Formal Methods at Intel — An Overview

John Harrison

Intel Corporation

Second NASA Formal Methods Symposium

NASA HQ, Washington DC

14th April 2010 (09:00–10:00)

Table of contents

Overview

A diversity of activities

Intel is best known as a hardware company, and hardware is still the core of the company's business. However this entails much more:

- Microcode
- Firmware
- Protocols
- Software

A diversity of activities

Intel is best known as a hardware company, and hardware is still the core of the company's business. However this entails much more:

- Microcode
- Firmware
- Protocols
- Software

If the Intel® Software and Services Group (SSG) were split off as a separate company, it would be in the top 10 software companies worldwide.

A diversity of verification problems

This gives rise to a corresponding diversity of verification problems, and of verification solutions.

- 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

Most of these techniques (trading automation for generality / efficiency) are in active use at Intel.

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.

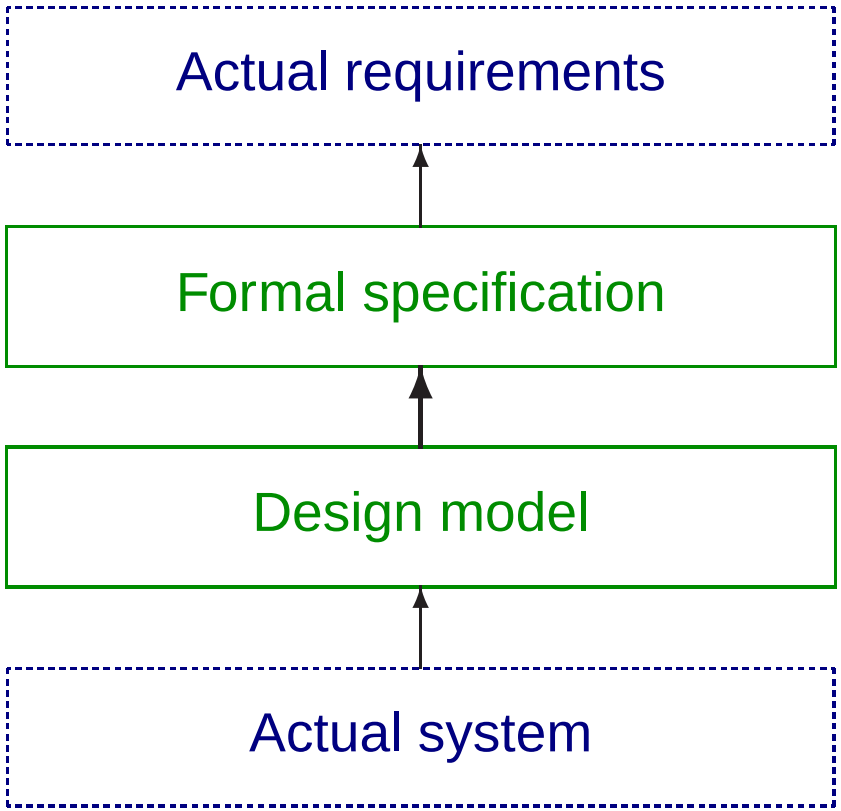
- Airbus A380 avionics
- Microsoft SLAM/SDV

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.

Models and their validation

We have the usual concerns about validating our specs, but also need to pay attention to the correspondence between our models and physical reality.



Physical problems

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.

Physical problems

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.

Physical problems

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.

The FDIV bug

Formal methods are more useful for avoiding design errors such as the infamous FDIV bug:

- 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.

This did at least considerably improve investment in formal verification.

Layers of verification

If we want to verify from the level of software down to the transistors, then it's useful to identify and specify intermediate layers.

