Formal verification of square root algorithms

John Harrison
Intel Corporation, JF1-13
2111 NE 25th Avenue
Hillsboro, OR 97124, USA

2 October 2002

Abstract. We discuss the formal verification of some low-level mathematical software for the Intel® Itanium® architecture. A number of important algorithms have been proven correct using the HOL Light theorem prover. After briefly surveying some of our formal verification work, we discuss in more detail the verification of a square root algorithm, which helps to illustrate why some features of HOL Light, in particular programmability, make it especially suitable for these applications.

1. Overview

The Intel® Itanium® architecture is a new 64-bit architecture jointly developed by Intel and Hewlett-Packard, implemented in the Itanium® processor family (IPF). Among the software supplied by Intel to support IPF processors are some optimized mathematical functions to supplement or replace less efficient generic libraries. Naturally, the correctness of the algorithms used in such software is always a major concern. This is particularly so for division, square root and certain transcendental function kernels, which are intimately tied to the basic architecture. First, in IA-32 compatibility mode, these algorithms are used by hardware instructions like `fptan` and `fdiv`. And while in "native" mode, division and square root are implemented in software, typical users are likely to see them as part of the basic architecture.

The formal verification of some of the division algorithms is described by Harrison (2000b), and a representative verification of a transcendental function by Harrison (2000a). In this paper we complete the picture by considering a square root algorithm. Division, transcendental functions and square roots all have quite distinctive features and their formal verifications differ widely from each other. The present proofs have a number of interesting features, and show how important some theorem prover features — in particular programmability — are.

The formal verifications are conducted using the freely available[1] HOL Light prover (Harrison, 1996). HOL Light is a version of HOL (Gordon and Melham, 1993), itself a descendent of Edinburgh LCF

---

[1] See `http://www.cl.cam.ac.uk/users/jrh/hol-light/index.html`.

(Gordon et al., 1979) which first defined the 'LCF approach' that these systems take to formal proof.

LCF provers like HOL explicitly generate proofs in terms of extremely low-level primitive inferences, in order to provide a high level of assurance that the proofs are valid. HOL's foundational approach to proof is maintained in the formalization of the underlying mathematics including natural number arithmetic and real analysis (Harrison, 1998). Rather than being axiomatized, these structures are constructed starting just with a few basic set-theoretic axioms such as the Axiom of Infinity. Thus, formalization of a proof in HOL achieves a high standard of formal precision, making errors much less likely than in a hand proof or one with more *ad hoc* machine support.

HOL Light allows the user to implement higher-level logical inference rules by programming them in the interaction and implementation language CAML Light (Cousineau and Mauny, 1998), using an abstract type of theorems to protect against arbitrary inferences. Thus, proofs can be partially automated, so although in some respects the formalization of a proof in HOL is painfully exacting, it can *sometimes* be easier than by hand because tedious parts can be dealt with automatically.

HOL notation is generally close to traditional logical and mathematical notation. However, the type system distinguishes natural numbers and real numbers, and maps between them by `&`; hence `&2` is the real number 2. The multiplicative inverse x^{-1} is written `inv(x)`, the absolute value $|x|$ as `abs(x)` and the power x^n as `x pow n`.

2. Square root algorithms based on `fma`

The centerpiece of the Intel® Itanium® floating-point architecture is the `fma` (floating-point multiply-add or fused multiply-accumulate) family of instructions. Given three floating-point numbers x, y and z, these can compute $x \cdot y \pm z$ as an atomic operation, with the final result rounded as usual according to the IEEE (1985) Standard 754 for Binary Floating-Point Arithmetic, but without intermediate rounding of the product $x \cdot y$. Of course, one can always obtain the usual addition and multiplication operations as the special cases $x \cdot 1 + y$ and $x \cdot y + 0$.

The `fma` has many applications in typical floating-point codes, where it can often improve accuracy and/or performance. In particular (Markstein, 1990) correctly rounded quotients and square roots can be computed by fairly short sequences of `fma`s, obviating the need for dedicated instructions. Besides enabling compilers and assembly language programmers to make special optimizations, deferring these operations to software often yields much higher throughput than with typical hardware implementations. Moreover, the floating-point unit becomes simpler and easier to optimize because minimal hardware need be dedicated to these relatively infrequent operations, and scheduling does not have to cope with their exceptionally high latency.

