Floating-point verification

John Harrison

Intel Corporation, JF1-13
Hillsboro OR 97124
johnh@ichips.intel.com

1 Introduction

Only in a few isolated safety-critical niches of the software industry (e.g. avionics) is any kind of formal verification widespread. But in the hardware industry, formal verification is widely practised, and increasingly seen as necessary. We can perhaps identify at least three reasons:

- Hardware is designed in a more modular way than most software, with refinement an important design method. Constraints of interconnect layering and timing means that one cannot really design 'spaghetti hardware'.
- More proofs in the hardware domain can be largely automated, reducing the need for intensive interaction by a human expert with the mechanical theorem-proving system.
- The potential consequences of a hardware error are greater, since such errors often cannot be patched or worked around, and may *in extremis* necessitate a hardware replacement.

To emphasize the last point, an error in the FDIV (floating-point division) instruction of some early Intel® Pentium® processors in 1994 resulted in a charge to Intel of approximately $475M. Given this salutary lesson, and the size and diversity of its market, it's therefore understandable that Intel should be particularly interested in formal verification.

Moreover, it is not surprising that a considerable amount of effort has been in the floating-point domain, not just at Intel [17, 10, 9], but also at AMD [15, 19] and IBM [9]. Floating-point algorithms have proven themselves difficult to get right. Yet in marked contrast to some other targets for formal verification, it is not hard to come up with widely accepted formal specifications of how floating-point operations *should* behave. In fact, many operations are specified almost completely by the IEEE Standard governing binary floating-point arithmetic [12]. However, in some other respects, floating-point operations present a difficult challenge for formal verification.

2 The role of theorem proving

In many other areas of verification, significant success has been achieved using highly automated techniques, usually based on a Boolean model of the state of the system.

For example, efficient algorithms for propositional logic [1,5,21] and their aggressively efficient implementation [16] have made possible a variety of techniques ranging from simple Boolean equivalence checking of combinational circuits to more advanced model checking of sequential systems [3,18,2,20].

But it is less easy to verify non-trivial floating-point arithmetic operations using such techniques. The natural specifications, including the IEEE Standard, are based on real numbers, not bit-strings. While simple adders and multipliers can be specified quite naturally in Boolean terms, this becomes progressively more difficult when one considers division and square root, and seems quite impractical for transcendental functions. So while model checkers and similar tools are of great value in dealing with low-level details, at least some parts of the proof must be constructed in general theorem proving systems that enable one to talk about high-level mathematics.

There are many theorem proving programs,[1] and quite a few have been applied to floating-point verification, including at least ACL2, Coq, HOL Light and PVS. We will concentrate later on our own work using HOL Light [6], but this is not meant to disparage other important work being done at Intel and elsewhere in other systems.

3 Examples

We will now give a brief overview of some of our verification projects using HOL Light. Of course, a significant component is the formalization of background theories of pure mathematics [7] and floating-point arithmetic [8]. We will not dwell on that in much detail, but it is an essential prerequisite for the verifications that are described.

Division

The Intel® Itanium® architecture performs division in software or microcode using sequences of 'fused multiply-adds', an approach pioneered by Markstein [14]. There are numerous variants depending on the required performance and accuracy characteristics (e.g. IEEE double-precision division with maximum throughput), and quite a few recommended sequences are made available by Intel so that they can be inlined by compilers, used as the core of mathematical libraries, or called on as macros by assembly language programmers. We have verified a large number of such algorithms [10], giving a much higher degree of assurance than had been provided by earlier hand proofs. A particularly gratifying experience was that as part of the process of formalization we observed that one of the hypotheses in a key theorem of [14] was stronger than necessary. As a result, we were able to design some more efficient algorithms [13].

Square root

Similarly, the Intel® Itanium® architecture defers square roots to software, and we have verified a number of sequences for the operation [11]. The process of formal verification

---

[1] See `http://www.cs.ru.nl/~freek/digimath/index.html` for a list, and `http://www.cs.ru.nl/~freek/comparison/index.html` for a comparison of the formalization of an elementary mathematical theorem in several.

follows a methodology established by Cornea [4]. A general analytical proof covers the majority of cases, but a number of potential exceptions are isolated using number-theoretic techniques and dealt with using an explicit case analysis.

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 HOL Light, and can be quite confident that no errors have been introduced.

Transcendentals

We have also proven rigorous error bounds for implementations of several common transcendental functions [9]. It is here that we really start to see the need for non-trivial mathematics. This proof, for example, involves verifying the accuracy of polynomial approximations to transcendental functions (optimal Remez polynomials rather than simply truncated Taylor series), precisely bounding rounding errors in sophisticated floating-point computations, and even diophantine approximation theory in order to deal with difficult cases where the input number is close to a multiple of $\pi/2$.

4 Conclusions

Formal verification in this area is a good target for theorem proving. The work outlined here has contributed in several ways: bugs have been found, potential optimizations have been uncovered, and the general level of confidence and intellectual grasp has been raised. In particular, two key strengths of HOL Light are important: (i) available library of formalized real analysis, and (ii) programmability of special-purpose inference rules without compromising soundness. Subsequent improvements might focus on integrating the verification more tightly into the design flow as in [17].

References

1. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.
2. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.
3. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, 1981. Springer-Verlag.
4. M. Cornea-Hasegan. Proving the IEEE correctness of iterative floating-point square root, divide and remainder algorithms. *Intel Technology Journal*, 1998-Q2:1–11, 1998. Available on the Web as `http://developer.intel.com/technology/itj/q21998/articles/art_3.htm`.

5. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.
6. J. Harrison. HOL Light: A tutorial introduction. In M. Srivas and A. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.
7. J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998. Revised version of author's PhD thesis.
8. J. Harrison. A machine-checked theory of floating point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, 1999. Springer-Verlag.
9. J. Harrison. Formal verification of floating point trigonometric functions. In W. A. Hunt and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design: Third International Conference FMCAD 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233. Springer-Verlag, 2000.
10. J. Harrison. Formal verification of IA-64 division algorithms. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 234–251. Springer-Verlag, 2000.
11. J. Harrison. Formal verification of square root algorithms. *Formal Methods in System Design*, 22:143–153, 2003.
12. IEEE. 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, 1985.
13. P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Prentice-Hall, 2000.
14. P. W. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34:111–119, 1990.
15. J. S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the $AMD5_K86$ floating-point division program. *IEEE Transactions on Computers*, 47:913–926, 1998.
16. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pages 530–535. ACM Press, 2001.
17. J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, 1999-Q1:1–14, 1999. Available on the Web as `http://developer.intel.com/technology/itj/q11999/articles/art_5.htm`.
18. J. P. Queille and J. Sifakis. Specification and verification of concurrent programs in CESAR. In *Proceedings of the 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 195–220. Springer-Verlag, 1982.
19. D. Rusinoff. 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, 1998. Available on the Web via `http://www.onr.com/user/russ/david/k7-div-sqrt.html`.
20. C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6:147–189, 1995.
21. G. Stålmarck and M. Säflund. Modeling and verifying systems and software in propositional logic. In B. K. Daniels, editor, *Safety of Computer Control Systems, 1990 (SAFECOMP '90)*, pages 31–36, Gatwick, UK, 1990. Pergamon Press.