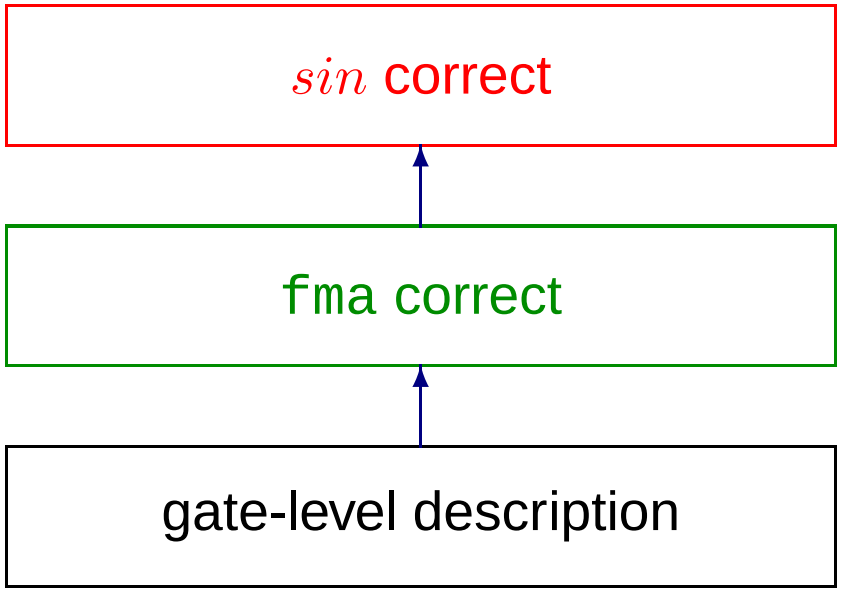
- Implement high-level floating-point algorithm assuming addition works correctly.
- Implement a cache coherence protocol assuming that the abstract protocol ensures coherence.

Many similar ideas all over computing: protocol stack, virtual machines etc.

If this clean separation starts to break down, we may face much worse verification problems. . .

How some of our verifications fit together

For example, the `fma` behavior is the *assumption* for my verification, and the *conclusion* for someone else's.



But this is not quite trivial when the verifications use different formalisms!

I: Hardware with SAT and STE

O'Leary, Zhao, Gerth, Seger, *Formally verifying IEEE compliance of floating-point hardware*, ITJ 1999.

Yang and Seger, *Introduction to Generalized Symbolic Trajectory Evaluation*, FMCAD 2002.

Schubert, *High level formal verification of next-generation microprocessors*, DAC 2003.

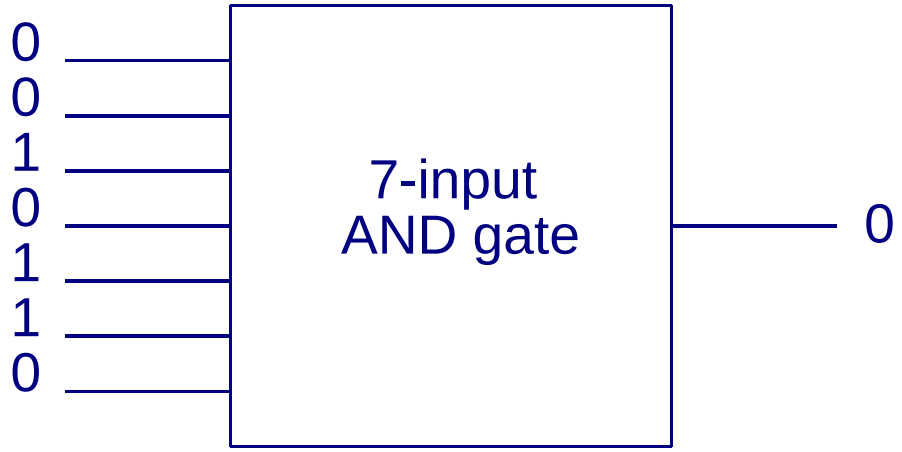
Slobodova, *Challenges for Formal Verification in Industrial Setting*, FMCAD 2007.

Kaivola et al., *Replacing Testing with Formal Verification in Intel® Core^{TM} i7 Processor Execution Engine Validation*, CAV 2009.