Itanium® architecture compilers for high-level languages will typically translate division or square root operations into appropriate sequences of machine instructions. Which sequence is used depends (i) on the required precision and (ii) whether one wishes to minimize latency or maximize throughput. For concreteness, we will focus on a particular algorithm for calculating square roots in double-extended precision (64-bit precision and 15-bit exponent field):

1. $y_0 = \mathtt{frsqrta}(a)$
2. $H_0 = \frac{1}{2}y_0 \qquad S_0 = ay_0$
3. $d_0 = \frac{1}{2} - S_0H_0$
4. $H_1 = H_0 + d_0H_0 \qquad S_1 = S_0 + d_0S_0$
5. $d_1 = \frac{1}{2} - S_1H_1$
6. $H_2 = H_1 + d_1H_1 \qquad S_2 = S_1 + d_1S_1$
7. $d_2 = \frac{1}{2} - S_2H_2 \qquad e_2 = a - S_2S_2$
8. $H_3 = H_2 + d_2H_2 \qquad S_3 = S_2 + e_2H_2$
9. $e_3 = a - S_3S_3$
10. $S = S_3 + e_3H_3$

All operations but the last are done using the register floating-point format with rounding to nearest and with all exceptions disabled. (This format provides the same 64-bit precision as the target format but has a greater exponent range, allowing us to avoid intermediate overflow or underflow.) The final operation is done in double-extended precision using whatever rounding mode is currently selected by the user.

This algorithm is a non-trivial example in two senses. Since it is designed for the maximum precision supported in hardware (64 bits), greater precision cannot be exploited in intermediate calculations and so a very careful analysis is necessary to ensure correct rounding. Moreover, it is hardly feasible to test such an algorithm exhaustively, even if an accurate and fast reference were available, since there are about 2^{80} possible inputs. (By contrast, one could certainly verify single-precision and conceivably verify double precision by exhaustive or quasi-exhaustive methods.)

3. Algorithm verification

It's useful to divide the algorithm into three parts, and our discussion of the correctness proof will follow this separation:

1 Form[2] an initial approximation $y_0 = \frac{1}{\sqrt{a}}(1+\epsilon)$ with $|\epsilon| \leq 2^{-8.8}$.

2–8 Convert this to approximations $H_0 \approx \frac{1}{2\sqrt{a}}$ and $S_0 \approx \sqrt{a}$, then successively refine these to much better approximations H_3 and S_3 using Goldschmidt (1964) iteration (a Newton-Raphson variant).

9–10 Use these accurate approximations to produce the square root S correctly rounded according to the current rounding mode, setting IEEE flags or triggering exceptions as appropriate.

3.1. INITIAL APPROXIMATION

The `frsrta` instruction makes a number of initial checks for special cases that are dealt with separately, and if necessary normalizes the input number. It then uses a simple table lookup to provide the approximation. The algorithm and table used are precisely specified in the Itanium® instruction set architecture. The formal verification is essentially some routine algebraic manipulations for exponent scaling, then a 256-way case split followed by numerical calculation. The following HOL theorem concerns the correctness of the core table lookup:

```
|- normal a ∧ &0 <= Val a
   ⟹ abs(Val(frsqrta a) / inv(sqrt(Val a)) - &1)
          < &303 / &138050
```

3.2. REFINEMENT

Each `fma` operation will incur a rounding error, but we can easily find a mathematically convenient (though by no means optimally sharp) bound for the relative error induced by rounding. The key principle is the '$1+e$' property, which states that the rounded result involves only a small relative perturbation to the exact result. In HOL the formal statement is as follows:

```
|- ¬(losing fmt rc x) ∧ ¬(precision fmt = 0)
   ⟹ ∃e. abs(e) <= mu rc / &2 pow (precision fmt - 1) ∧
         (round fmt rc x = x * (&1 + e))
```

---

[2] Using `frsqrta`, the only Itanium® instruction specially intended to support square root. In the present discussion we abstract somewhat from the actual machine instruction, and ignore exceptional cases like $a = 0$ where it takes special action.

