Floating-Point Verification

John Harrison
Intel Corporation

FM Industry Day

Newcastle

20 July 2005

Famous floating-point bugs

Bugs in computer systems are, unfortunately, a fact of life. Two fairly well-known examples involve floating-point arithmetic:

- Incorrect division in early Intel®Pentium® processor.
- Failure of the Ariane rocket due to untrapped floating-point exception

Intel wrote off $475M to cover the FDIV bug, and suffered considerable damage to its reputation.

A similar error today could be much more expensive.

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 over last 3 generations.
- Approximately 8000 bugs introduced during design of the Pentium 4.

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

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.

So it's natural that we turn to formal verification methods.

Formal verification in industry

Formal verification is increasingly becoming standard practice in the hardware industry.

- 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

Nevertheless, increasing interest in advanced static checkers like SLAM incorporating theorem proving.

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

Theorem provers

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.

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 have formally verified correctness of various floating-point algorithms designed for the Intel® Itanium® architecture.

- Division and square root (Marstein-style, using fused multiply-add to do Newton-Raphson or power series approximation with delicate final rounding).
- Transcendental functions like log and sin (table-driven algorithms using range reduction and a core polynomial approximations).

Proofs use the HOL Light prover

- `http://www.cl.cam.ac.uk/users/jrh/hol-light`

Our HOL Light proofs

The mathematics we formalize is mostly:

- Elementary number theory and real analysis
- Floating-point numbers, results about rounding etc.

Needs several special-purpose proof procedures, e.g.

- Verifying solution set of some quadratic congruences
- Proving primality of particular numbers
- Proving bounds on rational approximations
- Verifying errors in polynomial approximations

Example: 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.

The value of formal verification

The formal verifications we undertook had a number of beneficial consequences

- Uncovered several bugs
- Revealed ways that algorithms could be made more efficient
- Improved our confidence in the (original or final) algorithms
- Led to deeper theoretical understanding

This experience seems quite common.