Simulation

The traditional method for testing and debugging hardware designs is *simulation*.

This is just testing, done on a formal circuit model.



Feed sets of arguments in as inputs, and check whether the output is as expected.

Generalizations of simulation

We can generalize basic simulation in two different ways:

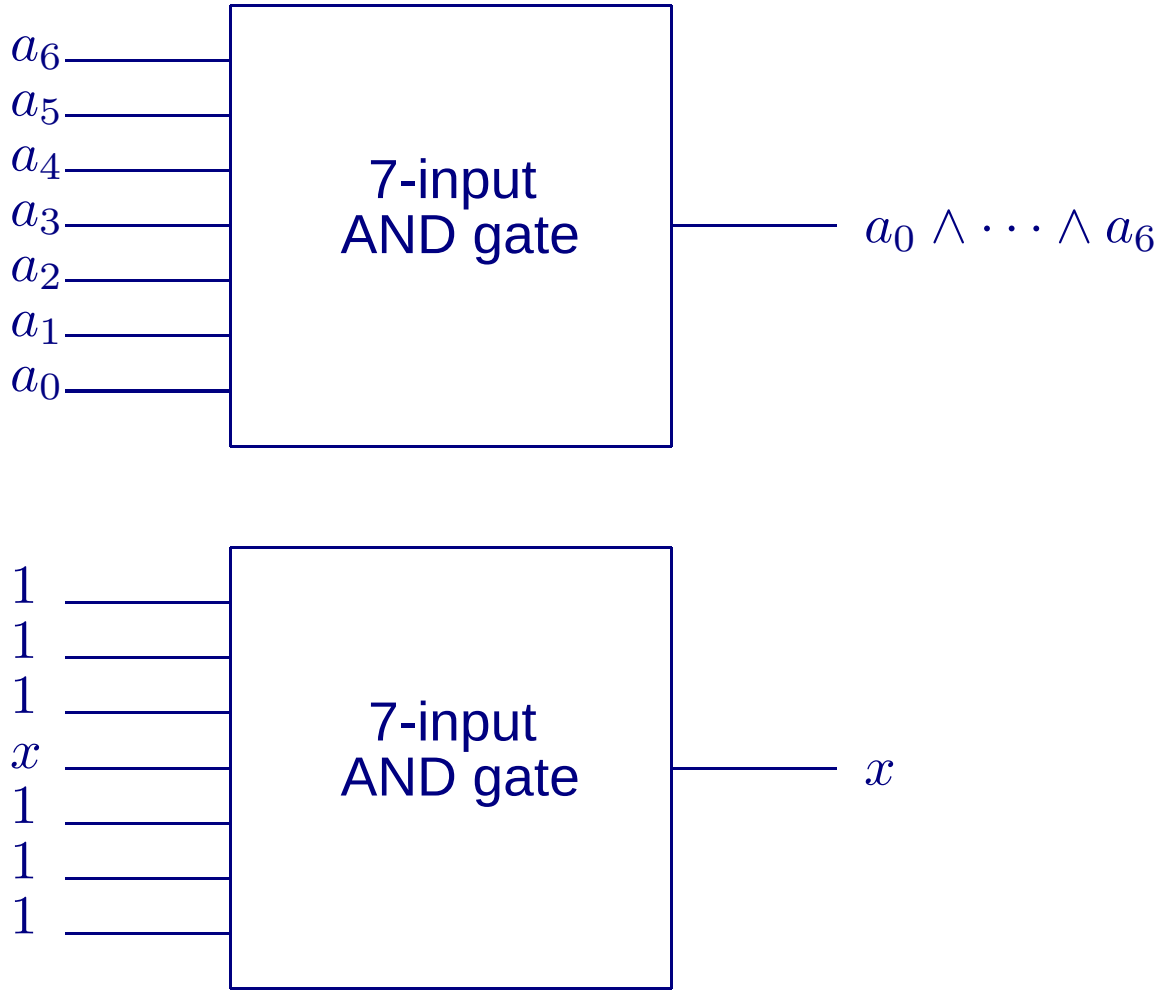
- *Ternary* simulation, where as well as 0 and 1 we have a “don’t care” value X.
- *Symbolic* simulation, where inputs may be parametrized by Boolean variables, and outputs are functions of those variables (usually represented as BDDs).