The bound on e depends on the precision of the floating-point format and the rounding mode; for round-to-nearest mode, `mu rc` is 1/2. The theorem has two side conditions, one being a nontriviality hypothesis, and the other an assertion that the value x does not *lose precision*. We will not show the formal definition (Harrison, 1999) here, since it is rather complicated. However, a simple and usually adequate sufficient condition is that the exact result lies in the normal range (or is zero).

Actually applying this theorem, and then bounding the various error terms, would be quite tedious if done by hand. We have programmed some special derived rules in HOL to help us. First, these automatically bound absolute magnitudes of quantities, essentially by using the triangle rule $|x + y| \leq |x| + |y|$. This usually allows us to show that no overflow occurs. However, to apply the $1 + e$ theorem, we also need to exclude underflow, and so must establish *minimum* (nonzero) absolute magnitudes. This is also largely done automatically by HOL, repeatedly using theorems for the minimum nonzero magnitude that can result from an individual operation. For example, if $2^e \leq |a|$, then either $a + b \cdot c$ is exactly zero or $2^{e-2p} \leq |a + b \cdot c|$ where p is the precision of the floating-point format containing a, b and c.

It's now quite easy with a combination of automatic error bounding and some manual algebraic rearrangement to obtain quite good relative error bounds for the main computed quantities. In fact, in the early iterations, the rounding errors incurred are insignificant in comparison with the approximation errors in the H_i and S_i. Thus, the relative errors in these quantities are roughly in step. If we write

$$H_i \approx \frac{1}{2\sqrt{a}}(1 + \epsilon_i) \qquad S_i \approx \sqrt{a}(1 + \epsilon_i)$$

then

$$d_i \approx \frac{1}{2} - S_i H_i = \frac{1}{2} - \frac{1}{2}(1 + \epsilon_i)^2 = -(\epsilon_i + \epsilon_i^2/2)$$

Consequently, correcting the current approximations in the manner indicated will approximately square the relative error, e.g.

$$S_{i+1} \approx S_i + d_i S_i = S_i(1 + d_i) \approx \sqrt{a}(1 + \epsilon_i)(1 - \epsilon_i - \epsilon_i^2/2) = \sqrt{a}(1 - \frac{3}{2}\epsilon_i^2)$$

Towards the end, the rounding errors in S_i and H_i become more significantly decoupled and for the penultimate iteration we use a slightly different refinement for S_3.

$$e_2 \approx a - S_2 S_2 = a - (\sqrt{a}(1 + \epsilon_2)^2) \approx -2a\epsilon_2$$

and so:

$$S_2 + e_2 H_2 \approx \sqrt{a}(1+\epsilon_2) - (2a\epsilon_2)(\frac{1}{2\sqrt{a}}(1+\epsilon_2')) \approx \sqrt{a}(1-\epsilon_2\epsilon_2')$$

Thus, $S_2 + e_2H_2$ will be quite an accurate square root approximation. In fact the HOL proof yields $S_2 + e_2H_2 = \sqrt{a}(1+\epsilon)$ with $|\epsilon| \leq 5579/2^{79} \approx 2^{-66.5}$.

The above sketch elides what in the HOL proofs is a detailed bound on the rounding error. However this only really becomes significant when S_3 is rounded; this may in itself contribute a relative error of order 2^{-64}, significantly more than the error before rounding. Nevertheless it is important to note that if $\sqrt{a}$ happens to be an exact floating-point number (e.g. $\sqrt{1.5625} = 1.25$), S_3 will be that number. This is a consequence of the fact that the error in $S_2 + e_2H_2$ is less than half the distance between surrounding floating-point numbers.

3.3. CORRECT ROUNDING

The final two steps of the algorithm simply repeat the previous iteration for S_3 and the basic error analysis is the same. The difficulty is in passing from a relative error before rounding to correct rounding afterwards. Again we consider the final rounding separately, so S is the result of rounding the exact value $S^* = S_3 + e_3H_3$. The error analysis indicates that $S^* = \sqrt{a}(1+\epsilon)$ for some $|\epsilon| \leq 25219/2^{140} \approx 2^{-125.37}$. The final result S will, by the basic property of the `fma` operation, be the result of rounding S^* in whatever the chosen rounding mode may be. The *desired* result would be the result of rounding exactly $\sqrt{a}$ in the same way. How can we be sure these are the same?

