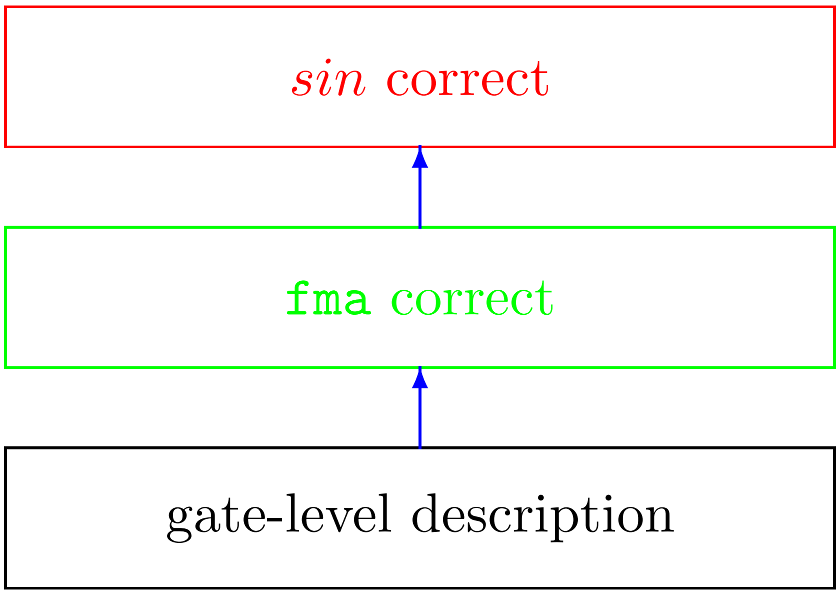
Many problems can be attacked using decision methods with (in principle!) limited human intervention, e.g.

- Boolean equivalence checking
- Temporal logic model checking
- Symbolic trajectory evaluation

However, sometimes we need more general theorem proving, especially for the kinds of applications I'm interested in...

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).

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 based on the approach to theorem proving pioneered in Edinburgh LCF in the 70s.

- 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.

The system can be extended reliably without making unsafe modifications

The user controls the means of production (of theorems).

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

Existing 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
```

HOL floating point theory

Generic floating point theory in HOL.

Can be applied to all the required formats, and others supported in software.

Precise specification of floating point rounding, floating point exceptions etc. Typical theorems include monotonicity of rounding:

```
 |- ~(precision fmt = 0) /\ x <= y
    ==> round fmt rc x <= round fmt rc y
```

and subtraction of nearby floating point numbers:

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

Example: tangent algorithm

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 $\pi/2$?

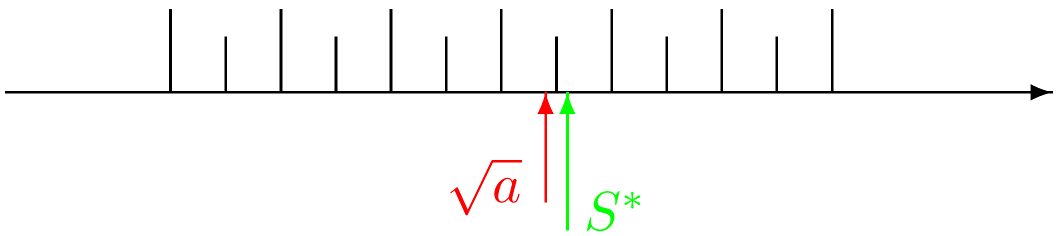
Even deriving the power series (for $0 < |x| < \pi$):

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

is much harder than you might expect.

Square root example

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 δ can be quite small, considered as a relative error. If the floating point format has precision p, then we can have $\delta_m \approx |m|/2^{2p+3}$.

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 number-theoretically, and explicitly “run” the algorithm on them inside the logic.

This approach is due to Marius Cornea, and is especially amenable to semi-automated formalization.

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 ≤ 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:

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

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

- Formal verification is industrially important.
- For high-level algorithms we need a general theorem prover and formalized mathematics.
- A large part of the work involves building up general theories about both pure mathematics and special properties of floating point numbers.
- It is easy to underestimate the amount of pure mathematics needed for obtaining very practical results.
- The mathematics required is often the sort that is not found in current textbooks: very concrete results but with a proof!
- Using HOL Light, we can confidently integrate all the different aspects of the proof, using programmability to automate tedious parts.