Rather surprisingly, it’s especially useful to do both at the same time, and have ternary values parametrized by Boolean variables.

This leads on to *symbolic trajectory evaluation* (STE) and its generalizations.

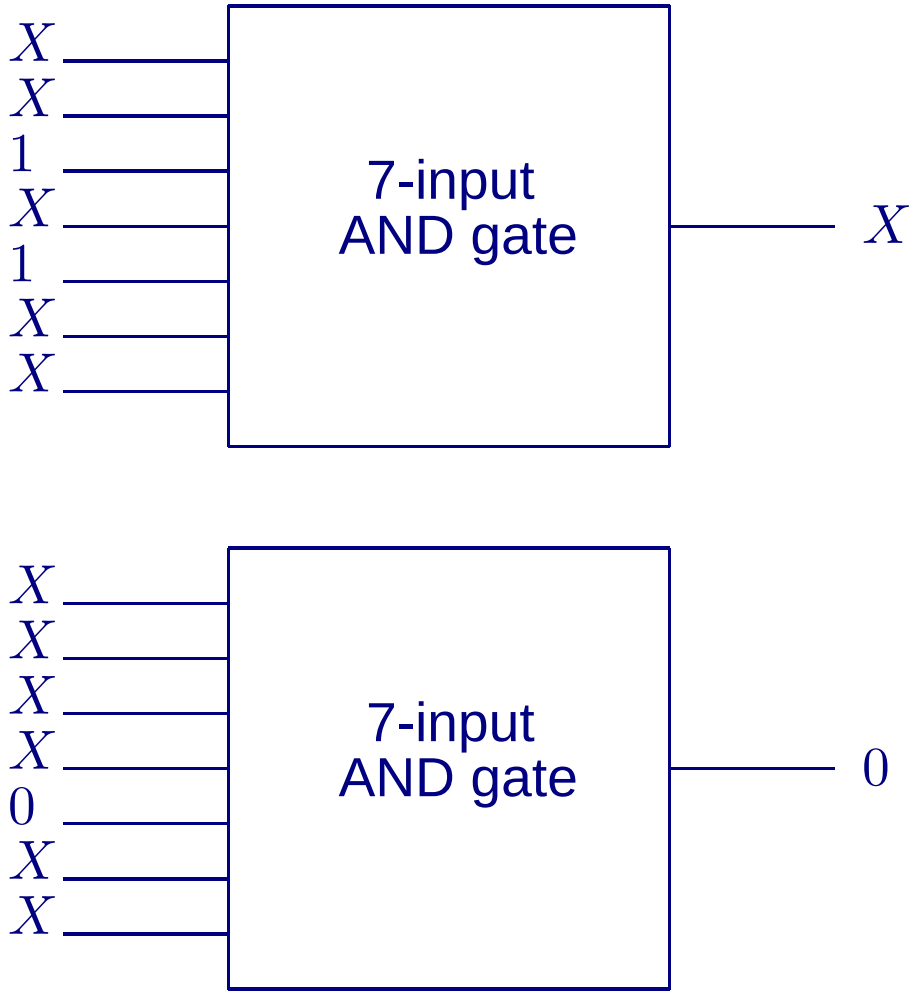
Example of symbolic simulation

We might use Boolean variables for all, or just some, inputs:



Example of ternary simulation

If some inputs are undefined, the output often is too, but not always:



Economies

Consider the 7-input AND gate. To verify it exhaustively:

- In conventional simulation, we would need 128 test cases, 0000000, 0000001, ..., 1111111.
- In symbolic simulation, we only need 1 symbolic test case, $a_0a_1a_2a_3a_4a_5a_6$, but need to manipulate expressions, not just constants.
- In ternary simulation, we need 8 test cases, $XXXXXX0$, $XXXXX0X$, ..., $0XXXXXX$ and 1111111.