First we can dispose of some special cases. We noted earlier that if $\sqrt{a}$ is already exactly a floating-point number, then S_3 will already be that number. In this case we will have $e_3 = 0$ and so $S^* = S_3$. Whatever the rounding mode, rounding a number already in the format concerned will give that number itself:

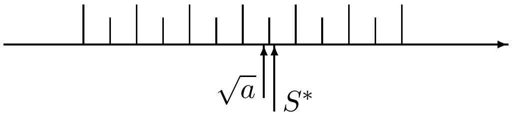
```
|- a IN iformat fmt ==> (round fmt rc a = a)
```

so the result will be correct. Moreover, the earlier observation extends to show that if $\sqrt{a}$ is fairly close (in a precise sense) to a floating-point number, then S_3 will be that number. It is then quite straightforward to see that the overall algorithm will be accurate without great subtlety: we just need the fact that e_3 has the right sign and roughly the correct magnitude, so S^* will never misround in directed rounding modes. Thus, we can also deal immediately with what would otherwise be difficult cases for the directed rounding modes, and concentrate our efforts on rounding to nearest.

On general grounds we note that $\sqrt{a}$ *cannot* be exactly the mid-point between two floating-point numbers. This is not hard to see, since the square root of a number in a given format cannot denormalize in that format, and a non-denormal midpoint has $p+1$ significant digits, so its square must have more than p.[3]

```
|- &0 <= a ∧ a IN iformat fmt ∧ b IN midpoints fmt
   ⟹ ¬(sqrt a = b)
```

This is a useful observation. We'll never be in the tricky case where there are two equally close floating-point numbers (resolved by the 'round to even' rule.) So in round-to-nearest, S^* and $\sqrt{a}$ could only round in different ways if there were a midpoint between them, for only then could the closest floating-point numbers to them differ. For example in the following diagram where large lines indicate floating-point numbers and smaller ones represent midpoints, $\sqrt{a}$ would round 'down' while S^* would round 'up':[4]



Although analyzing this condition combinatorially would be complicated, there is a much simpler sufficient condition. One can easily see that it would suffice to show that for any midpoint m:

$$|\sqrt{a} - S^*| < |\sqrt{a} - m|$$

In that case $\sqrt{a}$ and S^* couldn't 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)
```

---

[3] An analogous result holds for quotients but here the denormal case must be dealt with specially. For example $2^{E_{min}} \times 0.111\cdots 111/2$ is exactly a midpoint.

[4] Similarly, in the other rounding modes, misrounding could only occur if $\sqrt{a}$ and S^* are separated by a floating-point number. However as we have noted one can deal with those cases more directly.

One can arrive at an 'exclusion zone' theorem giving the minimum possible $|\sqrt{a} - m|$. However, this can be quite small, about $2^{-(2p+3)}$ relative to $\sqrt{a}$, where p is the precision. For example, in our context with $p = 64$, consider the square root of the next floating-point number below 1, whose mantissa consists entirely of 1s. Its square root is about 2^{-131} from a midpoint:

$$\sqrt{1 - 2^{-64}} \approx (1 - 2^{65}) - 2^{-131}$$

Therefore, our relative error in S^* of about $2^{-125.37}$ is far from adequate to justify perfect rounding based on the simple 'exclusion zone' theorem, for which we need something of order 2^{-131}. However, our relative error bounds are far from sharp, and it seems quite plausible that the algorithm does nevertheless work correctly. What can we do?

One solution is to utilize more refined theorems (Markstein, 2000), but this is complicated and may still fail to justify several algorithms that are intuitively believed to work correctly. An ingenious alternative developed by Cornea-Hasegan (1998) is to observe that there are relatively few cases like $0.111 \cdots 1111$ whose square roots come close enough to render the exclusion zone theorem inapplicable, and these can be isolated by fairly straightforward number-theoretic methods. We can therefore:

- Isolate the special cases $a_1, \ldots, a_n$ that have square roots within the critical distance of a midpoint.
- Conclude from the simple exclusion zone theorem that the algorithm will give correct results except possibly for $a_1, \ldots, a_n$.
- Explicitly show that the algorithm is correct for the $a_1, \ldots, a_n$, (effectively by running it on those inputs).

This two-part approach is perhaps a little unusual, but not unknown even in pure mathematics.[5] For example, consider "Bertrand's Conjecture" (first proved by Chebyshev), stating that for any positive integer n there is a prime p with $n \leq p \leq 2n$. The most popular proof, originally due to Erdös (1930), involves assuming $n > 4000$ for the main proof and separately checking the assertion for $n \leq 4000$.[6]

By some straightforward mathematics described by Cornea-Hasegan (1998) and formalized in HOL without difficulty, one can show that the

---

[5] A more extreme case is the 4-color theorem, whose proof relies on extensive (computer-assisted) checking of special cases (Appel and Haken, 1976).

[6] An 'optimized' way of checking, referred to by Aigner and Ziegler (2001) as "Landau's trick", is to verify that 3, 5, 7, 13, 23, 43, 83, 163, 317, 631, 1259, 2503 and 4001 are all prime and each is less than twice its predecessor.

difficult cases for square roots have mantissas m, considered as p-bit integers, such that one of the following diophantine equations has a solution k for some integer $|d| \leq D$, where D is roughly the factor by which the guaranteed relative error is excessive:

$$2^{p+2}m = k^2 + d \qquad 2^{p+1}m = k^2 + d$$

We consider the equations separately for each chosen $|d| \leq D$. 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. It's quite easy to program HOL to enumerate all the solutions of such diophantine equations, returning a disjunctive theorem of the form:

$$\vdash (2^{p+1}m = k^2 + d) \implies (m = n_1) \vee \ldots \vee (m = n_i)$$

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 deduce:

$$2^{24}m = 2k'^2 + 2k' - 3$$

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. For example, we have a formally proved HOL theorem asserting that for any double-extended number a,[7] rounding $\sqrt{a}$ and $\sqrt{a}(1+\epsilon)$ to double-extended precision using any of the four IEEE rounding modes will give the same results provided $|\epsilon| < 31/2^{131}$, with the possible exceptions of $2^{2e}m$ for:

$$\begin{aligned} m \in \{ & 10074057467468575321, 10376293541461622781, \\ & 10376293541461622787, 11307741603771905196, \\ & 13812780109330227882, 14928119304823191698, \\ & 16640932189858196938, 18446744073709551611, \\ & 18446744073709551612, 18446744073709551613, \\ & 18446744073709551614, 18446744073709551615\} \end{aligned}$$

and $2^{2e+1}m$ for

---

[7] Note that there is more subtlety required when using such a result in a mixed-precision environment. For example, to obtain a single-precision result for a double-precision input, an algorithm that suffices for single-precision inputs may not be adequate even though the final precision is the same.

$$m \in \{\, 9223372036854775809, 9223372036854775811,\\ 11168682418930654643\}$$

Note that while some of these numbers are obvious special cases like $2^{64} - 1$, the "pattern" in others is only apparent from the kind of mathematical analysis we have undertaken here. They aren't likely to be exercised by random testing, or testing of plausible special cases.[8]

Checking formally that the algorithm works on the special cases can also be automated, by applying theorems on the uniqueness of rounding to the concrete numbers computed. (For a formal proof, it is not sufficient to separately test the implemented algorithm, since such a result has no formal status.) In order to avoid trying all possible even or odd exponents for the various significands, we exploit some results on invariance of the rounding and arithmetic involved in the algorithm under systematic scaling by 2^{2k}, doing a simple form of symbolic simulation by formal proof.

3.4. FLAG SETTINGS

Correctness according to the IEEE Standard 754 not only requires the correctly rounded result, but the correct setting of flags or triggering of exceptions for conditions like overflow, underflow and inexactness. Actually, almost all these properties follow directly from the arguments leading to perfect rounding. For example, the mere fact that two real numbers round equivalently *in all rounding modes* implies that one is exact iff the other is:

