# Formal verification of floating-point arithmetic at Intel

John Harrison

Intel Corporation

6 June 2012

# Summary

- Some notable computer arithmetic failures

# Summary

- Some notable computer arithmetic failures
- Formal verification, testing and models

# Summary

- Some notable computer arithmetic failures
- Formal verification, testing and models
- Formal verification techniques at Intel

# Summary

- Some notable computer arithmetic failures
- Formal verification, testing and models
- Formal verification techniques at Intel
- Perspectives and future prospects

# Ariane rocket failure

In 1996, the Ariane 5 rocket on its maiden flight was destroyed; the rocket and its cargo were estimated to be worth \$500$M$.

# Ariane rocket failure

In 1996, the Ariane 5 rocket on its maiden flight was destroyed; the rocket and its cargo were estimated to be worth \$500$M$.

- Cause was an uncaught floating-point exception

# Ariane rocket failure

In 1996, the Ariane 5 rocket on its maiden flight was destroyed; the rocket and its cargo were estimated to be worth \$500$M$.

- Cause was an uncaught floating-point exception
- A 64-bit floating-point number representing horizontal velocity was converted to a 16-bit integer

# Ariane rocket failure

In 1996, the Ariane 5 rocket on its maiden flight was destroyed; the rocket and its cargo were estimated to be worth \$500$M$.

- Cause was an uncaught floating-point exception
- A 64-bit floating-point number representing horizontal velocity was converted to a 16-bit integer
- The number was larger than $2^{15}$.

# Ariane rocket failure

In 1996, the Ariane 5 rocket on its maiden flight was destroyed; the rocket and its cargo were estimated to be worth \$500$M$.

- Cause was an uncaught floating-point exception
- A 64-bit floating-point number representing horizontal velocity was converted to a 16-bit integer
- The number was larger than $2^{15}$.
- As a result, the conversion failed.

# Ariane rocket failure

In 1996, the Ariane 5 rocket on its maiden flight was destroyed; the rocket and its cargo were estimated to be worth $\$500M$.

- Cause was an uncaught floating-point exception
- A 64-bit floating-point number representing horizontal velocity was converted to a 16-bit integer
- The number was larger than $2^{15}$.
- As a result, the conversion failed.
- The rocket veered off its flight path and exploded, just 40 seconds into the flight sequence.

# Patriot missile failure

During the first Gulf War in 1991, 28 soldiers were killed when a Scud missile struck an army barracks.

# Patriot missile failure

During the first Gulf War in 1991, 28 soldiers were killed when a Scud missile struck an army barracks.

- Patriot missile failed to intercept the Scud

# Patriot missile failure

During the first Gulf War in 1991, 28 soldiers were killed when a Scud missile struck an army barracks.

- Patriot missile failed to intercept the Scud
- Underlying cause was a computer arithmetic error in computing time since boot

# Patriot missile failure

During the first Gulf War in 1991, 28 soldiers were killed when a Scud missile struck an army barracks.

- Patriot missile failed to intercept the Scud
- Underlying cause was a computer arithmetic error in computing time since boot
- Internal clock was multiplied by $\frac{1}{10}$ to produce time in seconds

# Patriot missile failure

During the first Gulf War in 1991, 28 soldiers were killed when a Scud missile struck an army barracks.

- Patriot missile failed to intercept the Scud
- Underlying cause was a computer arithmetic error in computing time since boot
- Internal clock was multiplied by $\frac{1}{10}$ to produce time in seconds
- Actually performed by multiplying 24-bit approximation of $\frac{1}{10}$

# Patriot missile failure

During the first Gulf War in 1991, 28 soldiers were killed when a Scud missile struck an army barracks.

- Patriot missile failed to intercept the Scud
- Underlying cause was a computer arithmetic error in computing time since boot
- Internal clock was multiplied by $\frac{1}{10}$ to produce time in seconds
- Actually performed by multiplying 24-bit approximation of $\frac{1}{10}$
- Net error after 100 hours about 0.34 seconds.

# Patriot missile failure

During the first Gulf War in 1991, 28 soldiers were killed when a Scud missile struck an army barracks.

- Patriot missile failed to intercept the Scud
- Underlying cause was a computer arithmetic error in computing time since boot
- Internal clock was multiplied by $\frac{1}{10}$ to produce time in seconds
- Actually performed by multiplying 24-bit approximation of $\frac{1}{10}$
- Net error after 100 hours about 0.34 seconds.
- A Scud missile travels $500m$ in that time

# Intel's FDIV bug

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

# Intel's FDIV bug

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's FDIV bug

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.

# Limits of testing

Bugs are usually detected by extensive testing, including pre-silicon simulation.

# Limits of testing

Bugs are usually detected by extensive testing, including pre-silicon simulation.

- Slow — especially pre-silicon

# 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

# 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).

# Limits of testing

Bugs are usually detected by extensive testing, including pre-silicon simulation.

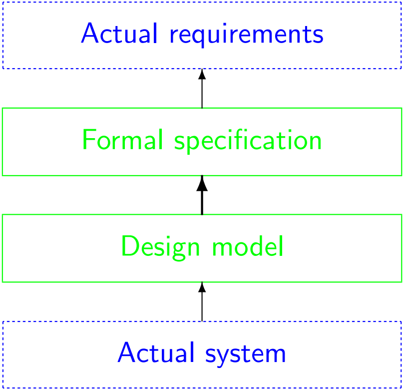
- Slow — especially pre-silicon
- Too many possibilities to test them all

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*.