If we combine symbolic and ternary simulation, we can *parametrize* the 8 test cases by just 3 Boolean variables.

This makes the manipulation of expressions much more economical.

Quaternary simulation

It's theoretically convenient to generalize ternary to *quaternary* simulation, introducing an 'overconstrained' value T.

We can think of each quaternary value as standing for a set of possible values:

$$\begin{aligned} T &= \{\} \\ 0 &= \{0\} \\ 1 &= \{1\} \\ X &= \{0, 1\} \end{aligned}$$

This is essentially a simple case of an abstraction mapping, and we can think of the abstract values partially ordered by information.

Extended truth tables

The truth-tables for basic gates are extended:

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
X	X	X	X	X	X
X	0	0	X	X	X
X	1	X	1	1	X
0	X	0	X	1	X
0	0	0	0	1	1
0	1	0	1	1	0
1	X	X	1	X	X
1	0	0	1	0	0
1	1	1	1	1	1

Composing gates in this simple way, we may lose information.

Symbolic trajectory evaluation

Symbolic trajectory evaluation (STE) is a further development of ternary symbolic simulation.

The user can write specifications in a restricted *temporal logic*, specifying the behavior over bounded-length *trajectories* (sequences of circuit states).

A typical specification would be: if the current state satisfies a property P, then after n time steps, the state will satisfy the property Q.

The circuit can then be checked against this specification by symbolic quaternary simulation.

STE plus theorem proving

STE (sometimes its extension GSTE) is the basic hardware verification workhorse at Intel

However, it often needs to be combined with theorem-proving for effective problem decomposition.

Intel has its own custom tool integrating lightweight theorem proving with STE, GSTE and other model checking engines.

This combination has been applied successfully to many hardware components, including floating-point units and many others.

II: Protocols with model checking and SMT

Chou, Mannava and Park: *A simple method for parameterized verification of cache coherence protocols*, FMCAD 2004.

Krstic, *Parametrized System Verification with Guard Strengthening and Parameter Abstraction*, AVIS 2005.

Talupur, Krstic, O'Leary and Tuttle, *Parametric Verification of Industrial Strength Cache Coherence Protocols*, DCC 2008.

Bingham, *Automatic non-interference lemmas for parameterized model checking*, FMCAD 2008.

Talupur and Tuttle, *Going with the Flow: Parameterized Verification Using Message Flows*, FMCAD 2008.

Parametrized systems

Important target for verification is *parametrized systems*.

N equivalent replicated components, so the state space involves some Cartesian product

$$\Sigma = \Sigma_0 \times \overbrace{\Sigma_1 \times \cdots \times \Sigma_1}^{N \text{ times}}$$

and the transition relation is symmetric between the replicated components.

Sometimes we have subtler symmetry, but we'll just consider full symmetry.

Multiprocessors with private cache

Example: multiprocessor where each processor has its own cache.

We have N cacheing agents with state space Σ_1 each, and maybe some special 'home node' with state space Σ_0.

We can consider Σ_1 as finite with two radical but not unreasonable simplifications:

- Assume all cache lines are independent (no resource allocation conflicts)
- Ignore actual data and consider only state of cache line (dirty, clean, whatever)

Coherence

The permitted transitions are constrained by a protocol designed to ensure that all caches have a coherent view of memory.

On some simplifying assumptions, we can express this adequately just using the cache states.

In classic MESI protocols, each cache can be in four states: `Modified`, `Exclusive`, `Shared` and `Invalid`.