```
|- ¬(precision fmt = 0) ∧
   (∀rc. round fmt rc x = round fmt rc y)
   ⟹ ∀rc. (round fmt rc x = x) = (round fmt rc y = y)
```

The correctness of other flag settings follows in the same sort of way, with underflow only slightly more complicated (Harrison, 1999).

4. Conclusions and related work

What do we gain from developing these proofs formally in a theorem prover, compared with a detailed hand proof? We see two main benefits: reliability and re-usability.

---

[8] On the other hand, we can well consider the mathematical analysis as a *source* of good test cases.

Proofs of this nature, large parts of which involve intricate but routine error bounding and the exhaustive solution of Diophantine equations, are very tedious and error-prone to do by hand. In practice, one would do better to use *some* kind of machine assistance, such as *ad hoc* programs to solve the Diophantine equations and check the special cases so derived. Although this can be helpful, it can also create new dangers of incorrectly implemented helper programs and transcription errors when passing results between 'hand' and 'machine' portions of the proof. By contrast, we perform all steps of the proof in a painstakingly foundational system, and can be quite confident that no errors have been introduced. The proof proceeds according to strict logical deduction, all the way from the underlying pure mathematics up to the symbolic "execution" of the algorithm in special cases.

Although we have only discussed one particular example, many algorithms with a similar format have been developed for use in systems based on the Itanium® architecture. One of the benefits of implementing division and square root in software is that different algorithms can be substituted depending on the detailed accuracy and performance requirements of the application. Not only are different (faster) algorithms provided for IEEE single and double precision operations, but algorithms often have two versions, one optimized for minimum latency and one for maximal throughput. These algorithms are all quite similar in structure and large parts of the correctness proofs use the same ideas. By performing these proofs in a programmable theorem prover like HOL, we are able to achieve high re-use of results, just tweaking a few details each time. Often, we can produce a complete formal proof of a new algorithm in just a day. For a even more rigidly stereotyped class of algorithms, one could quite practically implement a totally automatic verification rule in HOL.

Underlying these advantages are three essential theorem prover features: soundness, programmability and executability. HOL scores highly on these points. It is implemented in a highly foundational style and does not rely on the correctness of very complex code. It is freely programmable, since it is embedded in a full programming language. In particular, one can program it to perform various kinds of computation and symbolic execution by proof. The main disadvantage is that proofs can sometimes take a long time to run, precisely because they *always* decompose to low-level primitives. This applies with particular force to some kinds of symbolic execution, where instead of simply accepting an equivalence like $2^{94} + 3 = 13 \cdot 19 \cdot 681943 \cdot 7941336391 \cdot 14807473717$ based, say, on the results of a multiprecision arithmetic package, a detailed formal proof is constructed under the surface. To some extent, this sacrifice of efficiency is a conscious choice when we decide to adopt a highly foundational system, but it might be worth weakening this ideology at least to include concrete arithmetic as an efficient primitive operation.

While formal verification of transcendentals appears never to have been tackled by others, the present work, even when it was done in 1999, was by no means the first formal correctness proof of a square root algorithm of some kind. Both Rusinoff (1998) and O'Leary et al. (1999) describe the formal proof of other square root algorithms used in commercial systems. The former is a multiplicative algorithm based on Goldschmidt iteration, which has some similarity to the present one. Similar algorithms for the IBM Power[9] series are considered by Sawada and Gamboa (2002), who formally verifiy the accuracy of the polynomial approximations, but not the final rounding method (the rounding is not done by standard `fma`s as here). The issue of bounding polynomial approximation error in the context of a formal proof was first identified and solved in a somewhat different way by Harrison (1997) and then in yet another way by Harrison (2000a).