Actual requirements

Formal specification

Design model

Actual system

# Analogy with mathematics

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

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

# 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.

# 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).

# 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.

# Verification vs. testing

Verification has some advantages over testing:

# Verification vs. testing

Verification has some advantages over testing:

- Exhaustive.
- Improves our intellectual grasp of the system.

# Verification vs. testing

Verification has some advantages over testing:

- Exhaustive.
- Improves our intellectual grasp of the system.

However:

# Verification vs. testing

Verification has some advantages over testing:

- Exhaustive.
- Improves our intellectual grasp of the system.

However:

- Difficult and time-consuming.

# 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.

# 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?

# 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.

# Models versus the real world

Chips can suffer from physical problems, usually due to overheating or particle bombardment (‘soft errors’).

# Models versus the real world

Chips can suffer from physical problems, usually due to overheating or particle bombardment (‘soft errors’).

- In 1978, Intel encountered problems with ‘soft errors’ in some of its DRAM chips.

# Models versus the real world

Chips can suffer from physical problems, usually due to overheating or particle bombardment (‘soft errors’).

- In 1978, Intel encountered problems with ‘soft errors’ in some of its DRAM chips.
- The cause turned out to be alpha particle emission from the packaging.

# Models versus the real world

Chips can suffer from physical problems, usually due to overheating or particle bombardment (‘soft errors’).

- In 1978, Intel encountered problems with ‘soft errors’ in some of its DRAM chips.
- The cause turned out to be alpha particle emission from the packaging.
- The factory producing the ceramic packaging was on the Green River in Colorado, downstream from the tailings of an old uranium mine.

# Models versus the real world

Chips can suffer from physical problems, usually due to overheating or particle bombardment (‘soft errors’).

- In 1978, Intel encountered problems with ‘soft errors’ in some of its DRAM chips.
- The cause turned out to be alpha particle emission from the packaging.
- The factory producing the ceramic packaging was on the Green River in Colorado, downstream from the tailings of an old uranium mine.

However, these are rare and apparently well controlled by existing engineering best practice.

# 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.

# 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.

# 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.

# 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.

# 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 often necessary.

# A variety of verification methods

There is a diverse world of formal verification methods, trading automation for generality / efficiency, most of which are in active use at Intel.

- Propositional tautology/equivalence checking (FEV)
- Symbolic simulation
- Symbolic trajectory evaluation (STE)
- Temporal logic model checking
- Combined decision procedures (SMT)
- First order automated theorem proving
- Interactive theorem proving

# A spectrum of formal techniques

Traditionally, formal verification has been focused on complete proofs of functional correctness.

But recently there have been notable successes elsewhere for ‘semi-formal’ methods involving abstraction or more limited property checking.

- Airbus A380 avionics
- Microsoft SLAM/SDV

One can also consider applying theorem proving technology to support testing or other traditional validation methods like path coverage.

These are all areas of interest at Intel.

# Our work

We have formally verified correctness of various floating-point algorithms.

- 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.

# 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.

# 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

# 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

# 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

# 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.

# 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.

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

# 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:

# Overview of the verification

To verify this algorithm, we need to prove:

- The range reduction to obtain $r$ is done accurately.

# 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.

# 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.

# 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.

# 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.

# 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.

# Why HOL Light?

We need a general theorem proving system with:

- High standard of logical rigor and reliability
- Ability to mix interactive and automated proof
- Programmability for domain-specific proof tasks
- A substantial library of pre-proved mathematics

Other theorem provers such as ACL2, Coq and PVS have also been used for verification in this area.

# The value of formal verification

Formal verification has contributed in many ways, and not only the obvious ones:

# The value of formal verification

Formal verification has contributed in many ways, and not only the obvious ones:

- Uncovered bugs, including subtle and sometimes very serious ones

# The value of formal verification

Formal verification has contributed in many ways, and not only the obvious ones:

- Uncovered bugs, including subtle and sometimes very serious ones
- Revealed ways that algorithms could be made more efficient

# The value of formal verification

Formal verification has contributed in many ways, and not only the obvious ones:

- Uncovered bugs, including subtle and sometimes very serious ones
- Revealed ways that algorithms could be made more efficient
- Improved our confidence in the (original or final) product

# The value of formal verification

Formal verification has contributed in many ways, and not only the obvious ones:

- Uncovered bugs, including subtle and sometimes very serious ones
- Revealed ways that algorithms could be made more efficient
- Improved our confidence in the (original or final) product
- Led to deeper theoretical understanding

# The value of formal verification

Formal verification has contributed in many ways, and not only the obvious ones:

- Uncovered bugs, including subtle and sometimes very serious ones
- Revealed ways that algorithms could be made more efficient
- Improved our confidence in the (original or final) product
- Led to deeper theoretical understanding

This experience seems quite common.

# Perspectives and future prospects

Formal verification is an important method for giving the highest levels of assurance. But it is so difficult that is only deployed on a relatively small number of critical components.

# Perspectives and future prospects

Formal verification is an important method for giving the highest levels of assurance. But it is so difficult that is only deployed on a relatively small number of critical components.

- We need more research on making formal verification more efficient and automatic so it can be applied more widely, and applied by relative non-experts.

# Perspectives and future prospects

Formal verification is an important method for giving the highest levels of assurance. But it is so difficult that is only deployed on a relatively small number of critical components.

- We need more research on making formal verification more efficient and automatic so it can be applied more widely, and applied by relative non-experts.
- We need computer science curricula at universities to provide more rigorous treatment of mathematical rigor, logic and formal proof so that more programmers and engineers are able to deploy formal techniques.