Coherence means:

$$\forall \texttt{i. Cache(i) IN } \{\texttt{Modified, Exclusive}\} \Rightarrow \forall \texttt{j. } \neg(\texttt{j = i}) \Rightarrow \texttt{Cache(j) = Invalid}$$

Parametrized verification

For a specific N, the overall state space is finite.

We can specify the protocol and verify coherence using a traditional model checker (Murphi, SPIN, . . .).

This is already very useful and works well for small N. But:

- For a complex protocol, model checking may only be practical for very small N.
- In principle, the protocol is designed to work for *any* N, and we should like a general proof.

How can we do this?

Inductive proof

Find an inductive invariant I such that

$$I(\sigma) \wedge R(\sigma, \sigma') \Rightarrow I(\sigma')$$

The inductive invariant I is universally quantified, and occurs in both antecedent and consequent.

The transition relation has outer existential quantifiers $\exists i. \cdots$ because we have a symmetric choice between all components.

Inside, we may also have universal quantifiers if we choose to express array updates `a(i) := Something` as relations between functions:

$$a'(i) = \texttt{Something} \wedge \forall j. \neg(j = i) \Rightarrow a'(j) = a(j)$$

Our quantifier prefix

So our inductiveness claim may look like

$$(\forall i, j, \ldots. \cdots) \wedge (\exists i.\, \forall j.\, \cdots) \Rightarrow (\forall i, j, \ldots. \cdots)$$

If we put this into prenex normal form in the right way, the quantifier prefix is of the form $\forall \cdots \forall \exists \cdots \exists$.

If the function symbols have the right type structure, the Herbrand universe is finite and one can instantiate quantifiers in a complete way and solve it by SMT.

- Only works for certain classes of protocols; even quite simple ones like FLASH have arrays of nodes.
- Still the difficult job of *finding* the inductive invariant

Chou-Mannava-Park method

One practical approach that has been used extensively at Intel:

- Method due to Chou, Mannava and Park
- Draws inspiration from McMillan's work
- Made more systematic by Krstic
- Further generalized, extended and applied by Bingham, Talupur, Tuttle and others.

Basic idea of the method

Consider an abstraction of the system as a product of isomorphic finite-state systems parametrized by a *view*, which is a 2-element set of node indices.

Basically, for each pair of nodes $\{i, j\}$, we modify the real system by:

- Using as node indices the two elements i and j plus one additional node Other.
- Conservatively interpreting the transition relation, using Other in place of the 'ignored' nodes.

Too crude to deduce the desired invariant, but it is supplemented with *noninterference lemmas* in an interactive process.

Symmetric between components of the Cartesian product, so only need consider a finite-state system.

III: Floating-point firmware with HOL

Harrison, *A Machine-Checked Theory of Floating Point Arithmetic*, TPHOLs 1999.

Harrison, *Formal verification of IA-64 division algorithms*, TPHOLs 2000.

Harrison, *Formal verification of floating point trigonometric functions*, FMCAD 2000.

Harrison, *Floating-Point Verification using Theorem Proving*, SFM summer school 2006.

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.

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.

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.

Conclusions

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.

What's missing?

- Hardware verification proofs use STE as the workhorse, but sometimes want greater theorem-proving power than the current framework provides.
- The CMP method uses model checking with an ad hoc program for doing the abstraction and the successive refinements, not formally proved correct.
- The high-level HOL verifications assumes the correctness of the basic FP operations, but this is not the same as the low-level specs used in the hardware verification.

What's missing?

- Hardware verification proofs use STE as the workhorse, but sometimes want greater theorem-proving power than the current framework provides.
- The CMP method uses model checking with an ad hoc program for doing the abstraction and the successive refinements, not formally proved correct.
- The high-level HOL verifications assumes the correctness of the basic FP operations, but this is not the same as the low-level specs used in the hardware verification.

All in all, Intel has achieved a lot in the field of FV, but we could achieve even more with a completely seamless combination of all our favorite techniques!

The End