In summary, complete or partial proofs of correctness have been performed for the software algorithms for the Intel® Itanium® architecture, the hardware of at least one Intel and one AMD processor family, and for the latest incarnation of the IBM Power series. There may be other activity elsewhere that has not been reported in the literature or of which the present author is unaware. This already gives some indication of the perceived importance of formal verification in this domain, and can be considered something of a success in itself.

References

Aigner, M. and G. M. Ziegler: 2001, *Proofs from The Book*. Springer-Verlag, 2nd edition.

Appel, K. and W. Haken: 1976, 'Every planar map is four colorable'. *Bulletin of the American Mathematical Society* **82**, 711–712.

Cornea-Hasegan, M.: 1998, 'Proving the IEEE Correctness of Iterative Floating-Point Square Root, Divide and Remainder Algorithms'. *Intel Technology Journal* **1998-Q2**, 1–11. Available on the Web as `http://developer.intel.com/technology/itj/q21998/articles/art_3.htm`.

Cousineau, G. and M. Mauny: 1998, *The Functional Approach to Programming*. Cambridge University Press.

Erdös, P.: 1930, 'Beweis eines Satzes von Tschebyshev'. *Acta Scientiarum Mathematicarum (Szeged)* **5**, 194–198.

Goldschmidt, R. E.: 1964, 'Applications of division by convergence'. Master's thesis, Dept. of Electrical Engineering, MIT, Cambridge, Mass.

---

[9] All other brands are properties of their respective owners

Gordon, M. J. C. and T. F. Melham: 1993, *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press.

Gordon, M. J. C., R. Milner, and C. P. Wadsworth: 1979, *Edinburgh LCF: A Mechanised Logic of Computation*, Vol. 78 of *Lecture Notes in Computer Science*. Springer-Verlag.

Harrison, J.: 1996, 'HOL Light: A Tutorial Introduction'. In: M. Srivas and A. Camilleri (eds.): *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, Vol. 1166 of *Lecture Notes in Computer Science*. pp. 265–269.

Harrison, J.: 1997, 'Verifying the accuracy of polynomial approximations in HOL'. In: E. L. Gunter and A. Felty (eds.): *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs'97*, Vol. 1275 of *Lecture Notes in Computer Science*. Murray Hill, NJ, pp. 137–152.

Harrison, J.: 1998, *Theorem Proving with the Real Numbers*. Springer-Verlag. Revised version of author's PhD thesis.

Harrison, J.: 1999, 'A Machine-Checked Theory of Floating Point Arithmetic'. In: Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry (eds.): *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, Vol. 1690 of *Lecture Notes in Computer Science*. Nice, France, pp. 113–130.

Harrison, J.: 2000a, 'Formal verification of floating point trigonometric functions'. In: W. A. Hunt and S. D. Johnson (eds.): *Formal Methods in Computer-Aided Design: Third International Conference FMCAD 2000*, Vol. 1954 of *Lecture Notes in Computer Science*. pp. 217–233.

Harrison, J.: 2000b, 'Formal verification of IA-64 division algorithms'. In: M. Aagaard and J. Harrison (eds.): *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, Vol. 1869 of *Lecture Notes in Computer Science*. pp. 234–251.

IEEE: 1985, 'Standard for Binary Floating Point Arithmetic'. ANSI/IEEE Standard 754-1985, The Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA.

Markstein, P.: 2000, *IA-64 and Elementary Functions: Speed and Precision*. Prentice-Hall.

Markstein, P. W.: 1990, 'Computation of elementary functions on the IBM RISC System/6000 processor'. *IBM Journal of Research and Development* **34**, 111–119.

O'Leary, J., X. Zhao, R. Gerth, and C.-J. H. Seger: 1999, 'Formally Verifying IEEE Compliance of Floating-Point Hardware'. *Intel Technology Journal* **1999-Q1**, 1–14. Available on the Web as `http://developer.intel.com/technology/itj/q11999/articles/art_5.htm`.

Rusinoff, D.: 1998, 'A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7 Floating-Point Multiplication, Division, and Square Root Instructions'. *LMS Journal of Computation and Mathematics* **1**, 148–200. Available on the Web via `http://www.onr.com/user/russ/david/k7-div-sqrt.html`.

Sawada, J. and R. Gamboa: 2002, 'Mechanical Verification of a Square Root Algorithms using Taylor's Theorem'. In: M. Aagaard and J. O'Leary (eds.): *Formal Methods in Computer-Aided Design: Fourth International Conference FMCAD 2002*, Vol. 2517 of *Lecture Notes in Computer Science*.